

Ю.А. КИРЮТЕНКО, В.А. САВЕЛЬЕВ

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ
ПРОГРАММИРОВАНИЕ**

Язык Smalltalk



Москва
«Вузовская книга»
2006

УДК 681.3.06
ББК 32.973.26-018.2
К43

Кирютенко Ю.А., Савельев В.А.

К43 Объектно-ориентированное программирование. Язык Smalltalk / Ю.А. Кирютенко, В.А. Савельев. — М.: Вузовская книга, 2006. — 328 с.: ил.

ISBN 5-9502-0097-7

В учебном пособии обсуждаются базовые понятия объектной идеологии, их реализация и применение в языке Smalltalk. Основу книги составили материалы курсов, читавшихся авторами в Ростовском государственном университете и Донском государственном техническом университете.

Для студентов и преподавателей вузов, практикующих программистов и всех желающих изучить язык Smalltalk и объектно-ориентированное программирование.

УДК 681.3.06
ББК 32.973.26-018.2

ПРЕДИСЛОВИЕ

Чтобы переварить знания, надо поглощать их с аппетитом.

Анатоль Франс

Сядем за стол, на котором стоит компьютер, найдем место для чистых листов бумаги, ручки и начнем пир — пир познания нового. Нам бы хотелось, чтобы чтение нашего учебника вы рассматривали именно так! Очень интересно и необычно то, с чем вам предстоит познакомиться. Более десяти лет назад стали мы изучать тогда новый для нас язык и теперь постараемся рассказать вам о нем, о языке программирования высокого уровня Smalltalk (по-русски — Смолток) — одном из самых удивительных и самых малоизвестных в России языков программирования.

Если быть точным, речь у нас пойдет о системе программирования Смолток, объединяющей язык программирования, обширную библиотеку классов и среду разработки. Программирование в такой системе сводится к расширению библиотеки классов с помощью языка программирования.

Из истории языка Смолток

Как это ни удивительно, Смолток довольно старый язык. Его история длится уже тридцать лет. Он построен вокруг одной идеи — объектно-ориентированного программирования, которая занимает особое место в современном мире. Разработка языка Смолток началась в 70-х годах в группе по проблемам обучения (Learning Research Group) исследовательского центра Херох в Пало Альто (Калифорния, США). Основные концепции, положенные в основу Смолтока, были сформулированы Аланом Кеем (Alan Kay) (см. [10]) во время его работы в университете Юты над проектом Flex и, после перехода в Херох, во время работы над проектом Dynabook. Алан Кей [31] отстаивал позицию, что любая работа с компьютером, в конечном счете, сводится к моделированию тех или иных сторон реального мира или человеческого мышления: «вычисление есть моделирование».

Огромное влияние на конечный результат оказали появившийся в то время язык программирования Симула (Simula), созданный для программирования задач имитационного моделирования, и опыт эксплуатации систем

на основе языков Лисп (Lisp) и SketchPad. В разработке языка активное участие принимали психологи и специалисты по теории обучения.

Предполагалось, что язык Смолток в основном будет служить встроенным языком для программирования на микрокомпьютере Dynabook, предвосхитившем сегодняшние PDA. Поэтому язык должен быть максимально простым (даже для детей) и одновременно мощным, обеспечивающим естественную среду моделирования, то есть быть открытой, легко изменяемой системой с удобным и выразительным интерфейсом пользователя. Он должен развиваться вместе с пользователем, поддерживая, по мере понимания решаемых задач, все более сложные модели.

Реализовать идеи Алана Кея взялась группа разработчиков во главе с Дэном Ингалсом (Dan Ingalls), ведущим разработчиком системы, который создал ее общую архитектуру и графическое ядро, позволившее реализовать пользовательский интерфейс на основе перекрывающихся окон, контекстных (всплывающих) меню и активного использования только что появившихся устройств-указателей (теперь называемых мышью). Созданная им схема выполнения графических операций до сих пор применяется во многих системах без существенных изменений.

Некоторые версии языка (Smalltalk-72, -74, -76, -78) были реализованы для рабочих станций Xerox Alto, Xerox Dorado и Xerox Dolphin. Введенное в версии Smalltalk-76 наследование сделало систему более компактной, а байткоды — переносимыми с одной платформы на другую.

Совершенствование языка продолжалось, и была создана версия Smalltalk-78, которая уже широко использовалась внутри фирмы Xerox. Опыт ее эксплуатации позволил создать Smalltalk-80 — первую версию, выпущенную за пределы фирмы. Система Smalltalk-80 была предложена для изучения и рецензирования таким производителям аппаратных средств, как Apple Computer, Digital Equipment Corporation (DEC), Hewlett-Packard и Textonix, что позволило устранить множество ошибок и привело к появлению в 1983 году версии 1.2, которая стала поставляться как коммерческий продукт. Полное описание системы Smalltalk-80 и ее основной библиотеки классов было опубликовано в книгах [3–5], [18].

Кроме участников рецензирования, лицензию на систему Smalltalk-80 получили несколько крупных компаний. Но для массового разработчика система оставалась недоступной, как из-за политики фирмы Xerox, установившей очень высокие цены на систему Smalltalk-80 и рабочие станции для нее, так и из-за требований самой системы к аппаратным средствам. Ситуация изменилась только после выделения из фирмы Xerox компании ParcPlace, для которой развитие и продвижение языка Смолток стало основным видом деятельности. Этой компанией были созданы улучшенные

реализации Smalltalk-80 (VI 2.2, 2.3, 2.4, 2.5) для самых разных аппаратных платформ, включая Apple Mac, рабочие станции Sun и IBM AT 386+.

После выхода в свет книг [4, 18] фирма Digitalk выпустила *Methods*: компактную версию Смолтока, предназначенную для IBM XT. Эта реализация, развиваясь вместе с персональными компьютерами (Smalltalk/V, Smalltalk/V 286, Smalltalk/V for Windows, *Smalltalk Express*, Visual Smalltalk), долгое время была самой массовой реализацией Смолтока. Были предприняты и другие попытки реализовать Смолток для PC-совместимых машин, но они оказались не столь удачны.

Развитие оконных систем (X Window System, Presentation Manager, Windows) стимулировало ParcPlace к переходу от Smalltalk-80 через ObjectWorks к семейству продуктов VisualWorks, которые использовали встроенные в операционные системы менеджеры окон и были снабжены средствами взаимодействия с другими приложениями. А в 1993 году фирма IBM представила семейство систем программирования VisualAge, ядром которого является VisualAge for Smalltalk. Эта система предназначалась для коллективной визуальной разработки программ в многоплатформенной среде (возможно, включающей в себя «большие» машины и «унаследованные» приложения) и объединила в себе сильные стороны VisualWorks и Visual Smalltalk, что сразу обеспечило ей ведущие позиции на рынке.

Появилось несколько небольших фирм, интенсивно развивающих как коммерческие, так и некоммерческие версии языка Смолток.

Существование множества реализаций смолтоковских систем потребовало стандартизации языка. Летом 1993 года ANSI был сформирован технический комитет X3J20 для создания стандарта языка Смолток, куда вошли представители производителей языка и фирм, активно его использующих. Были предприняты все возможные меры, чтобы стандарту удовлетворяло максимальное количество ранее написанного кода и большинство реализаций. Тем более, что все системы использовали практически одинаковый синтаксис и семантику программных конструкций и чрезвычайно близкие объектные библиотеки (по крайней мере, в пределах классов, описанных в [4]). Принятый в 1998 году стандарт зафиксировал сложившуюся ситуацию, исключив пока из сферы стандартизации все, что касается графики и пользовательского интерфейса.

Долгий процесс созревания языка Смолток в недрах фирмы Хегох и последующее рецензирование позволили создать удивительно естественный, *мощный* и достаточно *простой* язык. Именно так: *мощный* и *простой*. Мы не согласны с теми, кто, на наш взгляд, необоснованно и несправедливо относит язык Смолток к сложным языкам программирования [25]. Как показала мировая практика, язык Смолток может быть отнесен к *непри-*

вычным языкам программирования, поскольку он не «алголоподобен», как большинство современных языков. Однако *непривычность* не есть *сложность*. Сложным может стать освоение огромной библиотеки классов, накопленной языком за время существования. Но великолепные инструменты смолтоковских систем делают это освоение более легким, чем освоение большинства библиотек в традиционных языках программирования. Возможно, миф о сложности языка связан с тем, что Смолток применяется для создания сложных программных систем, и их сложность воспринимается не как свойство предметной области, а как свойство самого языка.

Сегодня Смолток — это современный, продуманный и универсальный язык программирования, в котором объектная идеология реализована с максимальной последовательностью и полнотой. Он содержит минимум ловушек для разработчика и очень стабильный внешний интерфейс основных классов. В смолтоковской системе *все* сущности являются *объектами*, а свойства объектов описываются *объектами-классами*. Все объекты в совокупности составляют *образ* системы, «оживляемый» ее *виртуальной машиной*. Образ системы между сеансами работы сохраняется в *файле образа*, который представляет собой моментальный снимок состояния системы, содержащий переменную ее часть. Виртуальная машина и образ системы составляют минимальную смолтоковскую систему. Именно такую структуру имеют автономные (выделенные из среды разработки) смолтоковские приложения. При работе смолтоковские системы используют архив исходных текстов и файлы записи произведенных изменений (журналы). Конкретный вид архива исходных текстов зависит от реализации: от текстового файла в однопользовательских системах до объектной базы данных с управлением версиями и ограничениями доступа в системах групповой разработки, таких как ENVY или VisualAge for Smalltalk.

Современные реализации языка обеспечивают также перенос и распространение смолтоковских приложений в виде пакетов, содержащих набор новых классов, связанных с существующей библиотекой, вместе с необходимыми ресурсами и со сценариями инсталляции/деинсталляции.

Использование смолтоковских систем

Практически сразу смолтоковские системы заняли не предназначавшееся им место системы обучения программированию, а стали инструментом разработки сложных приложений с развитым пользовательским интерфейсом. Причина этого кроется, прежде всего, в простоте и мощи как языка Смолток, так и среды разработки приложений, которые позволяют вести коллективную работу, быстро реализовывать и эффективно поддерживать

сложные приложения с высокими требованиями к надежности и постоянной готовности к функционированию, эффективно сопровождать созданный программный продукт во время эксплуатации, производить регулярные модификации.

Среди типовых задач, для которых сегодня используется Смолток, можно выделить следующие:

- автоматизация, робототехника;
- обработка коммерческой информации;
- диспетчеризация, планирование;
- системы управления;
- интерфейс пользователя;
- тренажеры, моделирование;
- коммуникации, связь;
- обучение программированию.
- медицина, экспертные системы;

Так в начале 80-х разработчиками из Хегох и ЦРУ была создана большая система Analyst для ситуационного центра. Эта система обеспечивала работу с электронной почтой, совместное редактирование одного документа с нескольких рабочих мест и публикацию с различными уровнями доступа, электронные таблицы, способные содержать в своих ячейках и обрабатывать произвольные объекты системы, поддержку растровой и векторной графики, в том числе анимационной, и обладала встроенной геоинформационной системой. Известны также приложения для системы управления спутниками NASA, созданные в технологическом институте Джорджии.

Заметим, что сама область критически важных приложений такова, что многие компании стремятся не указывать какие-либо данные о своих системах этого класса, но опубликованные данные показывают, что основными языками программирования в этой области являются Кобол, С++ и Смолток, причем в настоящее время доля Кобола постоянно сокращается, и от Кобола переход обычно осуществляется именно к Смолтоку.

С середины 90-х количество пользователей Смолтока устойчиво растет [15, 16]. Оказывается, если организация начала использовать Смолток, то все новые ее разработки будут выполняться именно на нем. Возможно, это следствие того, что Смолток явно имеет *лучшее* отношение количества законченных работ к количеству начатых работ, чем сравнимые с ним по возможностям языки программирования. В отчете [16] приводятся статистические данные, показывающие высокую лояльность программистов, пишущих на языке Смолток, к выбранному языку.

Смолток в России

В то время когда во многих странах мира, начиная с 1983 года, Смолток активно используется, а монографии и учебники по языку издаются постоянно, в России он практически неизвестен. За прошедшие годы на

русском языке вышло всего несколько книг и статей, в которых Смолток только упоминается (см. [24], [25], [28–30], [34], [37]), но по ним совершенно невозможно понять, что он собой представляет и как на нем программировать. Книга, которую вы открыли, как предполагают авторы, должна восполнить этот пробел. Еще одной причиной неизвестности и благодатной почвой для мифа о сложности языка Смолток до последнего времени (пока не появились мощные свободно распространяемые версии) служила высокая стоимость смолтоковских систем.

Но сказать, что Смолток совсем не известен в России, нельзя. В последние годы он стал использоваться, хотя это часто не афишируется. Во многом его применение связано с постепенно расширяющимся использованием продуктов семейства IBM VisualAge. Существовали и существуют разрозненные группы программистов (Москва, Ростов-на-Дону, Таганрог, Новосибирск и др.), работающие на этом языке. В начале 90-х годов в ИПИ РАН группой программистов под руководством кандидата технических наук А.Г. Иванова [29, 30], [35] были разработаны версии «Русского Смолтока» (ТМООП, ГООСП) для MS-DOS, которые по своей структуре и возможностям примыкают к версии Smalltalk/V и представляют системы программирования на языке, подобном Смолтоку, но основанном на русской лексике¹.

В настоящее время интерес к Смолтоку в России растет. Предпринимаются попытки создать российскую группу пользователей языка Смолток. Создаются русские интернет-ресурсы, посвященные Смолтоку, такие как «Смолток по-русски»². Расширяется список вузов, использующих Смолток в обучении программированию и объектным технологиям.

Обзор учебника

Несколько слов о том, что представляет собой данная книга и как следует с ней работать. Прежде всего — это учебник, в котором представлены объектно-ориентированная методология и основы программирования на языке Смолток, а не справочник и не описание стандарта языка. Книга содержит описание основных понятий, принципов, технологий, ознакомившись с которыми вы сможете самостоятельно разбираться в документации, поставляемой с каждой смолтоковской системой.

¹ Учебная версия «Русского Смолтока» в начале 2005 года располагалась на странице (<http://www.math.rsu.ru/smalltalk/russian.ru.html>).

² (<http://www.smalltalk.ru/>)

Те, кто считает себя асом в Си, Паскале, Фортране, Лиспе, должны будут приложить определенные усилия и изменить многие привычные для них воззрения на практику программирования. Тем, кто мало знаком с программированием, будет, как это ни удивительно, легче, поскольку на них не будет давить груз приобретенного опыта. Именно об этом говорит практика преподавания языка Смолток как для школьников, так и для студентов. Но в любой среде учащихся и сам язык, и объектно-ориентированная методология программирования воспринимались намного легче, чем при изучении сравнимых по возможностям языков программирования Java, Lisp/CLOS, Ada или C++.

Читайте разделы внимательно, не торопясь. Не торопитесь начинать программировать, прочитав всего несколько страниц. Из-за непохожести Смолтока на все остальное в мире языков программирования излишняя торопливость чревата полным разочарованием. Вполне возможно, что какие-то моменты не удастся понять сразу — не расстраивайтесь. В действительности, в языке Смолток нет ничего сложного.

Обязательно пытайтесь выполнить задания для самостоятельной работы, это поможет вам быстрее понять особенности языка.

Книга разбита на пять частей. В первой части учебника рассказывается об *основных принципах* объектно-ориентированной методологии программирования и о том, *как* эти принципы реализуются в языке: в его синтаксисе, инструментах, в особенностях функционирования самой смолтоковской среды. Другими словами, излагается некоторая «теория».

Чтобы изложение было более конкретным, мы предполагаем, что на вашем компьютере установлена система *Smalltalk Express*. Наш выбор объясняется тем, что данная реализация распространяется бесплатно, компактна, работает на всех IBM-совместимых компьютерах, в основном соответствует стандарту, содержит все необходимое для *первоначального знакомства и работы*. Среди доступных нам реализаций Смолтока другой системы, удовлетворяющей *всем* этим требованиям, не нашлось, хотя по каждому отдельному показателю или некоторому их подмножеству другие реализации были бы лучше. Разумеется, читатель, работая с книгой, может пользоваться любой смолтоковской системой. Большая часть того, о чем мы будем рассказывать и что не будет касаться графики и интерфейса пользователя, справедлива в любой реализации.

Вторая часть — библиотека классов смолтоковской системы. Даже в достаточно небольшой системе *Smalltalk Express* эту библиотеку нельзя назвать маленькой, а в учебнике невозможно рассмотреть ее полностью. Мы ограничились только наиболее часто используемыми и важными классами и их функциональными возможностями.

В третьей — рассказывается, как программировать в системе Смолток, создавая новые классы. Сначала рассматривается технология построения нового класса системы, поиск и исправление ошибок в создаваемых методах. Темы примеров самые разные. Некоторые из них не предполагают никаких предварительных знаний у читателя, другие предполагают определенные знания по математике. В заключение перечисляются и поясняются некоторые правила и соглашения о стиле программирования в Смолтоке.

Четвертая часть — построение смолтоковских приложений, включающих интерфейс пользователя. По сути, это продолжение предыдущей части. Но излагаемый материал уже не является столь общим, как в предыдущих частях, и сильно опирается на специфику системы *Smalltalk Express*. Хотя проект приложений применим в любой реализации, их код будет сильно от нее зависеть. Поэтому те, кто работает в других системах, должны познакомиться с особенностями используемой ими среды и попытаться самостоятельно реализовать приведенные в этой части приложения.

В приложении мы постарались коротко описать известные нам коммерческие и некоммерческие реализации Смолтока.

Если по прочтении книги что-то останется неясным, или вы обнаружите ошибки, описки, неточности, найдете более изящное решение задачи, или захотите высказать критические замечания, пишите нам. Наши адреса электронной почты указаны ниже. Кроме того, интересную и полезную информацию вы сможете найти и на поддерживаемой авторами учебника интернетовской страничке «Смолток в России»³. На этой же веб-странице размещен и инсталляционный пакет рассматриваемой в книге системы программирования *Smalltalk Express*. Кроме того, там приведены учебные материалы, которые могут оказаться полезными для тех, кто использует другие реализации языка Смолток.

Наши благодарности

Самая большая благодарность — старшему научному сотруднику ИПИ РАН, кандидату технических наук А.Г. Иванову, который обратил наше внимание на язык Смолток, вдохновил на создание учебника, следил и помогал советами во время работы над ним.

Мы благодарим профессоров Я.М. Ерусалимского — декана механико-математического факультета Ростовского госуниверситета, Ю.Ф. Коробейника — заведующего кафедрой математического анализа механико-математического факультета РГУ, С.В. Жака — заведующего кафедрой исследова-

³ (<http://www.math.rsu.ru/smalltalk/>)

ния операций механико-математического факультета РГУ, доцента В.Н. Землянухина — заведующего кафедрой программного обеспечения вычислительной техники и автоматизированных систем факультета «Автоматизация и информатика» Донского государственного технического университета, которые поддержали нашу работу и предоставили возможность проводить эксперименты по преподаванию совершенно нового для этих двух вузов языка программирования. Мы благодарим специализировавшихся у нас студентов этих вузов за помощь в создании учебника.

Выражаем нашу благодарность профессорам Айдыну Айтуна и Бюленту Карасезену (г. Анкара), Тосуну Терзиоглу (г. Стамбул), а также доктору Якову Чернеру (PhD, Глочестер, США), которые предоставили в наше распоряжение оригинальные книги по языку Смолток, и среди них великолепную книгу [5], с перевода которой и началось наше знакомство и увлечение языком⁴.

Отдельная благодарность — доценту РГУ М.Э. Абрамяну и преподавателю Ростовского колледжа связи и информатики Е.Ф. Тарасевич, взявших на себя труд полностью прорецензировать эту книгу. Их замечания позволили сделать текст существенно лучше и избавиться от множества ошибок и неточностей.

Особо признательны и благодарны мы членам наших семей, которые на протяжении четырех лет терпеливо ждали того момента, когда, наконец, все это закончится, и морально поддерживали нас.

С надеждой, что чтение книги будет приятным, доценты РГУ

Ю.А.Кирютенко (jakir@math.rsu.ru)

В.А.Савельев (vasav@math.rsu.ru)

⁴ К сожалению, полностью завершённый ещё семь лет назад перевод этой книги нам так и не удалось издать.

ЧАСТЬ I

ОБЩИЙ ОБЗОР

Вершей пользуются при рыбной ловле. Наловив же рыбы, забывают про вершу. Ловушкой пользуются при ловле зайцев. Поймав же зайца, забывают про ловушку. Словами пользуются для выражения мысли. Обретя же мысль, забывают про слова. Где бы мне отыскать забывшего про слова человека, чтобы с ним поговорить.

Чжуан-цзы

ГЛАВА I

ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ МЕТОДОЛОГИИ ПРОГРАММИРОВАНИЯ

1.1. Объектно-ориентированная методология программирования

Основой любой методологии программирования является то, что практически полезные программы имеют размер и функциональность, необозримые для одного человека, и, как следствие, требуют координированных усилий группы людей, работающих над обозримыми *частями* программ. Различные способы выделения этих частей (*декомпозиции* программы) и описания правил их взаимодействия между собой (их *интерфейсов*) и составляют сущность различных методологий программирования.

В основе объектно-ориентированной методологии программирования (сокращенно — ООМП) лежит подход, согласно которому декомпозиция программы непосредственно отражает структуру прикладной предметной области, в которой выделяются вполне различимые сущности — *объекты*,

обладающие индивидуальностью, проявляемой как их состояние и поведение, а взаимодействие между объектами происходит посредством пересылки *сообщений*.

1.1.1. Понятия и термины

Полоний: Что Вы читаете, милорд?

Гамлет: Слова, слова, слова.

Шекспир. «Гамлет»

Под объектом в ООМП понимается некая сущность (конкретная или абстрактная) предметной области, обладающая *состоянием* и проявляющая четко выраженное *поведение*. Совокупность состояния и поведения позволяет различать объекты и формирует *индивидуальность* объекта. *Состояние* объекта определяется всеми его возможными свойствами (атрибутами, переменными), как правило, статическими, и текущими значениями каждого из этих свойств (обычно динамическими, изменяющимися во время работы программы). Тот факт, что объект характеризуется состоянием, означает, что он занимает определенное пространство (в реальном мире или в памяти компьютера). Свойства объекта характеризуются значениями их параметров. Значением свойства может быть любой другой объект, который строится или запрашивается у программ или аппаратных систем.

Поведение характеризует то, как объект отвечает на внешние воздействия (на посылаемые ему *сообщения*). Он может изменить свое состояние и/или воздействовать на другие объекты с помощью передачи сообщений. Совокупность всех сообщений, понимаемых объектом, называется его *интерфейсом*. Передача сообщений — это способ взаимодействия объектов. Состояние объекта доступно только через его сообщения.

Поведение объекта реализуется в виде подпрограмм, вызываемых объектом в ответ на полученное им сообщение. Такие подпрограммы называются *методами*. Разные объекты могут использовать разные методы для ответа на одно и то же сообщение.

Чтобы использовать преимущества ООМП, надо добиваться, чтобы объекты имели функционально полные и в то же время максимально простые и стабильные интерфейсы. В правильно написанной программе можно в любой момент изменить свойства объекта и реализацию его методов — это не скажется на работе остальных частей программы.

Этот подход позволяет локализовать принимаемые решения рамками объекта, объединяя в нем состояние и поведение, а следовательно, снижая сложность той программы, в которой используется объект. Такое объедине-

ние называется *инкапсуляцией* и позволяет скрыть структуру объекта и его данные, делая их невидимыми для всех, за исключением методов самого объекта. Инкапсуляция позволяет рассматривать объекты как изолированные «черные ящики», которые знают определенные действия и умеют их выполнять, функционируя независимо друг от друга и скрывая за интерфейсом детали реализации. Внутреннее устройство «черных ящиков» недоступно извне. Нам неизвестно, *как* все происходит. Важно только знать, как приказать «ящику» выполнить определенные действия и что мы при этом из него получим. Поэтому объекты в объектно-ориентированных системах можно рассматривать как минимальные (в смысле самодостаточности) единицы инкапсуляции, позволяющие навести должный порядок среди данных и кода, обрабатывающего эти данные.

К объекту может обращаться не только программист, скрывающийся под местоимением «МЫ», но и любой объект системы, посылая нужному объекту *сообщение*, которое представляет предписание (просьбу, приказ, требование) выполнить некоторые действия. Если предписание может быть выполнено получившим сообщение объектом (его называют объектом-получателем, или просто получателем), то оно выполняется и, как результат, объекту-отправителю, пославшему сообщение, может возвращаться некоторый объект (возвращаемый объект). Если по какой-либо причине получатель не может выполнить сообщение, он в какой-то форме информирует об этом пославший сообщение объект (или систему времени выполнения). В других терминах: объект-получатель — это *сервер*, предоставляющий обслуживание *клиенту* — объекту-отправителю.

Таким образом, объектно-ориентированное программирование сводится к моделированию некоторого числа объектов, которые для решения поставленной задачи взаимодействуют друг с другом, посылая сообщения, в ответ на которые выполняют действия, описанные в методах, соответствующих сообщениям. Поскольку все то, что объект может и должен делать, выражено его методами, этот простой механизм поддерживает все возможные взаимодействия между объектами.

Но как управлять таким миром объектов, когда их становится достаточно много? Во-первых, объекты могут сильно отличаться друг от друга, а могут быть очень похожи друг на друга. А во-вторых, часто бывают нужны несколько однотипных объектов, а иногда — только один такой объект. Например, программа, моделирующая движение автотранспорта на уличном перекрестке, может иметь дело с множеством автомобилей всех существующих в мире типов. Так что же, в каждой такой ситуации необходимо определять одни и те же свойства и одни и те же методы в каждом объекте снова и снова?

В объектно-ориентированном программировании для объектов, имеющих общие свойства и поведение, применяется совместное использование общих свойств и поведения. Существуют различные способы организации такого совместного использования. Наиболее распространены следующие два способа:

Классы и их наследование. Выделяются группы объектов, имеющих одинаковое поведение, они объединяются в класс, на который возлагается «хранение» поведения и информации о внутренней структуре объектов. Наследование позволяет разделять между несколькими классами поведение, внутреннюю структуру и, возможно, данные. Так организованы, например, языки Смолток, Java, C++ и Ada 95.

Прототипы. Объекты могут непосредственно наследовать поведение и структуру у своих объектов-прототипов (предков). В каждом новом объекте можно свободно изменять поведение и структуру объекта, но пока это не сделано, объект использует поведение и структуру прототипа. После изменения объект хранит только то, что отличает его от прототипа. Это сравнительно новый способ, положенный в основу языков SELF и JavaScript.

Поскольку наш предмет — Смолток, остановимся подробнее на идее единообразного описания похожих объектов в *классе*. Слово «похожих» означает, что объекты одинаково устроены и действуют. Такая идея впервые была реализована еще в 60-е годы в языке Симула. Класс — это шаблон, описывающий объекты определенной структуры и поведения и хранящий информацию, общую для всех таких объектов. Именно по таким шаблонам и создаются объекты в мире Смолтока. Каждый класс имеет идентифицирующее его имя. Класс через переменные описывает структуру объекта, а через методы — поведение объекта и определяет механизмы создания «своего» объекта — *экземпляра класса*. Таким образом, методы и переменные определяются в классе, в то время как значения переменных определяются в экземпляре, отражая индивидуальность объекта.

Наведение с помощью классов порядка в мире объектов — большое достижение, но можно пойти еще дальше, определяя некоторый порядок и среди самих классов. Достигается это с помощью введения механизма *наследования* — пожалуй, самого мощного средства в любой объектно-ориентированной системе. Именно оно позволяет один раз написать нужный код и многократно его использовать, избегая дублирования.

По своей сути механизм наследования прост: один класс, называемый в рамках этих отношений *суперклассом*, полностью передает другому классу, который называется его *подклассом*, свою структуру и поведение, то есть все свои переменные и все методы. Что дальше делать с «этим богатством»,

определяет подкласс: он может добавить в структуру и поведение что-либо свое, что-то из наследуемого может использовать без изменений, а что-то изменить. Таким образом, класс с помощью подклассов «уточняет» поведение экземпляров, и, как результат, создаваемые объекты становятся более специализированными.

Классы могут наследовать друг у друга без каких-либо ограничений, а наследуемые компоненты будут автоматически накапливаться сверху вниз через все уровни. Классы, расположенные по принципу наследования, начиная с самого общего, базового класса, образуют древовидную структуру, которая называется *иерархией классов*.

Перечислим те новые термины, пользуясь которыми мы начали описание объектно-ориентированной системы программирования: объект, сообщение, метод, класс, экземпляр, суперкласс, подкласс, иерархия, инкапсуляция, наследование. Как мы видели, они тесно связаны друг с другом.

Прежде чем двигаться дальше, отметим, что ООМП ни в коей мере не является отрицанием структурного подхода в программировании. Напротив, ООМП создает основу для применения принципов структурного программирования, так как для реальных задач характерно отсутствие «верхнего уровня», от которого можно строить структурную декомпозицию. Выделение и четкое определение интерфейсов как раз и создает основу для структурной реализации методов. Эту методологию правильнее представлять как инструмент, позволяющий снизить сложность задачи и подойти к созданию таких систем, поведение которых невозможно представить в виде исчерпывающего набора всех возможных ситуаций и разветвлений алгоритма. Сегодня практически доказана возможность создания на основе ООМП проектов высокой степени сложности. Но на характер мышления программиста и дисциплину проектирования программного продукта эта методология, безусловно, накладывает свой отпечаток.

1.1.2. Принципы и компоненты

Путь, по которому можно пройти, —
Это не Вечный Путь.

Имя, которое можно назвать, —
Это не Вечное Имя.

Лао-цзы. «Дао-дэ цзин»

После краткого описания части ключевых понятий можно сделать вывод, что концептуально объектно-ориентированная методология опирается на объектный подход, который включает в себя четыре основных принципа:

Абстрагирование. Выделение объектов и их существенных характеристик, которые отличают их от других видов объектов и четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа. Минимальной единицей абстракции в ООМ является класс.

Ограничение доступа. Защита отдельных элементов объекта от доступа к ним извне, которая, однако, не затрагивает существенных характеристик объекта как целого. Понятие «ограничение доступа» обычно соответствует понятию «защита информации».

Модульность. Свойство системы, связанное с возможностью ее представления в виде нескольких тесно связанных между собой частей (модулей). Модульность опирается на дискретное программирование объектов, представляющих собой минимальные единицы инкапсуляции, которые можно модернизировать или заменять, не воздействуя на другие объекты системы и систему в целом.

Существование иерархий. Иерархия — ранжированная и упорядоченная система абстракций. Наиболее употребительными видами иерархий являются иерархия по типам и иерархия составных частей.

Объектно-ориентированная методология (ООМ) включает в себя

- объектно-ориентированный анализ (Object Oriented Analysis),
- объектно-ориентированное проектирование (Object Oriented Design),
- объектно-ориентированное программирование (Object Oriented Programming).

ООА — методология анализа реального или воображаемого мира на основе понятий класса и объекта, основной задачей которой является понимание и объяснение того, как и какие сущности взаимодействуют между собой в этом мире.

Рассматривая реальную задачу, аналитик разбивает ее на некоторое число предметных областей. Каждая предметная область — мир, населенный специфическими объектами. В предметной области выделяются классы объектов, которые, если это необходимо, могут содержать подклассы.

Модели ООА преобразуются в объектно-ориентированный проект. ООД есть методология проектирования программ (см. [26], [7]), опирающаяся на выделение классов и объектов. Ее фундаментальными понятиями являются:

Инкапсуляция. Концепция сокрытия как бы в «капсуле» всей информации об объекте, то есть объединение в некое целое данных и процедур (методов) их обработки. Единицей инкапсуляции в ООД является объект, в котором содержатся и данные, определяющие состояние объекта, и методы, которые объект может выполнять.

Наследование. Такое отношение между классами, когда один класс (подкласс) повторяет структуру и поведение других классов (суперклассов), добавляя свою специфику.

Полиморфизм. Возможность единообразного обращения к объектам (посылки им одноименных сообщений) при сохранении их уникального поведения. Другими словами, поскольку поведение объектов определяется методами, полиморфизм означает, что методы, ассоциированные с одним и тем же именем, допускают разные реализации в разных классах.

Созданный проект превращается в программный продукт в процессе объектно-ориентированного программирования — методологии, которая основана на представлении программного продукта в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию на принципах наследования. Решение поставленной задачи сводится к построению необходимых классов и управлению создаваемыми ими экземплярами путем передачи сообщений. Для этого необходимо, чтобы объекты определялись вместе с методами, которые они выполняют. Такой стиль сильно отличается от процедурного стиля программирования, когда сначала определяются данные, которые затем передаются в процедуры (функции) как аргументы.

Средством программирования выступает один из объектно-ориентированных языков программирования. Язык программирования называется *объектно-ориентированным*, если:

- есть поддержка объектов как абстрактных данных, имеющих интерфейсную часть в виде поименованных операций и защищенную область локальных данных;
- все объекты относятся к соответствующим типам;
- структура и поведение объектов могут наследоваться.

Объектно-ориентированные языки программирования иногда делят на «чистые» (Смолток, Java, SELF) и «гибридные», выросшие из ранее существовавших процедурных языков (Ада 95, C++). К «чистым» относят те языки, в которых *все* данные хранятся как объекты, размещаемые с автоматическим выделением и освобождением памяти.

На основании сказанного можно сделать первые общие выводы. Традиционно процедурное программное обеспечение рассматривалось как средство заставить компьютер решить конкретную задачу. В результате программа, решающая поставленную задачу, оказывалась совершенно непригодной для решения других задач, даже если в задачах использовались те же самые данные. При объектно-ориентированном подходе к созданию программного обеспечения решение *собирается* из уже существующих или специально создаваемых объектов.

Технология программирования с объектами сохраняет дух построения объектов реального физического мира. Проект объектно-ориентированной системы начинается не с задачи, которую надо решить, а с анализа тех характеристик реального мира, которые должны присутствовать в программной модели, чтобы справиться с поставленной задачей. Как только такие характеристики и их носители — объекты — представлены, модель может использовать их для решения первоначальной задачи, но созданные объекты часто полезны и при решении многих других задач.

Объектно-ориентированный подход к построению систем программного обеспечения помимо гибкости имеет много других и весьма существенных преимуществ. Поскольку структура программного обеспечения отражает реальный или воображаемый мир, он позволяет размышлять о задаче в терминах существующих в этом мире объектов, а не в терминах языка программирования. Такое изменение процесса мышления становится возможным, так как теперь можно определять новые типы данных, описывая с их помощью населяющие мир объекты. В последующем программистам будет много легче понять и изменить построенные таким образом программы, даже если и не они их создавали.

С другой стороны, основные операции, заложенные в программное обеспечение, имеют тенденцию изменяться намного медленнее, чем информационные потребности определенных групп людей или учреждений. Это означает, что программное обеспечение, основанное на общих моделях, будет существовать и успешно работать много дольше, чем то, которое написано для решения определенной, сиюминутной проблемы.

Следовательно, при объектно-ориентированном подходе к решению задачи процесс построения программного обеспечения должен быть совершенно другим. Обычное процедурное программное обеспечение всегда пишется «от печки», с самого начала. Из более ранних программ, решающих точно обозначенные проблемы, повторно используется не так уж и много процедур, так как легче написать похожие процедуры заново, чем преобразовывать уже существующие. Объекты, в противоположность предыдущему, являются строительными блоками «общего назначения», которые моделируют существующие в моделируемом мире сущности, а не просто решают конкретные задачи. Это дает возможность их повторно использовать в последующих проектах, даже если цели новых проектов весьма далеки от первоначальных. Когда собрано большое число классов, усилия по созданию программного обеспечения начинают сдвигаться от создания новых классов к сборке нужных объектов из существующих. Совершенно понятно, что такую конструкцию проще изменять и сопровождать, приспособив к новым требованиям.

И, наконец, еще один важный момент: обширное повторное использование существующих отлаженных объектов не только сокращает сроки создания, но и ведет к более правильным, свободным от ошибок системам.

1.2. Смолтоковская реализация ООМП

1.2.1. Иерархия классов системы Смолток

... Неимоверные иерархии,
Точно сам коронованный Свет.

Даниил Андреев. «Бог»

Посмотрим, как рассмотренные ранее основные понятия реализуются в конкретной объектно-ориентированной системе — в системе *Smalltalk Express*. Начнем с самого важного и «большого» понятия — иерархии классов. В Смолтоке существует ограничение на построение иерархии классов: у каждого класса может быть только один непосредственный суперкласс. Такое наследование называется *одиночным наследованием*. В других объектно-ориентированных языках наследование может быть *множественным* (C++, CLOS), то есть у класса может быть более одного непосредственного суперкласса.

В иерархии, приведенной на рис. 1.1, показаны не все классы этой реализации. Их много больше. Мы привели в основном те, о которых будем, хотя бы кратко, рассказывать в книге. С частью классов вам придется по мере необходимости знакомиться самостоятельно, а с некоторыми существующими классами вы никогда в своей работе не встретитесь: они нужны только для обслуживания внутренних потребностей системы. Многоточие после имени класса говорит о том, что у класса есть подклассы, которые на рисунке не показаны. В правой колонке приведен перевод на русский язык имени соответствующего класса. Но в реальном программировании русские имена классов использовать нельзя.

Все представленные классы являются подклассами класса **Object**¹, в котором определены структура и поведение, общие для всех объектов системы. **Object** является базовым классом иерархии и единственным классом, не имеющим суперкласса. Например, класс **Collection** и класс **Behavior** — подклассы класса **Object**, класс **Bag** — подкласс класса **Collection**. Цепочка класс—суперкласс всегда завершается классом **Object**.

¹ Смолтоковский код далее всегда будет выделен рубленным шрифтом.

Object	Объект
Behavior	Поведение
Class	Класс
MetaClass	МетаКласс
Boolean...	Логический...
Collection	Набор
Bag	ПростойНабор
IndexedCollection...	ИндексированныйНабор...
Set...	Множество...
Compiler	Компилятор
Context...	Контекст...
CursorManager	АдминистраторКурсора
Directory	Каталог
Dos	Dos
File	Файл
Font	Шрифт
GraphicsObject...	ГрафическийОбъект...
GraphicsMedium...	ГрафическаяСреда...
GraphicsTools...	ГрафическиеИнструменты...
Magnitude	Величина
Association	АссоциативнаяПара
Character	Символ
Date	Дата
Number...	Число...
Time	Время
Menu...	Меню...
MenuItem	ПунктМеню
Point	Точка
Rectangle	Прямоугольник
Stream	Поток
ReadStream	ПотокЧтения
WriteStream...	ПотокЗаписи...
UndefinedObject	НеопределенныйОбъект
ViewManager...	АдминистраторВида...
Window...	Окно...

Рис. 1.1. Часть иерархии классов системы *Smalltalk Express*

В «чистом» однородном объектно-ориентированном языке программирования, каковым является Смолток, любой объект системы должен быть экземпляром некоторого класса, а единственным способом управления объектами — посылка объектам сообщений. Запись по синтаксическим правилам языка Смолток операции посылки объекту сообщения будем называть *выражением*. Каждое выражение отделяется от следующего за ним выражения точкой².

Процесс реакции объекта на полученное им сообщение, включая возврат некоторого объекта, будем называть *выполнением выражения*. Например, чтобы определить, экземпляром какого класса является объект, надо послать этому объекту сообщение `class`, требующее от объекта вернуть свой класс. Чтобы сделать это, необходимо сначала указать объект, которому посылается сообщение, а затем записать само сообщение. Таким образом, если мы желаем узнать класс, экземпляром которого является объект 3.2, мы должны создать выражение `3.2 class`. В процессе выполнения сообщения, объект 3.2 вернет свой класс, в данном случае `Float` (Вещественное)³. Класс `Float` — один из подклассов класса `Number`.

Но как реально послать объекту сообщение и получить на него ответ? Более того, как в системе создать объект — экземпляр класса? Чтобы ответить на эти два ключевых вопроса, надо набраться терпения и внимательно рассмотреть достаточно сложный вопрос функционирования самой системы. Можно было бы поговорить об этом и позже, но тогда во всем, что бы мы ни делали, оставались бы некоторые недоговоренности и тонкости, на которые пришлось бы не обращать внимания. Как нам кажется, лучше все принципиальные вещи рассмотреть сразу, хотя это и потребует больших усилий при первом знакомстве. Зато потом все будет понятнее и проще.

1.2.2. Классы и их метаклассы

Объект системы Смолток должен быть экземпляром некоторого класса. Чтобы создать такой объект согласно шаблону, определяемому классом, следует обратиться к классу с сообщением, требующим создания экземпляра. Следовательно, сами классы должны выступать как объекты систе-

² В *Smalltalk Express*, если выражение единственное или последнее, точку можно не ставить (граница выражения ясна и так), а в других реализациях языка Смолток (например, в IBM Smalltalk) выражение всегда должно завершаться точкой.

³ Когда в тексте нам впервые встретится класс, имя которого не отражено на рис. 1.1, мы будем рядом в круглых скобках приводить перевод на русский язык имени класса.

мы. Подобно тому, как объекты создаются из классов, так и сами классы должны рассматриваться в языке как объекты, создаваемые в соответствии с шаблоном, представленным некоторым классом. Такой класс для класса называется *метаклассом* данного класса. При этом класс — *единственный* экземпляр своего метакласса. В языке Смолток метакласс, в отличие от класса, не имеет имени, и доступ к нему осуществляется посылкой классу сообщения `class`. Например, пошлем сообщение `class` классу `Float`, то есть выполним выражение `Float class`. В ответ класс `Float` должен вернуть имя класса, экземпляром которого он является, то есть имя своего метакласса, но поскольку у метаклассов нет имен, будет просто возвращено `Float class`.

Таким образом, мы можем сделать очень важный для дальнейшего вывод о том, что в системе Смолток есть два вида объектов: те, которые могут создавать экземпляры, и те, которые не могут этого делать.

Метакласс — тоже класс системы и должен подчиняться всем правилам, имеющим место для классов, а значит, входить в иерархию классов. Но какое место он должен занимать в иерархии? Первое и совершенно очевидное правило состоит в том, что приведенная выше иерархия классов должна сохраняться и для их метаклассов. То есть если `Класс1` есть подкласс `Класс2`, то метакласс `Класс1` есть подкласс метакласса `Класс2`. Второе правило вытекает из общего принципа построения системы Смолток, согласно которому любой метакласс в системе, являясь ее объектом, должен быть экземпляром некоторого класса. Таким классом является класс `MetaClass`: любой метакласс системы есть экземпляр класса `MetaClass`, который, являясь шаблоном для метаклассов, описывает наиболее общие свойства тех объектов, которые умеют создавать единственный объект-класс.

Для завершения построения системы остается только связать между собой иерархию классов и метаклассов. Чтобы это сделать, вспомним, что базовым классом иерархии является класс `Object`, который сам не имеет суперкласса. Но класс `Object`, согласно предыдущему, имеет свой метакласс `Object class`, который, с одной стороны, есть экземпляр класса `MetaClass`, как и все другие метаклассы системы, а с другой стороны, является «базовым» классом иерархии метаклассов и, согласно принципу наследования при построении иерархии, определяет общую структуру и поведение всех экземпляров метаклассов, то есть классов. Для отражения этой роли метакласса `Object class` в системе Смолток создан класс с именем `Class`, который является суперклассом для `Object class`. Таким образом, все метаклассы являются подклассами класса с именем `Class`, одновременно являясь экземплярами класса `MetaClass`.

Такое строение иерархии классов имеет одно очень важное следствие: *цепочки суперклассов для самого класса и для его метакласса разные*. На-

пример, класс `Integer` имеет цепочку суперклассов (`Integer`, `Number`, `Magnitude`, `Object`) (мы всегда будем для полноты картины включать в такую цепочку сам класс). А класс `Integer class` (см. рис. 1.2)⁴ имеет цепочку суперклассов (`Integer class`, `Number class`, `Magnitude class`, `Object class`, `Class`, `Behavior`, `Object`).

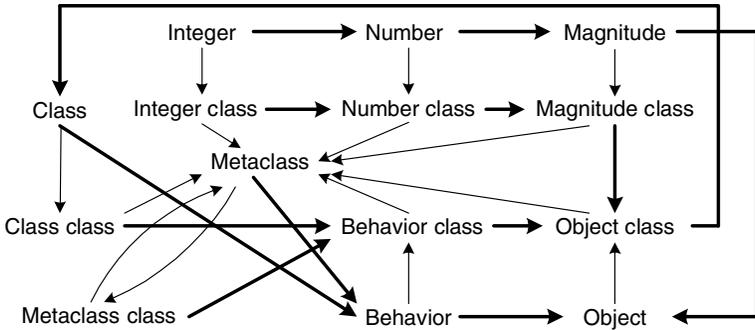


Рис. 1.2. Двойная иерархия класс/метакласс

Итак, два класса системы, а именно классы `Class` и `MetaClass`, описывают структуру и поведение тех объектов системы, которые могут создавать экземпляры. Все то общее, что присуще классам и метаклассам системы, описывается в классе `Behavior`, для которого классы `Class` и `MetaClass` являются подклассами. Кроме того, класс `Behavior` определяет информацию, необходимую для работы интерпретатора языка Смолток.

Чтобы полностью построить иерархию, осталось рассмотреть еще одно «тонкое место». Класс `MetaClass` занимает в системе особое место, поскольку порождает как свои экземпляры все метаклассы системы, в том числе и собственный метакласс `MetaClass class`, и, с другой стороны, сам является экземпляром метакласса `MetaClass class` (рис. 1.2). В этом месте система имеет «особенность», позволяющую избежать бесконечной последовательности метаклассов. Отметим, что наличие метаклассов в чистой объектно-ориентированной среде, основанной на классах, существенно, так как позволяет каждому классу иметь свое особое поведение и структуру (методы и переменные класса), а наличие метаклассов высших порядков (метаклассов метаклассов) практического значения не имеет.

⁴ Стрелки, выделенные на диаграмме полужирным, обозначают отношение «подкласс → суперкласс», а тонкие стрелки обозначают отношение «экземпляр → класс».

Мы полностью разобрали структуру иерархии классов системы Смолток. В силу того, что вторая половина иерархии автоматически определяется первой, мы будем иметь дело в основном с иерархией классов (не забывая о существовании второй половины). Метаклассы для классов система создает и устанавливает в иерархию автоматически. Как итог, сформулируем те основные правила, которым подчиняется иерархия классов системы Смолток:

- 1) каждый класс, кроме самого класса `Object`, не имеющего суперкласса, имеет один непосредственный суперкласс и, в конечном счете, каждый класс является подклассом класса `Object`;
- 2) каждый класс в системе является единственным экземпляром своего метакласса;
- 3) иерархия метаклассов подобна иерархии классов;
- 4) класс `Object class` и все метаклассы являются подклассами класса `Class`;
- 5) каждый метакласс является экземпляром класса `MetaClass`.

1.2.3. Определение класса и пример класса

Обратите внимание, что имена классов системы Смолток всегда начинались с прописной буквы. Это не случайно. Классы являются объектами, доступными для всех других объектов. Такие объекты называют *глобальными*. По правилам синтаксиса языка имена глобальных объектов системы Смолток должны начинаться с прописной буквы⁵. Имя переменной — это *идентификатор*, то есть последовательность букв латинского алфавита и цифр, начинающаяся с буквы.

Имена всех глобальных объектов содержатся в системном словаре с именем `Smalltalk`, и такие объекты могут вызываться по имени в любое время и в любом месте. Более подробно это будет рассматриваться в разделе 2.4 «Переменные в языке Смолток», а пока нам этого достаточно, чтобы продолжить изучение иерархии классов и самих классов.

Определение класса системы Смолток должно описывать место класса в иерархии, набор переменных самого класса, как объекта системы, набор переменных его экземпляра, другие структурные единицы, если они есть, сообщения, которые можно посылать самому классу (интерфейс класса или, как еще говорят, протокол класса), сообщения, которые можно посылать

⁵ Честно говоря, с точки зрения естественного языка это выглядит нелепо: имена собственные пишутся с маленькой буквы (иванов), а имена общие — с большой (Человек). К этому надо просто привыкнуть.

экземпляру класса (интерфейс экземпляра или протокол экземпляра). Но класс, как мы знаем, является единственным экземпляром своего метакласса, и именно метакласс ответствен за то, как ведет себя его экземпляр. Следовательно, определяя структуру и интерфейс класса, мы определяем метакласс данного класса, который создается автоматически и автоматически занимает подобающее ему место в иерархии. Вот почему мы говорили, что фактически будем иметь дело в основном с самими классами.

Все сказанное об определении класса представим следующей схематической записью:

```
ИмяСуперкласса <тип_подкласса>: #ИмяКласса
  instanceVariableNames: 'имя1 имя2 ...'
  classVariableNames: 'Имя1 Имя2 ...'
  poolDictionaries: 'ИмяПула1 ИмяПула2 ...' !
```

```
!ИмяКласса class methods !
```

```
  метод1
  ...!
  ...
  методN
  ...!!
```

```
!ИмяКласса methods !
```

```
  метод1
  ...!
  ...
  методS
  ...!!
```

О том, как все это записать в точном соответствии с синтаксическими правилами языка Смолток, чуть ниже. Для объяснения деталей определения класса приведем пример: часть описания класса *Association*. Комментарии — текст в двойных кавычках ("*...*") — не влияют на работу системы и переведены на русский язык. Весь остальной текст — синтаксически правильные последовательности выражений языка Смолток — так называемые *фрагменты* (chunks), разделенные восклицательными знаками '!'. Фрагмент, состоящий из посылки классу (или метаклассу) сообщения *methods*, открывает последовательность фрагментов — определений методов экземпляра (соответственно, класса). Эта последовательность завершается дополнительным восклицательным знаком. Такой формат (он подробно описан в [18]) использует большинство Смолток-систем при выводе программ в текстовой форме, в том числе и при ведении журналов.

```
Magnitude subclass: #Association 1
  instanceVariableNames: 'key value ' 2
  classVariableNames: '' 3
  poolDictionaries: '' ! 4
  5
!Association class methods ! 6
  7
key: anObject 8
  "Создает и возвращает экземпляр класса Association, 9
  с ключом, равным объекту anObject." 10
  ^self new key: anObject! 11
  12
key: aKey value: anObject 13
  "Создает и возвращает экземпляр класса Association 14
  с заданными ключом aKey и значением anObject." 15
  ^(self key: aKey) value: anObject! ! 16
  17
!Association methods ! 18
  19
key 20
  "Возвращает ключ получателя." 21
  ^key! 22
  23
key: anObject 24
  "Устанавливает ключ получателя равным anObject. 25
  Возвращает получатель как результат." 26
  key := anObject! 27
  28
printOn: aStream 29
  "Добавляет ASCII-представление получателя 30
  в поток aStream." 31
  key printOn: aStream. 32
  aStream nextPutAll: ' ==> '. 33
  value printOn: aStream! 34
  35
storeOn: aStream 36
  "Добавляет ASCII-представление получателя, 37
  из которого он может быть восстановлен, 38
  в поток aStream." 39
  aStream nextPutAll: 'Association key: ('. 40
  key storeOn: aStream. 41
  aStream nextPutAll: ') value: ('. 42
  value storeOn: aStream. 43
  aStream nextPut: $)! 44
```

value	45
"Возвращает значение получателя."	46
^value!	47
	48
value: anObject	49
"Устанавливает значение получателя равным anObject."	50
Возвращает получатель как результат."	51
value := anObject! !	52

Итак, в определении класса первая строка означает, что **Association** — имя нового подкласса **Magnitude**.

Структура каждого экземпляра класса **Association** описывается двумя переменными экземпляра с именами **key**, **value** в строке 2. Вне каждого конкретного объекта его переменные не видны. Переменные, доступные только одному объекту, называются локальными переменными. В данном случае мы имеем дело с локальными переменными, которые являются переменными экземпляра.

Переменные класса относятся к общим переменным (доступным многим объектам), поскольку доступны самому классу и всем его подклассам, а также всем экземплярам этого класса и его подклассов. Их имена пишутся с прописной буквы. В классе **Association** новые переменные класса не определяются, о чем свидетельствует строка 3 из определения класса.

Пул — словарь, хранящий общие переменные. Переменными из пула могут пользоваться те классы (и их экземпляры), в определении которых такой пул указан. Имя пула является именем глобальной переменной, поэтому начинается с прописной буквы. Из строки 4 мы видим, что у класса **Association** доступных ему пулов нет.

Обратимся к роли места класса в иерархии классов. Подкласс наследует от всех своих суперклассов переменные экземпляра, переменные класса и пулы. Значит, экземпляр подкласса содержит все переменные экземпляра, определенные в нем и во всех его суперклассах, имеет доступ ко всем переменным класса и пулам, определенным в нем и всех его суперклассах. Поскольку класс **Magnitude** не имеет ни переменных класса, ни переменных экземпляра, ни пулов, он ничего не добавляет ни в класс **Association**, ни в его экземпляры. Но класс **Magnitude** — подкласс класса **Object**, в котором есть три переменных класса: **RecursionInError**, **Dependents**, **RecursiveSet**. Следовательно, и сам класс **Association**, и все его экземпляры могут обращаться к этим переменным.

Класс **Association** является единственным экземпляром своего метакласса **Association class**, который имеет отличную от класса **Association** цепочку суперклассов, а именно, цепочку **Association class**, **Magnitude class**,

Object class, Class, Behavior, Object. Следовательно, класс **Association** наследует структуру и поведение еще и из классов **Behavior** и **Class**, и, если проверить, то увидим, что он наследует из них около десятка переменных. То есть класс **Association** имеет совсем другую структуру, чем его экземпляр. И не только структуру, но и поведение: класс **Association** может создавать экземпляры, а его экземпляр — нет. Это происходит именно потому, что они наследуют структуру и поведение по разным цепочкам суперклассов. Здесь сказывается та сложная структура иерархии классов, которую мы обсуждали выше.

Все, о чем мы так долго говорили, содержится в состоящем из одного выражения фрагменте в строках 1–4 листинга класса. Это выражение представляет собой посылку классу **Magnitude** сообщения с именем **subclass:instanceVariableNames:classVariableNames:poolDictionaries:** и с соответствующими объектами в качестве аргументов. Как ответ на это сообщение происходит определение в системе нового класса с именем **Association** и его метакласса. Другие сообщения, создающие классы, подробно описаны на стр. 44. Класс создан — двигаемся дальше.

Методы, расположенные в определении класса **Association** после разделителя **!Association class methods!**, называются методами класса и определяют поведение самого класса как объекта системы, то есть это те методы, которые в ответ на посланное ему сообщение может выполнить сам класс. Методы, расположенные в определении класса **Association** после разделителя **!Association methods!**, называются методами экземпляра и определяют поведение экземпляра класса, то есть это те методы, которые в ответ на посланное сообщение может выполнить экземпляр данного класса.

Но опять, в силу существования иерархии классов, это только часть методов класса и экземпляра. Ведь поведение точно так же наследуется, как и структура. Следовательно, в интерфейс экземпляра класса **Association** входят все методы экземпляра из классов **Magnitude** и **Object**. А в интерфейс самого класса **Association** входят все методы класса из классов **Magnitude** и **Object**, а также все методы экземпляра классов **Class**, **Behavior**, **Object**.

Остается выяснить, что из себя представляют методы и как они связываются с сообщениями, посылаемыми объектам.

1.2.4. Сообщения и методы

Метод (все равно, класса или экземпляра) состоит из шаблона сообщения и тела метода. Шаблон сообщения состоит из имени метода (или, как иногда говорят, селектора сообщения) и формальных параметров, если они нужны. Например, второй метод класса в классе **Association** имеет шаб-

лон сообщения вида `key: aKey value: anObject`, который состоит из имени сообщения (селектора) `key:value:` и двух формальных параметров `aKey` и `anObject`. Все, что после него, — тело метода, составляющее программу, выполняемую объектом-получателем в ответ на сообщение с селектором `key:value:`. В любом месте программы в нее могут добавляться комментарии — строки, заключенные в двойные кавычки. Традиционно тело метода начинают с комментария, поясняющего назначение и особенности применения метода.

Поскольку мы рассматриваем метод класса, то послать сообщение `key: aKey value: anObject` можно только классу `Association`, в ответ на которое класс `Association`, как написано в комментарии к методу (не задавайте пока себе вопрос *как* все произойдет, важно *что* сейчас произойдет), создаст экземпляр класса `Association` (ассоциативную пару) с заданными в сообщении ключом `aKey` и значением `anObject`. Например, рассмотрим выражение

```
myIndex := Association key: 'Index' value: 344017
```

В результате, как реакция класса на принятое им сообщение, будет создан экземпляр данного класса с заданными значениями переменных: строкой `'Index'` в качестве ключа и целым числом `344017` в качестве значения, а оператор присваивания `':='` заставит переменную с именем `myIndex` указывать на созданный объект.

Методы экземпляра выполняются в ответ на сообщения, посылаемые экземплярам класса, например, `myIndex value`. Здесь сообщение `value` посылается экземпляру класса `Association`, доступному через переменную `myIndex`. Как результат, будет возвращено число `344017`, которое при создании экземпляра было задано в качестве значения.

Но почему так получилось? Найдем в определении класса `Association` метод с шаблоном сообщения `value` (строки 45–47). Тело этого метода состоит всего из одного выражения, `^value`, которое и выполняется. В этом выражении нам встретился оператор возврата значения `^`. Когда он встречается в теле метода, результат выполнения стоящего после него выражения возвращается как результат выполнения всего метода, и дальнейшее выполнение метода прекращается. В данном случае все выражение состоит из имени переменной экземпляра, поэтому будет возвращен объект, на который указывает переменная экземпляра `value`.

Взглянув на класс `Association`, мы обнаруживаем, что оператор возврата значения присутствует в теле не каждого метода. Что же будет возвращаться как результат выполнения метода в таком случае? Общее правило языка Смолток гласит: если нет указаний, что возвращать как результат выполнения метода, по умолчанию возвращается *получатель* сообщения.

То, что мы рассмотрели, не отвечает на главный вопрос: как объект, получивший сообщение, находит метод, который надо выполнить? Остановимся подробно на механизмах поиска по сообщению необходимого метода и его выполнения. Итак, как уже отмечалось, выполнение любого действия в системе Смолток осуществляется с помощью посылки объекту сообщения. Получив сообщение, получатель ищет метод с соответствующим сообщению шаблоном, начиная поиск обычно со своего класса. Если объект — класс, то метод ищется среди методов класса, а если объект — экземпляр класса, то среди методов экземпляра класса. Если метод с соответствующим шаблоном находится в классе получателя, то он выполняется, и как результат его выполнения обязательно возвращается некоторый объект, который информирует того, кто послал сообщение, что выполнение метода завершилось с содержащимся в возвращаемом объекте результатом.

А если метода с нужным шаблоном нет в классе? Тогда к работе подключается иерархия классов, а точнее, цепочка суперклассов для класса объекта-получателя. Если в классе подходящего метода нет, метод ищется в ближайшем его суперклассе. Если нужного метода нет в суперклассе, то поиск продолжается в следующем по иерархии суперклассе и так далее, пока не доберемся до класса `Object`. А если нужного метода нет и там? Тогда виртуальная машина посылает объекту-получателю сообщение `doesNotUnderstand:`. Соответствующий ему метод, определенный в классе `Object`, возбуждает ошибку времени выполнения (см. разд. 3.3).

Таким образом, посылка сообщения включает в себя:

- определение объекта, которому посылается сообщение;
- определение, если они нужны, объектов-аргументов сообщения;
- определение метода, который ищется в классе или суперклассах;
- выполнение метода и возвращение некоторого объекта.

Что же произойдет в ответ на `Association key: 'Index' value: 344017`? Начнется поиск метода с именем `key:value`: среди методов класса в классе `Association`. Там такой метод есть, и он выполнится.

Давайте посмотрим на то, как он будет выполняться. Тело этого метода очень простое и состоит всего из одного выражения `^(self key: aKey) value: anObject`. При сравнении с шаблоном сообщения параметр `aKey` станет равным строке `'Index'`, а параметр `anObject` — целому числу `344017`. После этого начнет выполняться выражение `^(self key: 'Index') value: 344017`. В начале выражения стоит символ возврата значения, следовательно, тот объект, который получится в результате вычислений, и будет возвращен методом как результат его выполнения. Первая часть выражения заключена в круглые скобки `(self key: 'Index')`, поэтому вычисляется первой. Первое слово в скобках — имя псевдопеременной `self`. Более подробно о ней мы

поговорим в разд. 2.5, а пока нам достаточно знать, что оно обозначает получателя сообщения, то есть класс `Association`.

Таким образом, первым выполняется выражение `Association key: 'Index'`. Класс `Association` ищет и находит метод с именем `key`: среди своих методов. Найденный метод выполняется с аргументом `'Index'`. Его тело состоит из выражения `^self new key: aKey`, следовательно, начнет выполняться выражение `^Association new key: 'Index'`. В нем классу `Association` сначала посылается сообщение с именем `new`. Соответствующего метода среди методов класса `Association` нет. Поскольку нужен метод класса, то класс `Association` начинает поиск нужного метода, используя цепочку суперклассов (`Association class`, `Magnitude class`, `Object class`, `Class`, `Behavior`, `Object`). Нужный метод найдется в классе `Behavior` и будет выполнен классом `Association`, возвращая его новый экземпляр.

Созданный экземпляр становится объектом, которому посылается сообщение `key: 'Index'`. Поиск метода начнется среди методов экземпляра в классе `Association`. Там нужный метод есть, и состоит он из одного выражения `key := anObject`, которое присваивает значение переменной `key` созданного экземпляра класса `Association`. Поскольку оператора возврата нет, метод возвращает получателя сообщения, то есть экземпляр класса `Association`, у которого инициализирована первая переменная.

Для завершения выполнения первоначального метода надо еще послать сообщение `value: 344017` экземпляру класса `Association`, полученному в результате предыдущих вычислений. Соответствующий метод находится в классе `Association`. Окончательно экземпляр класса `Association` со всеми инициализированными переменными возвращается как результат выполнения выражения `Association key: 'Index' value: 344017`. Именно он и будет значением переменной `myIndex`.

Мы рассмотрели то, как общие принципы ООМП реализуются в языке Смолток. Теперь оставим на время общие подходы и механизмы ООМ. Перейдем к описанию синтаксиса языка.

ГЛАВА 2

СИНТАКСИС ЯЗЫКА СМОЛТОК

2.1. Определение объекта

2.1.1. Литеральное определение объектов

Объекты языка, которые можно определить, просто записывая их, называются *литеральными объектами* (*литералами*). Следовательно, особенность таких объектов состоит в том, что для создания их достаточно «написать», то есть представить в литеральной форме¹.

Символы

Чтобы писать, нужен алфавит. Алфавит системы Смолток не отличается от алфавита других языков программирования и состоит из букв, цифр, знаков пунктуации и так далее. Но в языке Смолток это, как и все в системе, объекты — экземпляры класса `Character` или символы.

Экземпляр класса `Character` не надо создавать, он всегда присутствует в системе после ее запуска, а чтобы им воспользоваться, его надо только записать тем или иным образом. Печатаемый объект-символ можно задать в виде символа `$` и последующего литерала. Например, `$A` — экземпляр класса `Character`, представляющий латинскую заглавную букву `A`, `$,` — экземпляр класса `Character`, представляющий запятую, `$ж` — экземпляр класса `Character`, представляющий строчную русскую букву `жэ`, `$7` — экземпляр класса `Character`, представляющий цифру `7`. Те символы, которые нельзя ввести с клавиатуры, можно получить с помощью методов класса `Character` (см. разд. 6.2). Экземпляры класса `Character` еще называются символьными константами, поскольку структуру данного объекта изменить нельзя.

Строки

Строка — последовательность символов, заключенная в одинарные кавычки. Например, `'Bold'`, `'seed'`, `'Hello word'`. Любая строка — это экземпляр класса `String` (Строка). Как и для символов, чтобы создать новый экземпляр класса `String`, достаточно написать нужную строку.

¹ С точки зрения организации системы это — посылка сообщений соответствующему классу, в результате чего и порождаются новые экземпляры. Но все это остается прозрачным для пользователя.

Числа

В системе Смолток могут использоваться целые числа (экземпляры класса `Integer`), рациональные дроби (отношения двух целых — экземпляры класса `Fraction`), рациональные числа в форме числа с плавающей точкой (экземпляры класса `Float`). Все числа — литеральные объекты. Числа в Смолтоке могут записываться в любой системе счисления. При выполнении арифметических операций все числа переводятся в десятичную систему.

Приведем несколько примеров чисел, одновременно объясняя особенности их представления:

`16r2FA1`, `16r-FF`, `-23`, `36r2AZ7` — примеры различных целых чисел. Число перед 'r' (всегда в десятичной записи) указывает на основание системы счисления. Если 'r' перед числом отсутствует, число записано в десятичной системе счисления. Для обозначения недостающих цифр используются буквы латинского алфавита, поэтому возможное наибольшее основание счисления — 36. Первые два числа записаны в 16-ричной системе счисления, третье — в десятичной, а последнее — в 36-ричной системе счисления.

`34/55`, `8r-5366/8r571`, `2r10011101/2r1011` — рациональные числа (дроби) в десятичной, восьмеричной и двоичной системе счисления соответственно. Числитель и знаменатель дроби могут представляться в разных системах счисления.

`3.141592653`, `2.54e-2` — десятичные числа с плавающей точкой, в обычной и научной форме записи.

`16rFe-2` (15×16^{-2}), `8r-2e-3` (-2×8^{-3}) — экземпляры класса `Float` в научной форме записи в системе счисления, отличной от десятичной (в скобках приведены их десятичные представления). Как видно из примеров, основание порядкового множителя в научной форме записи вещественного числа всегда совпадает по значению с основанием системы счисления. Показатель степени порядкового множителя всегда задается в десятичной системе.

Экземпляры класса `Number` еще называются числовыми константами, поскольку структуру созданного числа-объекта изменить нельзя.

Системные имена

Системное имя — уникальная последовательность символов одного из следующих видов:

- последовательность букв и цифр, обязательно начинающаяся с буквы (идентификатор);

- последовательность из одного или более ключевых слов (идентификаторов, в конце которых стоит двоеточие);
- последовательность из одного или двух специальных символов

+ - / \ * = < = > ~ @ | % & , ? !

Каждое системное имя — экземпляр класса **Symbol** (**СистемноеИмя**). Когда в программе записывается системное имя, перед ним ставится символ **#**. Например, **#Red**, **#Green**, **#at:put:**, **#size**, **#**,. Используемая выше фраза «уникальная последовательность» означает, что в системе не могут существовать два системных имени (то есть два экземпляра класса **Symbol**), которые имеют одну и ту же последовательность символов. Смолтоковская система внимательно следит за уникальностью системных имен. Кроме того, система запрещает манипуляции с внутренним содержанием системного имени. Системные имена используются для именованя уникальных объектов системы. Например, имена классов представляются системными именами. Имена методов — тоже экземпляры класса **Symbol** (см. 2.2.1). И об этом тоже заботится сама система.

Экземпляр класса **Symbol** часто вводится в систему и выводится на печать без префикса **#**. Это возможно, когда мы используем экземпляр класса **Symbol** как ранее определенное имя видимого в текущем контексте объекта. Текущий контекст определяется тем, с каким инструментом системы мы работаем и каким образом его используем. Однако при работе с системным именем как таковым или при определении нового имени обязательно надо явно указывать префикс **#**, как это было сделано в 1-й строке определения класса **Association**.

Массивы

Еще один объект, который можно задавать литерально, — массив. Массив — экземпляр класса **Array** (подкласса класса **FixedSizeCollection** (**НаборФиксированногоРазмера**)). Он представляет собой объект, содержащий упорядоченный набор других объектов, каждому из которых, в соответствии с занимаемым им местом, приписывается индекс. Например, запись **#\$A \$B \$C** порождает массив из трех объектов, на первом месте стоит символ **\$A**, на втором — символ **\$B**, на третьем — символ **\$C**.

Запись **#{1 'two' \$D Green}** порождает экземпляр класса **Array** с четырьмя литеральными объектами. Последний объект в массиве — системное имя, но поскольку символ **#** уже использован перед определением самого массива, перед системным именем его можно опустить. Обратите внимание и на то, что элементы массива совсем не обязаны быть экземплярами одного класса. Они могут быть любыми объектами, в том числе и массивами.

Экземпляры классов **String**, **Symbol**, **Array** можно создавать не только литерально. Поведение литерально заданных объектов, представляющих массивы и строки, различается в разных реализациях. В одних (*VisualAge for Smalltalk*) их структуру (как и структуру символов, чисел, системных имен) после создания изменить нельзя. В других (*Smalltalk Express*, *VisualWorks*) эти объекты ведут себя точно так же, как и любой другой экземпляр того же класса, и потому после создания можно изменять их структуру. Когда мы будем рассматривать вопросы эквивалентности и равенства объектов, мы к этой проблеме еще вернемся.

2.1.2. Определение объектов посылкой сообщения

Литеральных объектов очень мало, а классов в системе очень много, так что первый способ определения объектов не универсален и доступен только для наиболее простых и часто используемых объектов. Все остальные объекты создаются посредством посылки сообщения классу. Метод, соответствующий такому сообщению, определяется или самим классом, как, например, метод с именем `key:value:` в рассмотренном нами ранее классе **Association**, или наследуется классом у одного из его суперклассов.

Сначала приведем примеры специальных сообщений, создающих экземпляры. Экземпляр класса **Date** представляет собой конкретный день. Чтобы создать экземпляр класса **Date**, который представляет сегодняшнюю дату, следует послать классу **Date** сообщение `today`. Будет возвращена дата (по часам компьютера), когда было послано это сообщение. Аналогично, экземпляр класса **Time** представляет некоторое конкретное время в течение дня. Чтобы создать экземпляр класса **Time**, представляющий текущее время, требуется послать классу **Time** сообщение `now`. Будет возвращено время (по часам компьютера) посылки сообщения.

Существуют и более универсальные сообщения создания экземпляров. Это сообщения `new` и `new:`. Первое сообщение просто создает экземпляр класса по содержащемуся в классе описанию, ничего в экземпляре конкретно не определяя. А второе используется аналогичным образом для тех классов, экземпляры которых имеют индексированные переменные (например, массив); в качестве аргумента выступает число индексированных переменных. Например:

Association new — порождает экземпляр класса **Association**, его переменные `key` и `value` не имеют разумных значений. Точнее, системой им автоматически присваивается значение `nil` — неопределенный объект.

Bag new — порождает экземпляр класса **Bag** без элементов.

Array new: 2 — порождает экземпляр класса **Array** с двумя элементами, каждый из которых **nil**.

String new — порождает экземпляр класса **String** без символов.

Изучая протокол сообщений класса для каждого класса иерархии, можно найти методы создания его экземпляра. Но всегда ли они есть? Формально — всегда, поскольку все метаклассы наследуют из класса **Behavior** методы **new** и **new:.** Но протоколы некоторых классов недостаточны для описания объекта с полностью определенными структурой и поведением. Впрочем, и в реальном мире мы имеем точно такую же картину: не существует легкового автомобиля вообще, он всегда конкретен: это машина марки «Форд», «Волга» и т.д. Но все легковые автомобили имеют некоторые общие структурные элементы и поведение, которые их отличают от всех других автомобилей. Точно так же и в системе Смолток не существует набора вообще, а есть массивы, множества, упорядоченные наборы. Синтаксически выражение **myCollection := Collection new** правильно, его можно выполнить, создавая экземпляр класса **Collection** с именем **myCollection**, но такой экземпляр бесполезен. С другой стороны, при выполнении выражения **myCollection := Set new** будет создан полностью работоспособный экземпляр, поскольку протокол класса **Set** полон.

Классы, протоколы которых неполны (что не позволяет им создавать полноценные экземпляры), создаются для того, чтобы описывать те общие характеристики, которые присущи всем их подклассам. Такие классы называются *абстрактными классами*. Многие классы, имеющие большое число подклассов, являются абстрактными. К ним относятся классы **Object**, **Collection**, **Magnitude**, **Window** и другие. Классы, создающие полнофункциональные экземпляры, полностью реализуют протоколы своих абстрактных суперклассов. Примерами таких классов являются классы **Set** и **Float**.

2.2. Сообщения

Мы научились создавать объекты. Но объект сам по себе не может ничего делать, и чтобы заставить объект работать, ему надо послать сообщение. Сообщения не все одинаковы, они могут иметь разный вид и передавать разный объем информации получателю сообщения. Рассмотрим самую простую ситуацию. Например, пошлем целому числу **30** сообщение **factorial**².

² В математике эта функция определяется так: $0! = 1$, $n! = n * (n - 1)!$, когда $n > 0$. Она вычисляет произведение всех чисел от 1 до n включительно. Кстати, попробуйте вычислить это произведение в других языках программирования.

В результате будет найден соответствующий сообщению метод, он будет выполнен и будет возвращен результат — целое число.

Прежде чем продолжить, давайте договоримся записывать само выражение и результат его выполнения, пользуясь следующей краткой записью: <выражение> → <возвращаемый объект>.

Тогда предыдущий пример запишется так:

30 factorial → 265252859812191058636308480000000

Рассмотрим выражение `#('or' 'and' 'xor') size`. Экземпляру класса `Array` посылается сообщение `size`, которое требует от массива вернуть число содержащихся в нем элементов. Таким образом,

`#('or' 'and' 'xor') size` → 3

В приведенных примерах для выполнения операций достаточно только получателя сообщения. Но это не всегда так. Часто требуются дополнительные объекты — аргументы. Предположим, нам нужно выяснить, какой объект является первым элементом массива. Для этого массиву следует послать сообщение `at`: с индексом в качестве дополнительного объекта:

`#('or' 'and' 'xor') at: 1` → 'or'

В этом примере массив `#('or' 'and' 'xor')` — получатель сообщения, `at`: — имя сообщения (селектор сообщения), а `1` — аргумент сообщения; как результат возвращается строка 'or'.

Примерами сообщений с дополнительными объектами-аргументами являются сообщения, требующие выполнения арифметических операций:

`2 * 4` → 8

`2.422 / 2` → 1.211

`1.5 + 6.3e2` → 631.5

Как видите, посылка арифметических сообщений в языке Смолток выглядит точно так же, как и во многих других языках программирования. Но в Смолтоке даже сложение двух чисел рассматривается как результат посылки сообщения. Например, в выражении `3 + 4` целое число `3` — получатель сообщения, `+` — имя сообщения, целое число `4` — аргумент, целое число `7` — результат.

Поэтому выполнение арифметических выражений в Смолтоке отличается от общепринятых правил и производится всегда строго слева направо, без учета приоритетов арифметических операций, существующих в математике. Приоритеты у операций есть, но имеют совершенно другой смысл (см. ниже), согласно которому все арифметические операции имеют один и тот же приоритет. Вычисление выражения `3 + 4 * 2` происходит слева направо, сначала выражение `3 + 4` возвращает `7`, затем вычисляется выражение `7 * 2` и возвращается как результат `14`. Чтобы восстановить общепринятый математический порядок вычисления подобных выражений,

надо использовать круглые скобки: $3 + (4 * 2)$. Если выражение содержит круглые скобки, то подвыражение в круглых скобках выполняется первым. Это несколько неудобно, но опыт показывает, что преимущества предельно единообразного построения языка программирования перевешивают такие неудобства. Хотя числа представляются в системе экземплярами различных классов, но реализованы таким образом, что ведут себя так, как если бы они принадлежали наиболее общему из них. Общий протокол для всех числовых объектов наследуется из класса `Number`, подклассами которого и являются классы `Float`, `Fraction`, `Integer`.

$$\begin{array}{c} 3 + 4 * 2 \\ \underbrace{\hspace{1.5cm}} \\ \downarrow \\ 7 * 2 \\ \underbrace{\hspace{1.5cm}} \\ \downarrow \\ 14 \end{array}$$

Аргументов в сообщении может быть и более одного. Примером такого выражения является уже известное нам выражение `Association key: 'Index' value: 344017`. Все вроде бы понятно, но давайте посмотрим на выражение `#{5 10 15} at: 5 - #'or' 'and' 'xor' size`. Имеет ли такое выражение право на существование с точки зрения синтаксиса языка? Если имеет, то по каким правилам выполняется и каков результат? Для ответа на этот несложный вопрос нужны новые понятия.

2.2.1. Унарные, бинарные и ключевые сообщения

Сообщения, для выполнения которых требуется только получатель, называются *унарными*. Встретившись в одном выражении, унарные сообщения выполняются слева направо. Вот несколько выражений с унарными сообщениями (их смысл, как нам кажется, понятен):

```

#(1 2 3) reversed → (3 2 1)      (Набор в обратном порядке)
$A asciiValue    → 65
65 asCharacter   → $A
Set name size    → 3              (Длина имени класса Set)

```

Сообщение с именем, состоящим из одного или более ключевых слов (см. 2.1.1), называется *ключевым сообщением*. В выражении за каждым ключевым словом следует аргумент:

```

#(1 2 3 4 5) includes: 4 → true   (истина)
#(1 2 3 4 5) includes: 7 → false  (ложь)
#(9 8 7 6 5) copyFrom: 2 to: 4 → (8 7 6)
'1 love you' copyFrom: 8 to: 10 → 'you'

```

Обратите внимание: различные объекты могут отвечать на одно и то же сообщение, но по-разному. В данном случае массив и строка отвечают на сообщение с именем `copyFrom:to:`, но массив возвращает массив, а строка — строку. Это явление, как мы знаем, называется полиморфизмом.

Сообщения с одним аргументом и именем, состоящим из одного или двух специальных символов (см. 2.1.1), называются *бинарными сообщениями*. Бинарные сообщения всегда выполняются строго слева направо. Арифметические сообщения — частный случай бинарных сообщений. Существует множество других. Вот два простых примера:

$\#(1\ 2\ 3), \#(4\ 5\ 6) \rightarrow (1\ 2\ 3\ 4\ 5\ 6)$ (конкатенация)
 $51 < 100 \rightarrow \text{true}$ (сравнение)

2.2.2. Выполнение сложных выражений

В выражениях, в которых сообщения посылаются результатам выполнения других сообщений, а аргументы также получают как результаты послылки, надо использовать следующие правила:

- 1) унарные сообщения имеют приоритет над бинарными;
- 2) бинарные сообщения имеют приоритет над ключевыми;
- 3) все сообщения одинакового приоритета выполняются строго слева направо;
- 4) круглые скобки имеют наивысший приоритет, и прежде всего выполняются выражения, стоящие в скобках, после чего скобки заменяются на возвращаемые объекты.

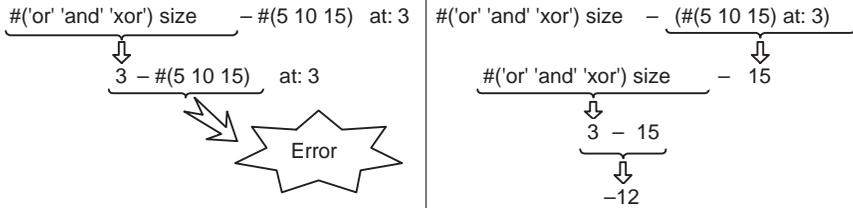
$\#(5\ 10\ 15)\ \text{at:}\ 5 - \underbrace{\#('or'\ 'and'\ 'xor')\ \text{size}}_{\downarrow}$
 $\#(5\ 10\ 15)\ \text{at:}\ 5 - 3$
 $\#(5\ 10\ 15)\ \text{at:}\ 2$
 \downarrow
 10

Учтем эти правила и вернемся к невыполненному выражению из предыдущего раздела. В силу первого правила первым выполняется выражение $\#('or'\ 'and'\ 'xor')\ \text{size}$, которое возвращает число 3. После этого возникает новое выражение $\#(5\ 10\ 15)\ \text{at:}\ 5 - 3$. По второму правилу следующим выполняется выражение

$5 - 3$, которое возвращает число 2. Теперь исходное выражение принимает вид $\#(5\ 10\ 15)\ \text{at:}\ 2$; оно выполняется и возвращает второй элемент массива — 10.

Теперь выполним похожее выражение $\#('or'\ 'and'\ 'xor')\ \text{size} - \#(5\ 10\ 15)\ \text{at:}\ 3$. После выполнения выражения $\#('or'\ 'and'\ 'xor')\ \text{size}$ исходное выражение примет вид $3 - \#(5\ 10\ 15)\ \text{at:}\ 3$. По второму правилу следующим выполняется выражение $3 - \#(5\ 10\ 15)\ \text{at:}\ 3$. Что означает сообщение, требующее от числа 3 отнять массив? Объект 3 такого сообщения выполнить не может, он его не понимает и сообщает об этом отправителю сообщения (как он это делает, мы увидим позже). Таким образом, попытка выполнить такое сообщение приводит к сообщению об ошибке. Если же

его «подправить» с помощью круглых скобок, то в результате выполнения будет возвращено число -12 .



А как следует поступить, чтобы выполнить несколько выражений? В этом случае проблем не возникает, если не забывать каждое выражение завершать точкой. До сих пор рассматривалось только одно выражение, и точек мы не ставили. Приведем подобный пример, объясняя в комментариях справа, что происходит:

<p>Window turtleWindow: 'Turtle Graphics'. Turtle home. Turtle go: 100. Turtle turn: 120. Turtle go: 100. Turtle turn: 120. Turtle go: 100. Turtle turn: 120.</p>	<p>"Создать окно." "Поставить перо в центр." "Пройти 100 пикселей." "Повернуть на 120°." "Пройти 100 пикселей" "Повернуть на 120°." "Пройти 100 пикселей." "Повернуть на 120°."</p>
--	--

В результате будет нарисован правильный треугольник со стороной в 100 пикселей. Как видите, каждое сообщение в последовательности сообщений отделяется от остальных точкой. Мы поместили каждое сообщение в отдельную строку только для удобства чтения. Для ясности еще стоит сказать, что Turtle — имя пера графического окна.

В этом примере одному объекту по очереди посылаются разные сообщения, и мы каждый раз указываем этот объект. Есть способ сделать то же самое, но более изящно. Следует воспользоваться каскадным сообщением — стенографическим способом записи нескольких сообщений, посылаемых одному получателю. В таком сообщении объект-получатель указывается один раз, а посылаемые ему сообщения разделяются точкой с запятой. Например, предыдущий пример запишется так:

Window turtleWindow: 'Turtle Graphics'.
 Turtle home; go: 100; turn: 120; go: 100; turn: 120; go: 100; turn: 120.

Приведем пример каскадного сообщения, используемого при определении нового множества. Создадим экземпляр класса Set и добавим в него три элемента: Set new add: 1; add: 2; add: 3.

2.3. Блоки

Работа не волк, в лес не убежит.

Пословица

Прежде чем двигаться дальше, рассмотрим подробнее очень важный объект, называемый блоком и часто используемый в управляющих структурах. Блок — экземпляр класса `Context`³, который представляет отложенную последовательность вычислений, описываемых выражениями языка внутри блока. Блок, как и любой другой объект, может присваиваться переменным, передаваться как параметр при посылке сообщения или возвращаться в качестве результата выражения. Все выражения, составляющие блок, заключаются в квадратные скобки и разделяются точками. Блок может иметь параметры, которые стоят в начале блока и отделяются от выражений блока символом `|`. Имени каждого параметра предшествует двоеточие `:`.

Блоки можно вкладывать один в другой. Блок выполняется только при посылке ему соответствующего сообщения, вид которого зависит от числа переменных блока. При выполнении блока, если не предписано что-то другое, возвращаемый блоком результат является значением последнего выполненного в блоке выражения. Обратите внимание: это совсем другое правило, отличное от правила, применяемого при выполнении метода. Блок может содержать выражение, перед которым стоит оператор возврата значения `^`. Тогда значение, полученное при выполнении этого выражения, возвращается как результат выполнения блока и дальнейшее выполнение блока прекращается. Если блок с оператором возврата значения `^` стоит в теле метода, то вместе с завершением выполнения блока прекращается выполнение метода, а в качестве возвращаемого методом объекта используется значение, возвращенное блоком. Вот примеры блоков:

```
[2 + 3]
[^ true]
[:letter | letter isVowel]
[:a :b | a < b]
```

Лучше понять, что такое блок, поможет такая аналогия. Блок с параметрами — это своеобразная функция от указанных параметров, тело которой расположено после символа `|`. И как для всякой функции, чтобы ее вычислить при конкретных значениях переменных, ей следует передать нужное число аргументов и «приказать» произвести вычисления.

³ В других реализациях — `BlockContext` или `BlockClosure`. Чтобы узнать имя этого класса, выполните выражение: `[] class`.

Блок без параметров выполняется, когда ему посылается унарное сообщение `value`. Например, при выполнении выражения `[2 + 3]` `value` возвращается число 5. Блок с одной переменной вычисляется тогда, когда ему посылается ключевое сообщение `value: anObject`, аргумент которого подставляется в блок на место его переменной. Например,

```
[array | array at: 1] value: #($A $B $C $D) → $A
```

Блок с двумя переменными выполняется при посылке ему ключевого сообщения `value: anObject1 value: anObject2`, аргументы которого по порядку подставляются в блок на место его переменных. Например,

```
[a :b | a < b] value: 3 value: 5 → true
```

В реализации *Smalltalk Express* блок не может иметь больше двух параметров, а в других реализациях — может, и там есть соответствующие сообщения для вычисления таких блоков.

2.4. Переменные в языке Смолток

Как мы знаем, структура любого объекта описывается переменными. Большинство таких переменных имеют имена. Каждая переменная запоминает один объект, и имя переменной — просто ссылка на этот объект. Поэтому переменные в системе Смолток не имеют типа. Переменные только указывают на объект, и присвоение переменной некоторого значения создает новый указатель на объект (новый псевдоним объекта), а не создает копию объекта. Для копирования объектов в системе существуют специальные методы (`copy`, `deepCopy`, `shallowCopy`). Объекты, к которым можно получить доступ из конкретного места программы, определяются через доступные в этом месте имена переменных.

Переменные отличаются друг от друга временем своего существования и областью действия (областью видимости). В системе *Smalltalk Express* существуют следующие разновидности переменных⁴: глобальные переменные; переменные пула; переменные класса; экземплярные переменные класса; переменные экземпляра; временные переменные методов; переменные блоков; псевдопеременные.

Глобальные переменные системы доступны любому объекту системы и находятся все они, как уже отмечалось, в словаре системы с именем `Smalltalk`. Переменные, содержащиеся в словарях, называемых пулами, доступны всем экземплярам классов, в которых пул объявлен при определении класса

⁴ В других реализациях языка Смолток могут определяться и другие разновидности переменных, например, локальные переменные блока.

(в аргументе ключевого слова `poolDictionaries`), а также всем экземплярам их подклассов. Все эти переменные «живут» в системе до тех пор, пока их явно не удалят.

Все переменные класса доступны этому классу, его подклассам, всем экземплярам класса и его подклассов. Сам класс и все его подклассы разделяют одно значение такой переменной. Эти переменные существуют в системе до тех пор, пока существует определяющий их класс. Имена всех этих переменных, так же как и имена пулов, являются общими переменными и начинаются с прописной буквы.

Экземплярные переменные класса⁵ (или переменные класса как экземпляра) доступны из класса и всех его экземпляров, но, в отличие от обычных переменных класса, не из его подклассов. Подклассы тоже имеют переменные с такими именами, но это другие переменные, значения их разделяются каждым классом только со своими экземплярами. Чтобы определить экземплярные переменные класса, метаклассу класса посылается сообщение `#instanceVariableNames`: с параметром — списком имен переменных. Некоторые реализации Смолтока имеют дополнительные сообщения создания подклассов, позволяющие задавать экземплярные переменные класса сразу при его создании.

В отличие от общих переменных, остальные переменные хранятся внутри каждого объекта, доступны только ему и исчезают вместе с ним. Они относятся к короткоживущим переменным, которые называются локальными. Их имена пишутся с маленькой буквы. Ссылки на эти переменные возможны только внутри того объекта, которому они принадлежат.

Сначала рассмотрим переменные экземпляра. Переменные экземпляра бывают именованные (имеющие имя), как переменные `key` и `value` в классе `Association`. Еще бывают индексированные переменные экземпляра. Вспомните, ведь к элементам массива мы обращались не по имени, а по индексу. Какие будут у экземпляра переменные, зависит от того, как определяется сам класс.

Тип допустимых переменных экземпляра зависит от первого ключевого слова сообщения, определяющего класс. В *Smalltalk Express* тип класса может вводиться одним из трех ключевых слов:

`subclass`: — класс может содержать только именованные переменные;

`variableSubclass`: — класс может содержать именованные и индексированные переменные;

⁵ Эти переменные были впервые введены в IBM Smalltalk и, после включения в проект стандарта, добавлены во все современные реализации Смолтока.

variableByteSubclass: — класс может содержать только индексированные переменные, каждая из которых представляется байтом.

В последнем случае определяется подкласс, экземпляры которого имеют область памяти, организованную как массив байтов, что позволяет эффективно хранить «сырые» данные. Такие объекты определяют элементарные значения данных, и таковыми, например, являются экземпляры классов **String** и **Symbol**. Доступ к байтам этих классов осуществляют те же сообщения, что и для объектов с индексированными переменными, но они работают быстрее, поскольку всегда возвращают или присваивают целые значения, способные разместиться в одном байте. Примером класса, который определяется с помощью ключевого слова **variableSubclass:**, является класс **Array**. Большинство классов системы вводится с помощью ключевого слова **subclass:**.

К локальным переменным относятся временные переменные, определяемые в методах, и переменные блоков. Они создаются в момент вызова метода или блока и уничтожаются по окончании выполнения метода или блока. Имена этих переменных должны начинаться со строчной буквы.

Чтобы использовать временные переменные в методах системы, их сначала надо описать. Описание локальных переменных метода находится в самом начале тела метода и состоит из имен переменных между вертикальными линиями-ограничителями `|имя1 имя2 ...|`. В качестве примера рассмотрим метод класса из системного класса **Time**:

millisecondsToRun: aBlock

```
"Возвращает число миллисекунд, необходимых
системе для выполнения блока aBlock."
| startTime |
startTime := self millisecondClockValue.
aBlock value.
^ self millisecondClockValue – startTime
```

Первая строка — **millisecondsToRun: aBlock** — это шаблон (образец) сообщения, который представлен ключевым сообщением (состоящим из одного ключевого слова **millisecondsToRun:** и его аргумента **aBlock**). Сразу после шаблона сообщения в кавычках находится комментарий к определяемому методу. Затем строкой `| startTime |` вводится временная переменная метода с именем **startTime**, с которой могут работать все выражения в теле метода. Далее в теле метода находятся выражения, которые определяют производимые методом операции.

Выражение `startTime := self millisecondClockValue` производит присваивание значения временной переменной **startTime**. В этом выражении псевдопеременная **self**, как мы уже знаем, ссылается на получателя сообщения.

В данном случае, поскольку рассматривается метод класса, псевдопеременная `self` ссылается на класс `Time`. Сообщение `millisecondClockValue` вызывает метод класса из класса `Time`, который вычисляет время в миллисекундах, прошедшее от начала суток до момента вызова метода; на это число и будет ссылаться переменная `startTime`.

Выражение `aBlock value` выполняет блок `aBlock`. Здесь `aBlock` — имя параметра метода, который используется для ссылки на аргумент сообщения в теле метода, и по смыслу должен являться блоком.

Выражение `^self millisecondClockValue - startTime` сначала вычисляет текущее время, из которого затем вычитается время `startTime`, и разность возвращается как результат выполнения метода. После этого временная переменная `startTime` прекратит свое существование.

Временные переменные могут использоваться не только в методах, но и при выполнении некоторой последовательности выражений. Но прежде чем использоваться, они должны описываться. Попробуем в любой смолтоковской системе выполнить уже встречавшееся ранее выражение

```
myIndex := Association key: 'Index' value: 344017.
```

Система сообщит, что переменная `myIndex` ей неизвестна. Синтаксически правильной является запись с объявлением временной переменной

```
| myIndex |
myIndex := Association key: 'Index' value: 344017
```

После выполнения такого выражения переменная `myIndex` перестанет существовать и к ней будет невозможно обратиться. Подобного рода выражения можно рассматривать как методы без имени.

С переменными (параметрами) блока мы уже встречались и видели, как они описываются. Рассмотрим простой пример, который высвечивает одну проблему, характерную для *Smalltalk Express*, и связанную с переменными блока. В этой системе областью видимости переменных блока является метод, в котором этот блок вычисляется. В примере мы используем несколько еще незнакомых нам объектов и сообщений. Позже мы их рассмотрим подробно. Здесь же отметим, что в системном окне `Transcript` можно, посылая ему сообщение `show:`, отображать в виде строки любой объект, а посылая сообщение `cr`, перейти на новую строку.

```
| a b |
a := #(4 3 2 1).
Transcript show: a printString; cr.
b := SortedCollection sortBlock: [:a :b | a <= b].
b addAll: a.
Transcript show: a printString; cr.
```

При первом отображении в окне Transcript массива **a** все в порядке. Когда элементы из массива **a** добавляются в отсортированный набор **b**, используется правило сортировки, задаваемое блоком `sortBlock`, которое позволяет определить, куда поместить добавляемые элементы. Но это изменит **a** и **b**. Сообщение `addAll`: выполнится, но объект **a**, который отобразится в окне Transcript, будет целым числом, а не массивом (как и **b**).

Если выражение `b := SortedCollection sortBlock: [:a :b | a <= b]` заменить выражением `b := SortedCollection sortBlock: [:i :j | i <= j]`, то все будет работать корректно. Будьте внимательны и не используйте одни и те же имена для переменных методов и содержащихся в них блоков. Старайтесь избегать того же и для разных блоков, если не уверены, что блоки не будут иметь перекрывающихся жизненных циклов (ведь блоки — отложенные последовательности выражений). Итак, сделаем попытку выполнить выражения следующего вида

```
aCollection <имя метода> [:i | ... ].
```

```
...
```

```
bCollection <имя метода> [:i | ... ].
```

Возможно, все будет работать правильно. Однако, если первый блок будет сохранен в какой-либо переменной и вычислен во время выполнения второго блока, то могут возникнуть проблемы⁶.

Еще раз подчеркнем, поскольку с этим связано довольно много ошибок, что, когда блок выполняется в теле метода, в нем нельзя воспользоваться оператором возврата объекта (^) и, выйдя из блока, остаться в методе, содержащем данный блок. Если необходимо выйти из блока и остаться в методе, который его содержит, следует так структурировать метод и блок, чтобы выход из блока происходил в его конце без применения оператора возврата объекта. Помните, что в таком случае возвращаемое блоком значение — это значение последнего выполненного в блоке выражения. Если надо вернуть конкретное значение, поместите его во временную переменную, а блок завершите именем этой переменной как отдельным выражением. Например:

```
| returnVal |
```

```
...
```

```
[:colour | returnVal := #red. ... returnVal ].
```

```
...
```

⁶ В других реализациях языка Смолток (например, в VisualWorks) блоки являются *замыканиями*. Это означает, что их переменные действительно локальны для блока и вне блока не видны.

2.5. Псевдопеременные

В системе Смолток есть еще и псевдопеременные. Их имена начинаются со строчной буквы, но они, тем не менее, доступны всем объектам системы. Псевдопеременные — это имена-указатели специальных объектов системы, и, в отличие от переменных, их значения не могут изменяться. Псевдопеременными являются `nil`, `true`, `false`, `self`, `super`.

Псевдопеременная `nil` указывает на специальный объект — единственный экземпляр класса `UndefinedObject`, используемый, если необходимо указать на отсутствие какого-либо другого подходящего объекта. Например, при создании нового экземпляра класса, если не позаботиться о значениях переменных экземпляра, все они получают значение `nil`. Кроме того, с помощью объекта `nil` часто представляют бессмысленный результат.

Псевдопеременные `true` и `false` представляют логическую истину и логическую ложь и являются единственными экземплярами классов `True` (Истина) и `False` (Ложь) соответственно. Сами классы `True` и `False` — подклассы класса `Boolean`, в котором описан общий протокол поведения этих двух логических объектов. Определения всех трех классов не вводят никаких переменных и не определяют доступа к пулам. В классах `True` и `False` реализованы следующие сообщения, выполняющие логические операции:

Класс <code>Boolean</code>	Протокол экземпляра
----------------------------	---------------------

<code>& aBoolean</code>	Возвращает <code>true</code> , если и объект, получивший сообщение, и аргумент <code>aBoolean</code> истинны (операция вычисляющего “И”, в которой аргумент <code>aBoolean</code> всегда вычисляется, независимо от получателя сообщения).
-----------------------------	--

<code> aBoolean</code>	Возвращает <code>true</code> , если или получатель сообщения, или аргумент <code>aBoolean</code> истинны (операция вычисляющего “ИЛИ”).
-------------------------	---

<code>not</code>	Отрицание: если получатель сообщения <code>false</code> , возвращает <code>true</code> , если получатель сообщения <code>true</code> , возвращает <code>false</code> .
------------------	--

<code>eqv: aBoolean</code>	Возвращает <code>true</code> , если получатель сообщения эквивалентен (тождествен) аргументу <code>aBoolean</code> , иначе возвращает <code>false</code> .
----------------------------	--

<code>xor: aBoolean</code>	Исключающее “ИЛИ”: возвращает <code>true</code> , если получатель сообщения не эквивалентен <code>aBoolean</code> , иначе возвращает <code>false</code> .
----------------------------	---

<code>and: alternativeBlock</code>	Если получатель сообщения <code>true</code> , возвращает результат выполнения блока <code>alternativeBlock</code> , который не должен иметь аргументов; если получатель сообщения <code>false</code> , возвращает <code>false</code> без выполнения блока.
------------------------------------	--

<code>or: alternativeBlock</code>	Если получатель сообщения <code>false</code> , возвращает результат выполнения блока <code>alternativeBlock</code> , который не должен иметь аргументов; если получатель сообщения <code>true</code> , возвращает <code>true</code> без выполнения блока.
-----------------------------------	---

Обычно логические объекты возвращаются после посылки сообщений, которые требуют простого ответа типа «да/нет» (например, сообщений для сравнения величин: $<$, $<=$, $>$, $>=$).

Псевдопеременные `self` и `super` занимают в системе особое место. Они используются в теле методов и указывают на объект, который вызвал данный метод. Различаются они способом поиска того метода, который необходимо выполнить объекту: `self` начинает поиск метода в классе, которому принадлежит объект, получивший сообщение, а `super` — в суперклассе того класса, в котором находится метод, содержащий псевдопеременную `super`. Использование псевдопеременной `super` не в качестве получателя сообщения (а, например, в качестве аргумента) полностью совпадает с использованием псевдопеременной `self`.

Чтобы разобраться в тонкостях применения этих псевдопеременных, рассмотрим простой поясняющий пример, взятый нами из книги [5, гл. 4]. Сначала проследим, как выполняются сообщения к `self`, для чего определим два класса с именами `One` и `Two` без переменных и без методов класса; при этом сделаем класс `One` подклассом класса `Object`, а класс `Two` — подклассом класса `One`.

```
Object subclass: #One
  instanceVariableNames: ' '
  classVariableNames: ' '
  poolDictionaries: ' '!
!One class methods !
!One methods !
test
  ^1!
result1
  ^self test ! !

One subclass: #Two
  instanceVariableNames: ' '
  classVariableNames: ' '
  poolDictionaries: ' ' !
!Two class methods !
!Two methods !
test
  ^2! !
```

Пусть `example1` — экземпляр класса `One`, а `example2` — экземпляр класса `Two`; этого можно добиться, выполняя выражения:

```
example1 := One new.    example2 := Two new.
```

Ниже приведены результаты выполнения четырех выражений:

```
example1 test      → 1
example1 result1   → 1
example2 test      → 2
example2 result1   → 2
```

Сообщение `result1` в двух приведенных выражениях вызывает один и тот же метод из класса `One`. Результат этих выражений различен из-за сообщения к `self`, содержащегося в этом методе. Когда сообщение `result1` посылается `example1`, поиск соответствующего метода начинается с класса `One`, которому принадлежит получатель сообщения. Метод для `result1` находится в этом классе и состоит из одного выражения `^self test`. Псевдопеременная `self` ссылается на получателя сообщения, то есть на `example1`. Поиск метода `test` начинается в классе, которому принадлежит получатель сообщения `test`, то есть в классе `One`. Такой метод в классе есть, он выполняется и возвращает 1.

Когда же сообщение `result1` посылается объекту `example2`, поиск соответствующего метода начинается с класса `Two`. Так как нужного метода в классе нет, поиск продолжается в суперклассе класса `Two` — в классе `One`, где нужный метод есть и состоит из одного выражения `^self test`. И в данном случае псевдопеременная `self` ссылается на получателя сообщения, но это `example2`. Поэтому поиск метода `test` начинается в классе, которому принадлежит получатель сообщения `test`, то есть в классе `Two`. В классе `Two` есть такой метод, он выполняется и возвращает 2.

Механизм выполнения сообщения к псевдопеременной `super` можно объяснить, определяя еще два класса с именами `Three` и `Four`; при этом сделаем класс `Four` подклассом класса `Three`, а класс `Three` — подклассом класса `Two`. Пусть в классе `Four` переопределяется метод для сообщения `test`, а в классе `Three` определяются методы для двух новых сообщений — `result2` и `result3`.

```
Two subclass: #Three
  instanceVariableNames: ' '
  classVariableNames: ' '
  poolDictionaries: ' '
!Three class methods!
!Three methods!
result2
  ^self result1
result3
  ^super test!!
```

```

Three subclass: #Four
  instanceVariableNames: ' '
  classVariableNames: ' '
  poolDictionaries: ' '!
!Four class methods!
!Four methods!
test
^4!!

```

Итак, экземпляры классов `One`, `Two`, `Three`, `Four` отвечают на сообщения `test` и `result1`. Определим два новых объекта, выполняя выражения:

```
example3 := Three new.    example4 := Four new.
```

Попытка послать сообщение `result2` или `result3` объектам `example1` или `example2` вызовет сообщение об ошибке, так как экземпляры классов `One` и `Two` не понимают их. Ниже приводятся результаты посылки сообщений:

```

example3 test           → 2
example4 result1       → 4
example3 result2       → 2
example4 result2       → 4
example3 result3       → 2
example4 result3       → 2

```

Когда сообщение `test` посылается объекту `example3`, используется метод из класса `Two`, так как класс `Three` не переопределяет метод с именем `test`. Объект `example4` отвечает на сообщение `result1` возвращением числа 4 по той же причине, по которой объект `example2` возвращал число 2. При посылке сообщения `result2` объекту `example3` поиск метода начинается с класса `Three`. Обнаруженный там метод возвращает результат выражения `self result1`. Поиск метода для сообщения `result1` также начинается в классе `Three`, но нужного метода нет ни в классе `Three`, ни в классе `Two`. Однако он находится в классе `One` и возвращает результат выполнения выражения `self test`. Поиск метода для сообщения `test` еще раз начинается в классе `Three`. На этот раз нужный метод обнаруживается в классе `Two` — суперклассе класса `Three`, и возвращает число 2.

Эффект посылки сообщений к псевдопеременной `super` иллюстрируется ответами объектов `example3` и `example4` на сообщение `result3`. Сообщение `result3`, посланное объекту `example3`, вызывает поиск метода в классе `Three`. Найденный там метод возвращает результат выражения `super test`. Так как сообщение `test` посылается псевдопеременной `super`, поиск метода с именем `test` начинается в суперклассе класса `Three`, то есть в классе `Two`. Метод `test` из `Two` возвращает 2. Когда сообщение `result3` посылается

объекту `example4`, все равно возвращается 2, поскольку схема выполнения полностью идентична предыдущей, несмотря на переопределение классом `Four` метода для сообщения `test`.

Подчеркнем: использование псевдопеременной `super` не означает, что поиск нужного метода начнется в суперклассе получателя первоначального сообщения. Использование `super` означает, что поиск начинается в суперклассе того класса, в котором содержится метод, использующий псевдопеременную `super`. В рассмотренном примере классом, содержащим метод с псевдопеременной `super`, был класс `Three`, поэтому поиск нужного метода начинался в классе `Two`. Даже если бы класс `Three` переопределял метод `test`, результатом выполнения выражения `example4 result3` было бы 2.

Рассмотренные классы иллюстрируют еще одну отмеченную ранее особенность — полиморфизм. Обратите внимание, что в классах `One`, `Two`, `Four` содержится метод с одним и тем же именем `test`, но тела методов, исполняемые объектами разных классов в ответ на это сообщение, разные. Все определялось классом, которому принадлежал получатель сообщения. Как следует из предыдущего, основной смысл псевдопеременной `super` состоит в том, чтобы позволить объектам получить доступ к методам суперкласса, даже если эти методы данным классом были переопределены.

Особенно часто подобный доступ нужен при написании корректного метода создания нового экземпляра класса. Здесь очень важно правильно использовать псевдопеременные `super` и `self`. Если в таком методе вместо правильного выражения `^super new initialize` написать `^self new initialize`, то будет создан бесконечный цикл вызовов метода `new`.

2.6. Методы и примитивные методы

Вся работа осуществляется объектами, которым посылаются сообщения. В ответ объекты вызывают для исполнения методы, в которых в свою очередь посылаются сообщения другим объектам, и так далее. Но только пересылая сообщения от одного объекта к другому, толку не добьешься. Кто-то должен взаимодействовать с внешним миром. Но кто и как?

В методах многих классов встречается строка вида `<primitive: nn>`, где `nn` — некоторое число. Такое обозначение имеют *примитивные методы* системы, реализованные в виртуальной машине, а числа `nn` идентифицируют их в выражениях, написанных на Смолтоке. В теле метода такая конструкция располагается самой первой. Примитивные методы, как правило, выполняют операции низкого уровня (такие как арифметические операции, выделение памяти) или операции, критичные по производительности.

Например, в классе `Time` реализован следующий метод класса:

`clockTickPrimitive: anInteger`

"Частный — разрешить прерывание по таймеру."

```
<primitive: 18>
```

```
^self primitiveFailed.
```

Данный метод состоит из двух частей: примитивного метода с номером 18 и части, написанной на Смолтоке. Сначала выполняется примитивный метод. Его выполнение заканчивается либо успехом (возвращением некоторого объекта), либо неудачей. В первом случае выполнение метода прекращается и возвращенный примитивом объект возвращается как результат выполнения всего метода. Код на языке Смолток в этой ситуации выполняться на будет. Если примитивный метод не может быть выполнен, то выполняется часть метода, написанная на языке Смолток.

Такое разделение ответственности работает очень эффективно, поскольку примитивы управляют наиболее часто используемыми, но простыми случаями. Программу на языке Смолток изменять намного проще, чем виртуальную машину, поэтому часть, написанная на Смолтоке, отвечает за редкие, но более сложные ситуации.

До сих пор программирование на языке Смолток мы изучали чисто теоретически и уже знаем достаточно много. Пришло время познакомиться с одной из реализаций языка Смолток, а именно, со средой *Smalltalk Express* и ее основными инструментами.

ГЛАВА 3

СРЕДА *SMALLTALK EXPRESS*

Начнем работу со смолтоковской системой *Smalltalk Express*. Эта программа — обычное Windows-приложение. Свободно распространяемую версию системы *Smalltalk Express* можно получить по сети, обратившись на страницу «Смолток в России». Перед инсталляцией и первым запуском надо обязательно установить цветовую палитру в *256 цветов*. Это сделать просто: щелкнуть на свободной поверхности рабочего стола правой кнопкой мыши, открыть окно свойств экрана и на вкладке «Настройка» установить необходимую цветовую палитру. На том же сервере можно взять файл расширения системы до версии 2.0.4, что позволит после инсталляции в систему файла *instal.st* и последующего сохранения образа запускать систему в любых видеорежимах. Многие из того, о чем будет говорить далее, справедливо и для других смолтоковских систем (особенно для *Visual Smalltalk*), но будут отличия в деталях оформления и меню.

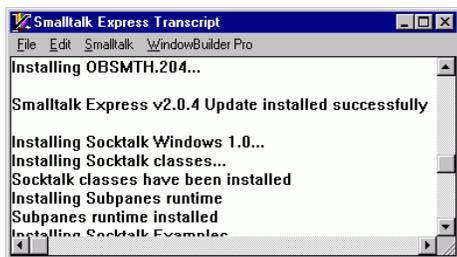


Рис. 3.1. Окно Transcript

Для запуска системы *Smalltalk Express* достаточно по порядку выбрать в Главном меню Windows (которое открывается кнопкой Пуск) раздел Программы, в нем папку *Smalltalk Express*, а в ней пункт *Smalltalk Express*¹. На экране появится главное окно среды разработки — *Smalltalk Express Transcript* (рис. 3.1).

Чтобы немного представить графические возможности системы, запустите приложение *GraphicsDemo*. Для этого выберите в окне *Transcript* пункты меню *File* \triangleright *Graphics Demo*.

Еще один момент: выход из системы *Smalltalk Express* происходит стандартным для Windows образом, только всегда будет появляться диалоговое окно, в котором вас спросят, надо ли сохранять образ системы. Чуть позже мы расскажем, что это такое, и вы будете самостоятельно принимать решение, а пока отвечайте на вопрос *No*. Будем считать, что первое поверхностное знакомство состоялось.

¹ Далее последовательность выбора пунктов меню будем записывать в виде: Главное меню \triangleright Программы \triangleright *Smalltalk Express* \triangleright *Smalltalk Express*.

3.1. Основные операции с окнами

Мы предполагаем, что читатели знают операционную систему Windows. Интерфейс системы *Smalltalk Express* фактически является компромиссом между классическим интерфейсом Smalltalk-80 и интерфейсом MS Windows. Поскольку многие особенности интерфейса, унаследованные *Smalltalk Express* от Smalltalk-80, «попали в стиль» пользовательского интерфейса Windows, читателям они хорошо знакомы.

3.1.1. Окна и панели

Под окном в среде *Smalltalk Express* понимается обычное окно Windows. Как объекты системы Смолтока, окна — экземпляры подклассов класса *Window*. Область внутри рамки окна является объединением одной или нескольких панелей — экземпляров подклассов класса *SubPane*. Окно Transcript (см. рис. 3.1), которое открывается при запуске системы, — пример окна с одной панелью.

Smalltalk Express использует интерфейс SDI (Single Document Interface), на котором основано большинство графических оконных систем. В этом режиме для каждого документа, функционального блока программы создается отдельное окно, независимое от других окон программы. Все окна Смолтока равноправны. Несколько особое место занимает уже знакомое окно Transcript — закрытие этого окна означает выход из системы.

Если на экране есть несколько открытых смолтоковских окон, то, нажимая функциональную клавишу [F9], можно переходить из одного окна в следующее. Если продолжать ее нажимать, то одно за другим выбираются все окна Смолтока. Этот механизм называется циклом по окнам. Он полезен тогда, когда открыто много окон, перекрывающих друг друга. Данный цикл затрагивает только окна среды *Smalltalk Express*.

Панели

Окно содержит одну или несколько панелей. Наиболее распространенные виды панелей — текстовые, списковые, анимационные, графические, панель группы. Но есть и множество других. Все они — экземпляры подклассов класса *SubPane*. Многие панели могут иметь связанные с ними контекстные (всплывающие) меню, которые вызываются при нажатии в активной панели правой кнопки мыши.

Панели, позволяющие редактировать текст, называются текстовыми панелями. Окно Transcript и рабочее окно Workspace (рис. 3.2), которое можно открыть, выбирая пункты меню File ▷ New Workspace или нажимая соче-

тание клавиш **[Alt]+[N]**, состоят из единственной текстовой панели. Все текстовые панели смолтоковской системы используют один и тот же интерфейс текстового редактора.

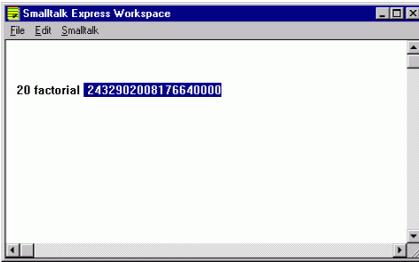


Рис. 3.2. Рабочее окно

Класс **ControlPane** (подкласс класса **SubPane**) содержит все управляющие элементы окна (*виджеты*), реализуемые такими классами, как **Button** (Кнопка), **EntryField** (ПолеВвода), **GraphBox** (ПолеГрафики), **ListVox** (ПолеСписка), **ScrollBar** (ПолосаПрокрутки) и **StaticPane** (НеизменяемаяПанель). Каждый из них имеет многочисленные подклассы, реализующие более специализированные управляющие элементы. Пользуясь окном просмотра иерархии классов (**Class Hierarchy Browser**), можно просмотреть иерархию классов системы.

Панель группы (экземпляр класса **GroupPane**) используется для того, чтобы объединить несколько разных панелей в единое целое, создавая внутри окна составную панель. В окне просмотра иерархии классов кнопки переключения **instance/class**, предоставляющие взаимно исключающий выбор, являются экземплярами класса **RadioButton**, сгруппированными внутри экземпляра класса **GroupPane**.

Графические и анимационные панели позволяют отображать всевозможную графику. Так как эти панели не используются в стандартных окнах среды *Smalltalk Express*, мы их рассмотрим позже, при изучении графических возможностей системы *Smalltalk Express*.

Прокрутка

Есть много примеров, когда содержимое, которое нужно отобразить в панели окна, требует для своего отображения больше места, чем может предоставить соответствующая панель. Можно рассматривать панель как область просмотра, которая позволяет сразу увидеть только часть из всего содержимого панели. Прокрутка позволяет перемещать область просмотра вдоль «виртуального пространства», включающего все содержимое панели.

Панели, позволяющие сделать выбор из отображаемого ими списка, называют списковыми панелями. Окна, имеющие списковые панели, называют окнами просмотра (браузерами). Окно просмотра иерархии классов (**Class Hierarchy Browser**) — пример окна с тремя списковыми панелями, текстовой панелью и панелью переключателей экземпляра/класс (см. рис. 3.3).

Smalltalk Express, как и большинство приложений Windows, поддерживает работу с полосами прокрутки. Как альтернативу работы с мышью, можно проводить прокрутку, пользуясь на клавиатуре клавишами **[PgUp]** и **[PgDn]**, **[Home]** и **[End]** или перемещая текстовый курсор клавишами со стрелками.

Пользовательские меню

Меню — стандартный способ представления исполняемых операций с окнами системы Смолток, панелями окон и их содержимым. В *Smalltalk Express* используется как типичный для Смолтока подход с контекстными меню (меню панели)², так и привычный подход Windows со строкой меню окна, в которой представлены имена меню, доступные окну в целом и каждой панели в отдельности. Каждое из них имеет список команд (или пунктов меню) для выбора. Многие команды можно вызывать, не обращаясь к меню, а используя клавиатурные сокращения. В большинстве случаев они стандартны для системы Windows. Этот способ мы далее будем отмечать только для наиболее важных команд.

Стандартные меню — File, Edit, Smalltalk. Они одинаковы во всех окнах системы *Smalltalk Express* (кроме окна Transcript). В строке меню могут появляться специальные меню в зависимости от функциональности окна. Например, в окне просмотра иерархии классов имеются меню Classes, Variables, Methods. Обычно это еще один способ доступа к контекстным меню панелей окна.

Работа с меню в *Smalltalk Express* происходит так же, как и в других приложениях Windows. Начиная с Windows 95, стандартными стали и контекстные меню. Поэтому мы будем описывать только функциональность меню, а не способы работы с ними.

Управление окнами

Окна обычно открываются из меню, хотя можно открыть окно и выполняя выражение, которое произведет определение, создание и активацию окна. Как пример, откроем новое рабочее окно — окно Workspace. Для этого в системном окне Transcript выберем в меню File ▷ Workspace. Откроется и

² Реализации Smalltalk-80 для рабочих станций фирмы Xerox использовали трехкнопочную мышь с разноцветными кнопками. Левая кнопка (красная) использовалась для указания и выбора объектов; средняя (желтая) кнопка вызывала контекстное меню для выбранного объекта; правая (синяя) — меню рамки (окна). Так как в Windows обычно используются двухкнопочные мыши, то функции красной кнопки достались левой кнопке, желтой — правой, а функции синей пространственно выделены в системное меню окна.

станет активным новое рабочее окно. Оно обладает всеми свойствами окна среды MS Windows. Рабочее окно, как подсказывает само его имя, является «оперативной» площадкой для работы с приложениями, а также стандартным текстовым редактором системы. Будучи открытым, рабочее окно сохранится в среде Смолток, если образ системы был сохранен в конце сеанса работы. Пользуясь меню панели рабочего окна, содержащийся текст можно сохранить на диске в отдельном файле, а потом работать с ним вне среды *Smalltalk Express*, используя привычный текстовый редактор.

Когда выбирается пиктограмма системного меню, на экране возникает меню с пунктами, позволяющими воздействовать на состояние окна в целом (размеры окна и его расположение). Кроме действий, стандартных для всех окон Windows, системное меню выполняет в *Smalltalk Express* следующие дополнительные операции:

ZoomText — расширяет активную текстовую панель окна до размера всего окна; если панель уже была увеличена, эта команда возвращает панель в первоначальное состояние. В текстовых окнах (**Workspace**, **Transcript**) ничего не изменяет.

Fonts... (Шрифты...) — открывает диалоговое окно, которое позволяет выбрать шрифт для исходного окна.

Особое положение окна **Transcript** выражается в дополнительном пункте его системного меню:

Exit Smalltalk/V... (Выйти из Smalltalk/V...) — выбор этого элемента заканчивает текущий сеанс работы системы *Smalltalk Express*; при этом вас спросят о том, желаете ли вы сохранить текущее состояние системы (образ системы) прежде, чем сеанс завершится.

По стандартным соглашениям Windows каждое окно должно содержать меню **File**, служащее для общего управления системой. В *Smalltalk Express* это меню предоставляет следующие возможности:

New Workspace (НовоеРабочееОкно) — открывает пустое рабочее окно.

Open... (Открыть...) — запрашивает имя текстового файла и открывает его для редактирования в новом рабочем окне.

Install... (Установить...) — устанавливает в смолтоковскую систему код из файла специального вида, который определяется пользователем в процессе диалога.

Save (Сохранить) — сохраняет (фиксирует) изменения в текущей текстовой панели. В окнах просмотра и инспекторах может выполнять дополнительные функции, например, компиляцию метода.

Save As... (Сохранить как...) — сохраняет содержимое текущей текстовой панели в дисковом файле, который определяется пользователем.

Browse Classes (Просмотр классов) — открывает окно просмотра иерархии классов системы.

Browse Disk (Просмотр диска) — открывает новое окно просмотра дисков.

Print (Печатать) — печатает содержимое текущей текстовой панели.

Restore (Восстановить) — восстанавливает состояние текущей текстовой панели в последнее сохраненное состояние.

Save Image. . . (Сохранить образ. . .) — сохраняет текущее состояние системы *Smalltalk Express* в файле образа.

Дополнительно окно Transcript предоставляет следующие пункты меню:

Graphics Demo (Графическая демонстрация) — открывает окно, показывающее некоторые графические возможности системы.

About V. . . (О системе. . .) — открывает окно информации о системе, содержащее имя системы, номер ее версии и описание владельца авторских прав на систему.

Switch to MDI (Переключиться в MDI) — при установленной заплатке переключает среду в режим MDI (Multiple Document Interface).

3.1.2. Текстовый редактор системы

Подробно рассмотрим возможности, предоставляемые текстовыми панелями Системы *Smalltalk Express*.

Редактирование

Текстовая панель предоставляет программисту редактор, по своим возможностям аналогичный редактору Notepad (Блокнот). Создатели текстовой панели специально сделали ее клавиатурные команды совпадающими с командами Блокнота. Все текстовые панели используют буфер обмена Windows, что позволяет обмениваться данными не только между текстовыми панелями Смолтока, но и с другими приложениями. Операции редактирования доступны также из стандартного меню Edit (Правка).

Однако текстовая панель (и созданные на ее основе рабочие окна) все же отличается от Блокнота. Это связано с тем, что текстовая панель существует для редактирования текстового представления объектов системы и выполнения выражений языка Смолток.

Сохранение (Save) и восстановление (Restore)

Текстовый редактор системы *Smalltalk Express* всегда работает с текстовым представлением объекта системы. Это может быть файл, строка или некоторый смолтоковский код. Например, в любом рабочем окне обычно

редактируют объекты-строки, а в текстовой панели окна просмотра иерархии классов редактируют исходный код методов. Каждая панель, которая позволяет редактировать текст, кроме полного набора функций редактирования из меню Edit поддерживает функции Save (Сохранить) и Restore (Восстановить) из меню File.

Поскольку редактируется текстовое представление объекта, когда редактирование завершено, об этом необходимо сообщить системе, выбирая пункты File▷Save. При сохранении отредактированный текст преобразуется в новый объект, заменяющий собой старый. Например, если редактируется метод, то текст, содержащийся в панели, компилируется, а его исходный текст сохраняется в файле журнала системы.

Если необходимо, всегда можно восстановить первоначальное состояние отредактированного текста. Функция Restore отбрасывает все изменения, сделанные в активной текстовой панели, начиная с того момента, когда информация сохранялась в последний раз. Если редактируется строка, то текст строки панели заменится на текст исходной строки. Если редактируется метод, первоначальный текст метода вновь копируется в панель.

Выбор из меню File▷Save As... сохраняет копию текста из активной панели в дисковом файле, имя которого указывается с помощью открывающегося диалогового окна.

3.1.3. Выполнение выражений

В среде *Smalltalk Express* текст выражения можно вводить в любую текстовую панель, выполнять его и отображать результат, поскольку все текстовые панели поддерживают непосредственное выполнение выражений через меню с именем Smalltalk.

Чтобы выполнить выражение, необходимо сначала выделить его, а затем из меню Smalltalk (или из меню текстовой панели, которое содержит все пункты этого меню) выбрать или Show It (Показать), или Do It (Выполнить). Если выбирается пункт Show It, то выражение выполняется, и символьное представление возвращаемого объекта вставляется после выполненного выражения. Если выбирается пункт Do It, то выражение выполняется, но возвращаемое значение не выводится. Обратите внимание, что выполняется только выделенный текст, остальной текст панели игнорируется. Дополнительные пробелы в выбранном тексте также игнорируются.

В текстовой панели можно выделить и выполнить любое допустимое выражение или ряд выражений языка Смолток. Выполняемое выражение должно быть правильным *фрагментом* (chunk) смолтоковского кода, то есть удовлетворять тем же синтаксическим правилам, что и тело метода.

Когда выбираются и выполняются выражения, Смолток их сначала компилирует (переводит в байткоды), а только затем выполняет. Если при компиляции обнаруживается ошибка, *Smalltalk Express* вставляет в исходный текст в месте обнаружения ошибки краткое сообщение о ней. Для удобства сообщение об ошибке сразу же выделяется, так что простое нажатие клавиши [←] (или [Del]) удаляет его и тем самым восстанавливает исходное состояние текстовой панели. Устранив ошибку, можно попытаться выполнить код снова.

Совсем не обязательно заново набирать выполняемые выражения. Можно просто отредактировать некоторый уже существующий в панели текст, а затем его выбрать и выполнить. Поэтому имеет смысл создать окно с коллекцией полезных выражений.

3.1.4. Prompter и MessageBox

Классы Prompter (Подсказчик) и MessageBox (ОкноСообщения) представляют собой специальные диалоговые окна, позволяющие задать вопрос пользователю и получить от него ответ. Подсказчики в *Smalltalk Express* реализованы как диалоговые окна с однострочным редактором текста и строкой подсказки в виде текстового сообщения внутри окна. Чтобы открыть подсказчик, нужно послать классу Prompter одно из двух сообщений:

Prompter prompt: question default: answer.

Prompter prompt: question defaultExpression: answer.

В них каждый аргумент — строка. После того как окно откроется, строка `question` будет представлять заданный вопрос, а строка `answer` будет отображаться в текстовой панели как предлагаемый ответ по умолчанию. Первое сообщение возвращает введенную пользователем текстовую строку, в то время как второе сообщение возвращает объект, который является результатом *вычисления* введенного пользователем выражения.

Пользователь вводит свой ответ в единственную текстовую панель подсказчика. Текстовый редактор, используемый подсказчиком, почти ничем не отличается от текстового редактора, описанного выше, за исключением следующих особенностей:

- Диалоговое окно подсказчика является модальным окном.
- Клавиши управления курсором ← и → перемещают точку вставки (каретку) внутри поля редактирования подсказчика, а клавиши ↑ и ↓ позволяют циклически перемещаться по управляющим элементам диалогового окна.
- Подсказчик содержит кнопки [OK] и [Cancel]. После нажатия на кнопку [OK] или [Cancel] окно подсказчика закрывается, а управление пе-

редается объекту, вызвавшему его. Когда выбирается кнопка **[OK]**, подсказчик посылает текст из текстовой панели объекту, запросившему информацию. Нажатие на клавиатуре клавиши **[Enter]** равносильно выбору кнопки **[OK]**. Когда выбирается кнопка **[Cancel]** (или нажимается **[Esc]**), подсказчик посылает nil объекту, запросившему информацию.

Класс **MessageBox** позволяет получить от пользователя быстрый ответ типа «да/нет». Чтобы открыть окно сообщения, нужно послать классу **MessageBox** одно из двух сообщений:

MessageBox confirm: aString.

MessageBox message: aString.

Оба отображают окно сообщения со строкой aString в качестве заголовка. Первое сообщение создает окно с двумя кнопками Yes и No, которое возвращает соответствующий логический объект. Второе сообщение создает окно с единственной кнопкой OK и всегда возвращает false.

3.2. Специальные окна системы

Опишем специальные окна, которые система Смолток предоставляет разработчику приложений. От реализации к реализации изменяются сами окна, меняется число и расположение элементов в их меню, но основные принципы организации работы окна и выполняемые операции во многом похожи. В любой системе Смолток обязательно есть следующие окна:

Class Hierarchy Browser — окно просмотра иерархии классов. Отображает иерархию классов системы Смолток, позволяет просматривать и редактировать определения классов и методов (см. 3.2.2).

Class Browser — окно просмотра класса. Окно предоставляет возможность просматривать и редактировать методы, сконцентрированные в указанном при открытии окна классе (см. 3.2.3).

Inspector — окно инспектора. Позволяет исследовать структуру и данные объекта и редактировать данные (см. 3.2.4).

Walkback и **Debugger** — окно уведомления об ошибке и окно отладчика. Дают возможность посмотреть состояние программы в момент ошибки или останова; являются средствами отладки в системе Смолток (см. разд. 3.3).

Methods Browser — окно просмотра методов. Предоставляет возможность просматривать и редактировать список методов (см. 3.2.5).

Messages Browser — окно просмотра сообщений. Удобный инструмент просмотра сообщений, входящих в выражения, и исследования перекрестных ссылок (см. 3.2.6).

Disk Browser — окно просмотра диска. Это инструмент для работы с файловой системой. Позволяет просматривать и редактировать текстовые файлы, а также вычислять имеющиеся в них смолтоковские выражения (см. 3.2.1).

Все окна просмотра состоят, по крайней мере, из двух панелей: панели списка и текстовой панели. Выбор из списка отображает в текстовой панели информацию, связанную с выбранным элементом списка. Этот текст можно изменять и сохранять, тем самым, изменяя систему.

3.2.1. Окно просмотра диска

Окно просмотра диска предназначено для просмотра файловой системы и редактирования файлов. Вызывается при выборе из меню File ▷ Browse Disk или клавиатурной командой **[Alt]+[D]**.

Окно просмотра диска отображает пять панелей:

- панель дисков — содержит список имен (букв) доступных дисков;
- панель каталогов (directories) — позволяет просматривать дерево каталогов на выбранном диске;
- панель файлов — обеспечивает доступ к файлам в выбранном каталоге;
- панель сортировки файлов — позволяет через свое всплывающее меню (меню **SortBy**) сортировать файлы каталога по дате, размеру или имени;
- панель текста — позволяет просматривать и редактировать выбранный файл. Если файл не выбран, панель отображает расширенный список файлов в выбранном каталоге (показывая размеры и атрибуты файлов). Если выбран большой файл (больше 10К), эта панель показывает только его часть, вставив в начало соответствующее предупреждение. Чтобы в этом случае получить возможность просматривать и редактировать весь файл, надо в контекстном меню панели файлов выбрать Read Entire File.

Как принято в Смолтоке, часть полезной информации размещается в заголовке окна браузера. Пока не выбран диск, заголовок содержит только название окна. После выбора диска в заголовке появляется его метка тома, заключенная в квадратные скобки, полное имя текущего каталога (пока каталог не выбран, это просто имя диска) и свободное место на выбранном диске в байтах.

3.2.2. Открытие окна просмотра иерархии классов

Окно просмотра иерархии классов позволяет не только просматривать иерархию классов смолтоковской системы, но и редактировать определения классов и их методов.

Как открыть окно просмотра иерархии классов

Чтобы открыть окно просмотра иерархии классов, надо из меню любого окна системы выбрать File ▸ Browse Classes (Просмотр классов) или нажать клавиши **[Alt]+ [B]**.

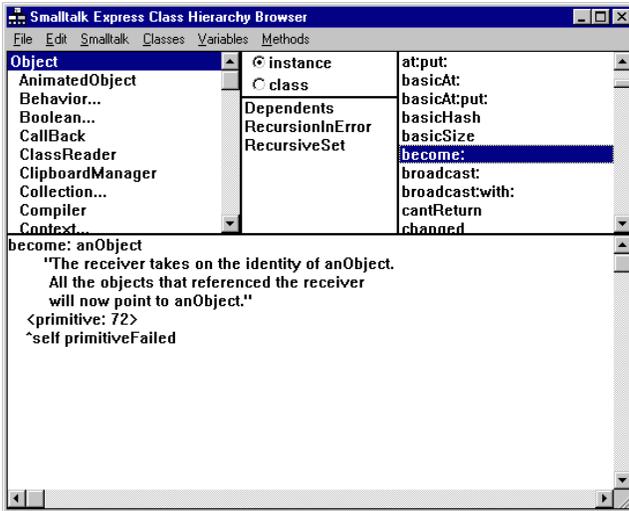


Рис. 3.3. Окно просмотра иерархии классов

Окно просмотра иерархии классов разделено на пять панелей.

Список, отражающий текущее состояние иерархии классов, появляется в левой верхней панели. В этой панели имена всех классов системы приводятся в иерархическом порядке. Класс **Object** является первым в этом списке, поскольку он — корневой, самый верхний класс иерархии. Имена подклассов отображаются с отступом относительно имени суперкласса; классы с общим суперклассом упорядочиваются по алфавиту. По умолчанию в этой панели видны только первые два уровня иерархии классов. Если у класса есть неотображаемые подклассы, после его имени ставится многоточие. Чтобы раскрыть (или, наоборот, скрыть) все подклассы данного класса, необходимо сделать двойной щелчок по имени класса либо выбрать пункт Hide/Show в контекстном меню панели.

Список переменных отображает средняя панель в верхней половине окна просмотра. Она расположена ниже пары радиокнопок instance/class (экземпляр/класс). Панель переменных показывает переменные выбранного класса (как определенные в нем, так и унаследованные по цепочке суперклассов). Наследование переменных показывается в обратном порядке: на

самом верху списка располагаются переменные, определенные в выбранном классе. Затем по очереди перечисляются переменные, определенные в суперклассах, начиная от непосредственного суперкласса и до класса `Object`. Перед переменными, определенными в конкретном суперклассе, в список включается строка вида “ – *ИмяСуперкласса* – ”. Переменные, определенные в одном суперклассе, перечисляются в алфавитном порядке. Содержимое панели переменных зависит от выбранной радиокнопки: если выбрана кнопка `instance`, отображаются и переменные экземпляра, и переменные класса, а если выбрана кнопка `class` — только переменные класса.

Панель методов расположена справа от панели списка переменных. В ней отображается (в соответствии с выбранной радиокнопкой) или список методов экземпляра, или список методов класса.

Сам смолтоковский код отображается в панели исходных текстов — в текстовой панели, занимающей нижнюю половину окна. Когда в панели иерархии классов выбирается класс, в панели исходных текстов отображается сообщение, посылка которого позволяет определить выбранный класс. Когда выбирается метод, в этой панели отображается исходный текст выбранного метода. Панель исходных текстов позволяет редактировать и определение самого класса, и исходный текст его методов. Иногда система может сообщить, что исходный текст метода недоступен: `selector can't access source code`. Такое возможно лишь в двух случаях:

- 1) При просмотре класса, поставляемого без исходного текста, например, `Compiler` (таких классов очень мало). В этой ситуации ничего сделать нельзя.
- 2) При просмотре класса пользователя, что свидетельствует о нарушении соответствия файлов образа и журнала (см. разд. 3.4).

Просмотр иерархии

Класс для просмотра можно выбрать в списковой панели иерархии классов — левой верхней панели окна. Используя прокрутку, можно просмотреть все классы в иерархии. Если выбрать класс, в панели исходных текстов отобразится определение класса, а списки переменных и методов, определенных в классе, отобразятся в панели переменных и в панели методов, соответственно.

Контекстное меню панели иерархии классов позволяет изменять иерархию: производить добавление и удаление подклассов, обновлять панель иерархии после сделанных изменений, открывать окно просмотра класса для выбранного класса, записывать в файл определение класса и определения его методов. Это меню аналогично меню `Classes` окна и содержит следующие пункты:

- Add Subclass...** (Добавить подкласс...) — позволяет добавить подкласс к выбранному классу (как это сделать, описано ниже).
- File Out...** (Вывести в файл...) — записывает определение данного класса со всеми его методами в файл. Файл создается в специальном формате файла регистрации изменений (см. 3.4.2). Имя файла запрашивается стандартным диалоговым окном сохранения файла.
- Update** (Обновить) — требует, чтобы окно просмотра иерархии классов повторно перечитало список классов в иерархии и отобразило его. Это надо делать всегда, когда иерархия классов изменяется другими окнами или программно.
- Browse** (Просмотреть) — открывает отдельное окно просмотра для выбранного класса. Это удобно, если необходимо при просмотре одного класса обратиться к другому.
- Hide/Show** (Скрыть/Показать) — переключатель, позволяющий показывать или скрывать подклассы выбранного класса. Если подклассы скрыты, после имени класса в панели иерархии классов появляется многоточие (...). В этой ситуации в меню **Classes** отображается пункт **Show Subclasses** (Показать подклассы). Чтобы показать подклассы, можно выбрать этот пункт. То же последует, если просто дважды щелкнуть на нужном классе. Если выбран класс, подклассы которого отображаются в панели, то в меню **Classes** стоит пункт **Hide Subclasses** (Скрыть подклассы). Выберите его или просто снова дважды щелкните на классе, чтобы свернуть иерархию. Если класс не имеет подклассов, эта функция будет недоступна.
- FindClass...** (НайтиКласс...) — вызывает простое диалоговое окно, позволяя ввести имя класса, который надо отыскать в иерархии. Этот пункт чрезвычайно полезен при поиске классов, которые содержатся в иерархии на уровнях, которые в настоящее время в панели не показаны.
- RemoveClass** (УдалитьКласс) — удаляет выбранный класс из системы.

Добавление нового класса

Чтобы добавить класс в систему, в панели иерархии классов надо выбрать класс, который станет суперклассом для нового класса. Затем из меню выбрать **Classes** ▸ **Add Subclass...**, который выводит на экран диалоговое окно с именем “Add a SubClass”, предоставляющее поле редактирования для ввода имени нового класса и группу радио-кнопок, которые позволяют выбрать тип создаваемого подкласса. Тип подкласса зависит от того, должны ли объекты, определяемые данным классом, содержать именованные переменные экземпляра, индексированные переменные экземпляра или

массивы байтов (см. с. 44). После того, как выбор сделан, создается новый класс. При этом список иерархии классов автоматически обновляется, новый класс выбирается в панели иерархии классов, а его определение отображается в текстовой панели.

Чтобы в классе определить переменные экземпляра, переменные класса и пулы, следует соответствующим образом отредактировать текст определения класса, вводя между кавычками нужные имена. Затем надо выбрать пункт File ▷ Save или нажать клавиши **[Alt]+[S]**. Класс, если это возможно (см. 3.2.2), изменится в соответствии со сделанными изменениями текста определения. При изменении определения класса система сама перетранслирует все методы в классе, все определения и методы его подклассов. Кроме того, в файл журнала change.log запишется определяющее новый класс сообщение.

Напомним, что панель исходных текстов, как всякая текстовая панель, имеет достаточно большое контекстное меню (вызывается при нажатии в панели правой кнопки мыши), содержащее все элементы меню Edit и Smalltalk, а также пункт Save. Таким образом, большинство операций можно выполнить, не обращаясь к меню окна.

Изменение определения класса ведет к немедленным последствиям, так что все будущие новые экземпляры класса будут иметь новую структуру. Следует быть очень осторожным при изменении тех классов, которые используются средой Смолток. Пока нет уверенности в том, что все сделано правильно, следует определять подклассы основных классов, а не изменять структуру существующих. Создание новых классов на основе существующих, без изменения стандартных классов Смолтока, считается хорошим стилем программирования.

Удаление класса и его экземпляров

Чтобы удалить класс, надо сначала в панели иерархии выбрать тот класс, который будет удаляться. Затем из меню Classes выбрать пункт RemoveClass (УдалитьКласс). Система попросит подтверждения на выполнение такой операции и, в случае его получения, удалит класс.³

Когда удаляется класс, автоматически удаляются все методы данного класса. Смолток не допустит удаления класса, если он имеет подклассы

³ Если в *Smalltalk Express* все же остается ссылка на удаленный класс, она будет указывать на этот класс как на `deletedClass` (удаленныйКласс), и он не будет удаляться сборщиком мусора. Это сделано в целях безопасности, хотя такие фиктивные ссылки и занимают место.

или в системе есть его экземпляры⁴. В этом случае при попытке удалить класс возникнет окно Walkback (см. с. 75), объясняющее причину ошибки.

Экземпляр класса удаляется системой автоматически, если на него больше нет ссылок. Поэтому часто бывает полезно выявить все объекты, которые обращаются к данному объекту. Это можно сделать, посылая последнему сообщение `allReferences`. Если нужно удалить экземпляры классов, связанные с окнами, то прежде чем что-либо делать, надо удостовериться, что в открытых окнах системы сохранена вся необходимая информация, и выполнить выражение `Notifier reinitialize`. При этом закроются все открытые окна, и повторно инициализируется только окно системы Transcript.

Если некоторые экземпляры все же остаются в системе, следует сначала сохранить образ системы (см. 3.4.1), а затем выполнить выражение

```
MyClass allInstances do: [:each | each release.  
                        each become: String new].
```

В результате произойдет переназначение объектных указателей с экземпляров удаляемого класса на объект `String new`, после чего сборщик мусора удалит эти объекты.

Список методов

Для выбранного класса в панели методов отображается список всех его методов (экземпляра или класса). Выбор имени метода в панели методов приводит к появлению исходного текста метода в текстовой панели. Контекстное меню панели методов аналогично меню `Methods` и содержит следующие пункты:

New Method (Новый Метод) — используется для того, чтобы добавить новый метод экземпляра или класса к выбранному классу. Процедура добавления новых методов описана ниже.

Senders (Отправители) — заставляет Смолток искать все те методы, которые отправляют сообщение с выбранным именем. Появляется окно просмотра методов, содержащее все методы в системе, которые посылают сообщение с выбранным именем.

Implementors (Реализаторы) — заставляет Смолток искать все методы в системе с выбранным именем. Появляется окно реализаторов — окно просмотра методов, содержащее все методы с выбранным именем.

Local Senders (Локальные отправители) — во всем подобен пункту `Senders`, за исключением того, что среда поиска отправителей ограничивается текущим классом и его подклассами.

⁴ В этом случае система не даст изменить и структуру класса.

Local Implementors (Локальные реализаторы) — во всем подобен пункту Implementors, за исключением того, что среда поиска реализаторов ограничивается подклассами текущего класса.

Messages (Сообщения) — заставляет Смолток собрать в список все имена методов, вызываемых в выбранном методе, и открывает окно SelectorBrowser, отображающее этот список.

File Out. . . (Вывести в файл. . .) — заставляет Смолток послать выбранный метод в файл, создаваемый в формате файла регистрации изменений.

Remove (Удалить) — удаляет из класса выбранный метод. Список методов сразу же модифицируется.

Меню панели переменных Variables работает вместе с панелью переменных и помогает сузить круг методов, отображаемых в панели методов. Если в панели переменных выбирается некоторая переменная, то список методов уменьшается до подмножества тех методов данного класса, которые ссылаются на выбранную переменную. Меню Variables позволяет точнее настроить выбор методов: кроме выбора списка методов, ссылающихся на выбранную переменную (Variables ▷ Both, по умолчанию), можно выбрать список методов, использующих (читающих) эту переменную (Variables ▷ Used), или список методов, присваивающих что-либо этой переменной (Variables ▷ Assigned). Текущий выбранный режим помечен «галочкой». В классах с большим числом методов сужение списка может быть очень полезно для быстрого поиска нужного метода или методов.

Добавление, изменение и удаление метода

Чтобы увидеть исходный текст метода, следует выбрать нужный метод в панели методов. Панель исходного текста отобразит текст метода. Чтобы изменить текст метода, требуется отредактировать исходный текст в текстовой панели, а затем выбрать функцию Save. Можно редактировать текст любого существующего в классе метода.

Имя нового метода принимается из первой строки текста. Но если вы изменили имя метода и провели операцию сохранения, то будет создан новый метод с новым именем, а первоначальный метод останется без изменений. Это полезно при создании внутри класса разновидностей метода.

Чтобы добавить в класс новый метод, сначала следует выбрать, какой метод (экземпляра или класса) будет создаваться, а затем выбрать пункты меню Methods ▷ New Method. В текстовой панели появится доступный для редактирования шаблон метода. Когда текст нового метода создан, следует выбрать пункты меню File ▷ Save (или **[Alt]+[S]**), вызывая компилятор и устанавливая новый метод в систему. Если в процессе трансляции добав-

ляемого или изменяемого метода будет обнаружена ошибка, ее надо исправить и снова использовать File ▸ Save.

Когда метод успешно откомпилируется, система Смолток автоматически установит его в класс и все будущие вызовы этого метода будут использовать его новую версию. Исходный текст нового или измененного метода будет сохранен в журнале системы.

Чтобы из класса удалить существующий метод, необходимо выбрать его в панели методов, и выбрать пункты меню Methods ▸ Remove. Список методов немедленно изменится, показывая, что выбранный метод удален.

3.2.3. Окно просмотра класса

Окно просмотра класса можно открыть одним из двух способов: послать классу сообщение `edit`, или в окне просмотра иерархии классов выбрать нужный класс, а затем выбрать пункт `Browse` из меню `Classes`.

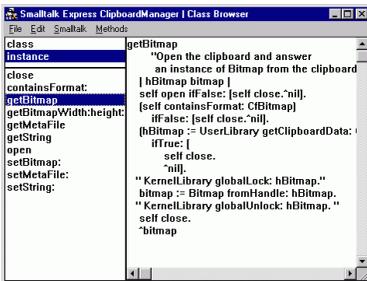


Рис. 3.4. Окно просмотра класса

Левая верхняя панель определяет просматриваемый словарь методов и позволяет сделать всего два выбора: выбрать строку `class` (класс) или строку `instance` (экземпляр). Если выбирается строка `class`, то в панели методов, расположенной ниже первой панели, отображается список методов класса, если выбирается строка `instance` — список методов экземпляра. Если выбирать в этой панели метод, с его исходным текстом можно работать в панели исходных текстов — самой большой панели, расположенной справа от списковых панелей. Чтобы добавить в класс новый метод, в этом окне достаточно только выбрать нужный словарь (`class` или `instance`), а затем, не выбирая метода, в панели исходных текстов набрать текст нового метода и сохранить его. Имя (селектор) нового метода немедленно появится в панели методов. Напомним, что имена методов располагаются в списковой панели методов в алфавитном порядке.

Например, можно сделать следующее: переместить курсор в любую текстовую панель, набрать в ней выражение `Collection edit` и выбрать его, из меню `Smalltalk` текстовой панели выбрать пункт `Do it`. Окно просмотра откроется на классе `Collection`. На просматриваемый класс указывает заголовок открывшегося окна. В окне можно просматривать, добавлять и изменять методы класса и экземпляра только выбранного класса.

3.2.4. Инспекторы

Окно Inspector (Инспектор) используется как инструмент для исследования и изменения данных любого объекта системы. Чтобы открыть инспектор для объекта, надо послать этому объекту сообщение `inspect`. Например, чтобы открыть инспектор на объекте `Display pen` (на пере экрана), надо выполнить выражение `Display pen inspect`. Можно поступить и по-другому: выбрать в любой текстовой панели имя объекта, который надо просмотреть (например, `Display pen`), и выбрать пункт `InspectIt` из меню `Smalltalk`. Таким способом можно выбирать и просматривать объекты почти из любого контекста.

Заголовок окна инспектора содержит указание на класс осматриваемого объекта. Само окно имеет две панели. Левая списковая панель показывает сам объект (`self`) и все его переменные, включая унаследованные. Если объект имеет индексированные переменные, то они перечисляются последними с указанием числовых индексов. Правая текстовая панель отображает текстовое представление значения выбранной переменной. При выборе `self` отобразится значение inspectedируемого объекта.

Если выбрать пункты `Inspect > Inspect` (или сделать двойной щелчок на переменной), то откроется новое окно инспектора для значения текущей переменной.

Панель исходных текстов переменной экземпляра — текстовая панель. Чтобы вызвать нужные функции, можно использовать соответствующие пункты из меню `File`, `Edit`, `Smalltalk` или воспользоваться пунктами из контекстного меню текстовой панели. Панель можно использовать и для выполнения любого желаемого выражения. Есть две очень важные особенности этой панели:

- Любое выполняемое выражение компилируется в контексте, определяемом осматриваемым объектом. Это означает, что в выражениях можно использовать имена всех переменных экземпляра.
- Если выбирается пункт `Save`, все содержимое текстовой панели компилируется и выполняется, а результат заменяет текущее значение выбранной переменной экземпляра. Если выбирается пункт `Restore` (Восстановить) из меню `File`, то Смолток отобразит в текстовой панели текущее значение выбранной переменной экземпляра.

При инспектировании словарей возникают некоторые особенности. Напомним, что словарь — объект, который связывает с уникальным объектом-ключом некоторый объект-значение. Точно так же, как и обычные инспекторы, инспекторы словарей имеют те же две панели, однако, в списковой панели перечисляются ключи словаря, а не переменные объекта. Когда в

этой панели выбирается ключ, связанное с ним значение отображается в текстовой панели. Чтобы увидеть это, откройте инспектор для словаря `ColorConstants`, выполняя выражение `ColorConstants inspect`. Обратите также внимание на то, что в списковой панели инспекторов словарей отсутствует `self`, а к строке меню окна добавляется меню `Dictionary` (Словарь). Меню `Dictionary` содержит три пункта:

`Add` — добавляет в словарь новый элемент. Система открывает диалоговое окно для ввода нового ключа. Значение, связанное с новым ключом, равно `nil`, но его можно изменить в текстовой панели и сохранить.

`Remove` — удаляет из словаря выбранный ключ (и значение).

`Inspect` — создает окно инспектора на значении выбранного ключа.

3.2.5. Окно просмотра методов

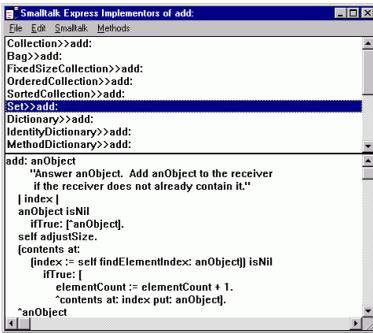


Рис. 3.5. Окно просмотра методов

Выбор пункта `Implementors` в меню любого окна, предлагающего этот пункт, открывает окно просмотра методов на списке всех методов, которые реализуют метод с таким именем. Аналогично, но только в классе и его подклассах, работают пункты `Local Senders` и `Local Implementors`. Поэтому окно просмотра методов часто называют окном отправителей или окном реализаторов, в зависимости от отображаемой окном информации.

Окно просмотра методов имеет две панели: панель методов в верхней части окна и текстовую панель в нижней его части. Панель методов отображает список методов, идентифицированных строкой вида `ИмяКласса>>имяМетода`. Когда в списке выбирается метод, его текст отображается в текстовой панели. Текстовая панель позволяет редактировать исходный текст выбранного метода. Когда отредактированный текст сохраняется, метод транслируется заново.

Окно (`Methods Browser`) позволяет просматривать и редактировать список методов. Есть четыре основанных на меню способа открыть окно просмотра методов. Если в какой-либо панели методов есть выбранный метод, то выбор пункта `Senders` в меню любого окна, предлагающего этот пункт (например, меню `Methods` окна просмотра иерархии классов), открывает окно просмотра методов на списке всех методов системы, которые посылают сообщение с выбранным именем. Выбор

Меню `Methods` в строке меню окна имеет шесть пунктов, пять из которых повторяют пункты из одноименного меню окна просмотра иерархии классов. Но есть и новый пункт:

`Remove from List (Удалить из Списка)` — обеспечивает удобный способ отбросить те записи из панели методов, которые в данный момент не нужны; удаление записи не удаляет метод из класса.

3.2.6. Окно просмотра сообщений

Окно просмотра сообщений (`Messages Browser`), которое можно открыть, выбирая пункт `Messages` из меню `Methods` любого окна, предлагающего его, имеет три панели, позволяющие полностью идентифицировать сообщение для каждого селектора в методе, для которого было открыто данное окно.

Левая верхняя списковая панель отображает селекторы всех сообщений, представленных в методе, отображаемом в нижней текстовой панели окна просмотра. Кроме того, окно имеет еще одну списковую панель (правую верхнюю), благодаря которой можно просмотреть список методов (отправителей или реализаторов) для селектора, выбранного в панели селекторов. Если в панели селекторов произвести выбор, то в тексте метода, отображаемом в нижней панели, выделится сообщение, связанное с этим селектором. Это может оказаться полезным при попытке идентифицировать сообщения, в которых присутствуют сложно вычисляемые аргументы.

Если для выделенного в левой верхней панели селектора найти его отправителей или реализаторов, то соответствующий список появится в правой верхней панели, а выделение в ней любой строки приведет к отображению в нижней панели текста соответствующего метода. Все меню, расположенные в строке меню, нам знакомы по другим окнам.

3.3. Окна, предназначенные для отладки

Если бы другие не делали ошибок, откуда бы я узнал, что поступаю правильно?!

П.С. Таранов

Даже опытные программисты время от времени делают ошибки в создаваемом ими коде. Что уж говорить о неопытных! К счастью, интерактивный характер среды разработки системы Смолток позволяет достаточно просто и изящно организовать поиск и исправление ошибок.

Основными инструментами для отладки являются окна уведомлений, окна-инспекторы и отладчик. Об окнах-инспекторах мы уже рассказывали. Остановимся на двух окнах, непосредственно предназначенных для отладки. Первым, в случае обнаружения ошибки времени выполнения, автоматически появляется на экране *окно уведомлений* с именем Walkback. Если информации, содержащейся в этом окне, не достаточно для выяснения причины ошибки, следует открыть второе окно — окно Debugger, выбирая пункт Debug (Отладить) из меню Walkback окна Walkback или нажимая кнопку Debug, размещенную в текстовой панели окна Walkback. Окно Debugger предоставляет больше информации о случившемся. В *Smalltalk Express* эти важные окна легко различимы, они определены с разным цветным фоном: красным для окна Walkback и желтым для окна Debugger.

Следует отметить, что в смолтоковском коде встречаются три разных вида ошибок. Первый вид ошибок состоит в том, что в смолтоковском выражении встречается синтаксически неправильный код или используются имена, не определенные в этом контексте. Об этом типе ошибок мы уже говорили (см. 3.1.3). Дополнительно заметим, что компилятор (вызываемый всякий раз, когда вызывается команда Save) не позволит добавить в систему код с такими ошибками. В этом разделе мы ошибки подобного рода рассматривать не будем.

Второй вид ошибок происходит тогда, когда ваш код выполняется, и проблемы возникают во время выполнения. Другими словами, система находит нечто, что она не может или не желает выполнять. Именно в такой ситуации возникает окно уведомлений Walkback, сообщая, в чем состоит проблема. Поиск и исправление именно таких ошибок и составляет предмет дальнейшего разговора.

Третий вид ошибок составляют ошибки, которые возникают при правильном коде, но неправильном проекте. Ваша программа прекрасно выполняется, но не делает того, что должна была бы делать. Это может произойти, например, потому, что вы не поняли, как работает некоторый механизм в существующем системном классе, или потому, что неправилен сам проект, ошибочна реализация одного из ваших классов или одного из ваших методов. Ниже мы увидим, как проследить за выполнением вашего кода и кода системы, отыскивая и такого рода ошибки.

Конечно, на практике крайне редко, но встречаются ошибки и в стандартной библиотеке классов. Если такая ошибка вдруг возникнет, и вы поняли основные идеи технологии поиска ошибок, то сможете найти и ее. Но опыт показывает, что в современных смолтоковских средах плодотворнее искать ошибки в своем коде.

3.3.1. Окно Walkback

Окно уведомления об ошибке Walkback появляется на экране тогда, когда происходит хотя бы одно из следующих событий:

- некоторому объекту послано сообщение `halt` (при этом говорят, что установлена программная контрольная точка), например, `self halt`;
- нажаты клавиши **[Ctrl]+ [Break]**;
- объекту послано сообщение `error:`, которое использует в качестве параметра строку, описывающую ошибку;
- во время выполнения смолтоковского кода возникла ошибка в виртуальной машине Смолтока.

Окно Walkback уже содержит некоторую полезную информацию. Можно начинать поиск причины его появления. Но сначала — небольшое «лирическое» отступление. Отладка в системе Смолток имеет много общего с отладкой в других языках программирования. Везде применяются одни и те же общие правила:

Правило первое: прочитайте сообщение об ошибке. Любая система может сообщить только непосредственную и поверхностную причину возникшей проблемы, а не ее первопричину. Но все же внимательно прочитайте то, что говорится в сообщении об ошибке. Помните, что система пытается сообщить вам *нечто важное*.

После того, как вы прочитали сообщение об ошибке, не торопитесь. По крайней мере, постарайтесь понять то, что сообщает система. Подумайте, в чем могла бы состоять причина. Отладка — одна из тех вещей, где весьма важно постараться сразу же определить причину. Если вы не очень внимательны, можно ходить вокруг да около, пробуя разные, не относящиеся к сути, способы решения, не приближаясь к самой проблеме ни на шаг.

Правило второе: ничего не предполагайте заранее. Только то, что вы «абсолютно» уверены, что «эта переменная» в конкретный момент времени должна иметь «такое» значение или что управление передается «именно в эту часть» исполняемого кода, еще не означает, что все именно так. Проверьте, чтобы убедиться в собственной правоте. Как и многое другое, Смолток позволяет сделать это достаточно просто.

Вернемся к вопросам отладки в системе Смолток и попытаемся выполнить в окне Workspase выражение `'Привет' at: 8`, в котором происходит обращение к восьмой букве строки `'Привет'`. Таковой в строке нет. Появится окно Walkback с заголовком: `"8 is outside of collection. . ."` (`"8 вне пределов набора. . ."`). Заголовок окна Walkback пытается описать возникшую ошибку. Поэтому в качестве заголовка вы всегда будете получать «имя»

возникшей ситуации: “Message not understood” (“Сообщение не понято”), “Division by zero” (“Деление на нуль”) и т.д. Кроме того, в панели окна отобразится список контекстов выполнения методов (в *Smalltalk Express* — стек вызовов методов), в котором перечислены все выполнявшиеся, но еще не завершённые методы. Каждая строка в панели представляет один метод, при этом первым стоит метод, выполнявшийся самым последним. В строке сначала написано имя класса объекта, получившего сообщение, а затем, после ‘>>’, селектор самого сообщения. Если используемый метод определен в суперклассе получателя, имя класса, в котором определен метод, отображается в круглых скобках. Иногда строка будет иметь вид [] in ClassName >> methodName. Это означает, что ошибка произошла во время выполнения блока, расположенного в методе methodName из класса ClassName. Поскольку система Смолток не делает никаких различий между «системной библиотекой» и «вашим кодом», список контекстов может состоять из смеси ваших методов и методов из библиотеки классов.

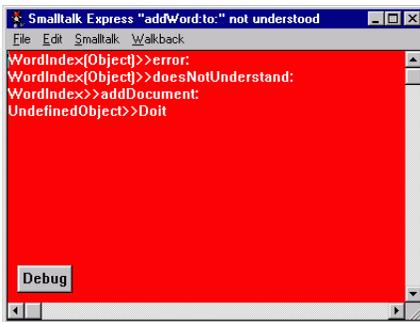


Рис. 3.6. Окно уведомления об ошибке

как правило, означает, что в вашем коде было сделано нечто такое, что привело к возникновению некоторой проблемы, которая не обнаруживала себя до тех пор, пока система не попыталась выполнить запрещенную в ее собственном коде операцию (подобную доступу к ключу, которого нет в указанном словаре).

Список контекстов можно просматривать снизу вверх, следуя тем путем, которым система выполняла код. Если наверху находятся методы системы, переместитесь ниже (обратно по времени), чтобы найти место, где начинается ваш код. Но, как уже было сказано, ваш код мог создать «плохой объект» намного раньше, и в счастливом неведении передать его системе, где он и вызвал ошибку. Если метод, который приводит к ошибке, уже *возвратил объект*, вы его не увидите в стеке вызовов. В этом случае, чтобы

Бывает так, что правильный кусок системного кода запускает ваш код, содержащий ошибку. Встречается и другая ситуация: когда ваш код вызывает некоторый метод из библиотеки классов, а тот, в свою очередь, вызывает множество других методов из библиотеки классов, прежде чем, в конечном счете, в одном из системных методов не возникнет исключительная ситуация. Это не означает, что вы нашли ошибку в самой системе. Это,

найти ошибку, вы должны исследовать контекст (явно вызывая отладчик) или в некотором месте прервать исполнение вашего кода, а затем проследить за его дальнейшим «пошаговым» выполнением. Как это сделать, мы коротко рассмотрим в следующем разделе, когда будем изучать второе окно для отладки.

Информация в окне Walkback рассмотрена, и теперь предстоит сделать одно из трех:

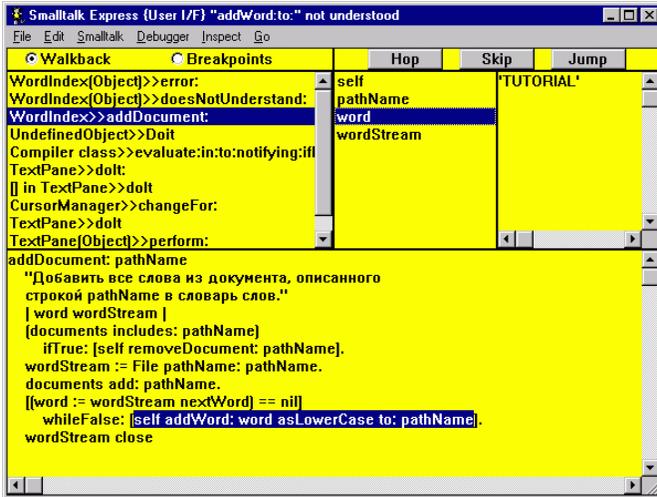
- Определить, в чем проблема, опираясь только на информацию, содержащуюся в окне Walkback, сразу закрыть окно Walkback и перейти к решению возникшей проблемы.
- Продолжить выполнение кода от точки прерывания, если окно Walkback возникло либо в результате прерывания по комбинации клавиш **[Ctrl]+[Break]**, либо потому, что было послано сообщение `halt`; в этих случаях с программой все в порядке, так что можно выбрать пункт Resume (Продолжить) из меню Walkback, после чего окно Walkback закроется и выполнение кода продолжится.
- Прийти к выводу, что нужна более подробная информация о происшедшем, и воспользоваться окном Debugger, для чего или выбрать пункт Debug (Отладить) из меню Walkback, или нажать кнопку Debug; в результате окно Walkback закроется, а окно Debugger появится на экране.

3.3.2. Окно Debugger

Окно Debugger (Отладчик) расширяет возможности окна Walkback по изучению контекста выполнения методов, вызвавших проблемы, и, что очень важно, позволяет управлять процессом выполнения кода.

Окно имеет четыре панели и пять кнопок. Левая верхняя списковая панель служит двум целям: представлению цепочки контекстов на момент останова или перечислению контрольных точек. Если нажата радиокнопка Walkback, расположенная выше этой панели, то верхняя левая списковая панель содержит тот же список, что и окно Walkback, вызвавшее данное окно Debugger. Когда в этой панели выбирается строка, другие панели окна отображают связанную с выделенной строкой информацию о контексте выполнения. Если нажата радиокнопка Breakpoints (Контрольные точки), то верхняя левая списковая панель состоит из строк с именами классов и методов, содержащих контрольные точки. Когда в этой панели выбирается строка, панель, расположенная ниже, отображает исходный текст выбранного метода.

Панель в нижней части окна всегда отображает исходный текст выбранного метода. Если выбрана радиокнопка Walkback, в исходном тексте вы-

Рис. 3.7. Отладчик системы *Smalltalk Express*

бранного метода выделяется то посланное сообщение, выполнение которого не завершилось. Эта панель служит текстовым редактором, с помощью которого можно работать с кодом метода так же, как в и окне просмотра иерархии классов. Например, можно изменить метод и сохранить его. При этом все строки левой верхней списковой панели, расположенные выше самого нижнего вхождения измененного метода в список, отбрасываются, поскольку изменился метод, из-за которого они попали в этот список.

Но если в это же время на том же методе открыто другое окно просмотра, и впоследствии вы перейдете в это окно и, используя его, сохраните метод, вы возвратите ошибку обратно в код. Это произойдет потому, что окна просмотра автоматически себя не обновляют, когда код, который они показывают, изменяется в другом месте.

Две панели наверху справа служат окном-инспектором для получателя сообщения, аргументов и временных переменных выбранного метода. Панель имен переменных, левая из двух панелей окна-инспектора, содержит `self`, представляя получателя, имена всех аргументов и временных переменных. Самая правая текстовая панель отображает значение выбранной из левой панели переменной и позволяет редактировать его. Двойной щелчок на элементе в панели имен переменных откроет окно инспектора на выбранном объекте. То же можно сделать, выделяя имя переменной и выбирая пункты `Inspect` ▷ `Inspect`. Например, можно просмотреть переменные экземпляра получателя сообщения, дважды щелкая на `self`.

Панель меню, кроме известных, содержит и новые меню `Debugger` и `Go`. Меню `Go` дублирует кнопки `Hot`, `Skip` и `Jump`, о которых речь ниже, а меню `Debugger` содержит пункты:

`Resume` (Продолжить) — как и в окне `Walkback`, позволяет продолжить выполнение кода после сообщения `halt` или прерывания **[Ctrl]+ [Break]**; окно `Debugger` исчезает, и выполнение продолжается до следующей точки прерывания; окно не позволит продолжить выполнение, если был изменен метод из `Walkback`-списка.

`Restart` (Перезапустить) — если был выбран `Walkback`-метод, то окно `Debugger` исчезает, и выполнение кода повторится, начиная с выбранного метода, посредством посылки соответствующего сообщения, возможно, с измененными данными.

`Senders` (Отправители) — как и в окне просмотра иерархии классов, открывает окно просмотра методов, которое будет содержать все методы системы, которые посылают выбранное сообщение.

`Implementors` (Реализаторы) — как и в окне просмотра иерархии классов, открывает окно просмотра методов, которое будет содержать все методы системы с тем же самым именем, что и выбранный.

`Add Breakpoint` (Добавить контрольную точку) — открывает диалоговое окно, которое запросит у пользователя имя класса и имя метода, куда необходимо вставить контрольную точку.

`Remove Breakpoint` (Удалить контрольную точку) — позволяет пользователю выбрать контрольную точку, которая затем будет удалена из списка контрольных точек.

`More Levels` (Больше уровней) — инициализирует, если такое возможно, дополнительные строки, которые будут включены в списковую панель `Walkback`.

Как мы отмечали, иногда в отладчике нельзя увидеть фактическую причину проблемы, поскольку реальная причина находится в методе, который был вызван, отработал и уже возвратил объект. Вот в этом случае вы можете установить в коде контрольные точки, а затем проследить за последующим выполнением кода, пытаясь отыскать ошибку. Для этого надо только вставить в нужном месте текста выражение `self halt`. Это сообщение реализовано в классе `Object`, так что каждый объект системы понимает его и делает одно и то же — открывает окно уведомлений `Walkback`.

Когда останов произошел, вы можете просто продолжить выполнение, но, более вероятно, раз уж здесь вы поставили контрольную точку, вы откроете окно отладчика. Ведь в нем можно просматривать значения переменных и посылать сообщения. А самое главное — можно использовать

кнопки `Hor`, `Skip` и `Jump`, исследуя выполнение кода в пошаговом режиме. Функции кнопок также доступны через одноименные пункты из меню `Go`. Работают кнопки следующим образом:

`Hor` — пошаговая посылка сообщений: или посылает одно сообщение языка Смолток, или производит одно назначение.

`Skip` — пошаговая посылка сообщений без захода в код выполняемых при этом методов.

`Jump` — переход на следующую контрольную точку.

Рассматривая пример по отладке класса, мы вернемся к этим кнопкам. Сейчас же отметим, что иногда трассировка программы может оказаться более эффективной, чем отладка с использованием окон `Walkback` и `Debugger`. Для целей трассировки смолтоковская система предоставляет возможность вывода информации в системное окно `Transcript` с помощью выражений, которые вставляются в методы на время отладки. Чтобы напечатать нужную информационную строку в окне `Transcript`, используются выражения, подобные следующим:

```
Transcript show: 'Here is a method next'; cr.
```

```
Transcript show: 'anArray is ', anArray printString; cr.
```

Можно придумать и другие способы регистрации происходящего, например создать файл и записывать в него соответствующие сообщения. Позже в любой момент можно просмотреть файл и увидеть, что и в какой последовательности происходило во время работы программы.

3.4. Управление системой в целом

Система *Smalltalk Express* фактически состоит из динамических библиотек — нескольких специализированных `dll`-файлов, которые содержат виртуальную машину Смолтока, примитивные методы и обеспечивают поддержку памяти объектов⁵, а также из следующих четырех файлов:

`vw.exe` — очень маленький файл, который содержит стартер среды разработки; в качестве параметра может передаваться имя выполняемого файла, содержащего образ пользователя (по умолчанию `v.exe`);

⁵ В систему можно добавлять новые примитивные методы, добавляя новые библиотеки в виде `dll`-файлов. Естественно, это не вполне обычные по формату файлы, поскольку есть специальные требования к формату экспортируемых функций, реализующих примитивные методы. Кроме того, система может делать внешние вызовы к любым функциям, экспортируемым другими библиотеками динамической компоновки.

v.exe — файл, который содержит стартер приложения пользователя и последний сохраненный образ системы (этот файл изменяется всякий раз, когда сохраняется образ между сеансами работы);

change.log — журнал системы, в котором во время работы регистрируются все изменения исходного кода и выполнение выражений;

sources.sml — файл-хранилище, содержащий стабильные исходные тексты смолтоковской системы.

3.4.1. Системные файлы и сохранение образа

Образ системы — это совокупность всех объектов. Когда система запускается, состояние образа считывается из файла образа, который может задаваться пользователем (по умолчанию v.exe), в память объектов, создавая образ системы, оживляемый виртуальной машиной Смолтока.

Так как Смолток — интерактивно модифицируемая среда, образ в процессе работы пользователя над приложениями постоянно изменяется. Изменения не записываются на диск, пока этого явно не сделать, выбирая пункты File ▷ Save Image.

Можно сохранить образ и при выходе из системы, что происходит, когда выбирается пункт Exit Smalltalk/V... из системного меню окна Transcript. При этом возникает диалоговое окно, в котором спрашивается: надо ли сохранить образ с произведенными изменениями? Если в ответ на вопрос Save Image? (Сохранить образ?) выбрать Yes, происходит сохранение полного состояния системы, включая расположение и содержимое всех окон. В следующий раз, когда система будет запускаться, она возникает в том виде, в каком она находилась в момент сохранения образа. Если выбрать No — все изменения, сделанные в среде Смолток во время текущего сеанса работы, будут отброшены (забыты), но, несмотря на это, файл change.log их все же запомнит. Когда в следующий раз Смолток будет запущен, он возникнет в предыдущем сохраненном состоянии, игнорируя все изменения, сделанные после сохранения образа. Если же выбрать Cancel (Отменить), то произойдет возврат в среду Смолток.

Таким образом, файл v.exe представляет последнюю сохраненную версию среды. Он же, в случае аварийного отказа системы или серьезной ошибки, становится отправной точкой для восстановления работоспособности системы. Поэтому всегда следует хранить резервную копию файла v.exe, но не одну, а вместе с тесно связанными с ним файлами sources.sml и change.log. Дело в том, что во время работы Смолток в файле v.exe поддерживает указатели из компилируемых методов среды на последние версии исходного текста методов. Если это исходный текст инструментов среды

разработки или базовых классов — указатель направлен на файл `vwsrsc20.dll`. Иначе — на файлы `change.log` и `sources.sml`. Как результат, при просмотре в классе методов происходят обращения к диску. Следовательно, файл `v.exe`, меняясь сам, содержит указатели и на меняющиеся файлы `change.log` и `sources.sml`. Именно поэтому следует сохранять эти три файла вместе.

3.4.2. Файл журнала

Когда происходит определение новых или изменение существующих классов и методов, Смолток автоматически регистрирует все эти изменения в файле журнала системы `change.log`, который формируется из последовательных порций кода, ограниченных восклицательными знаками `!` (его формат очень похож на описанный в статье Г. Краснера из [18]). По этой причине файл журнала *нельзя* изменять. Его можно просматривать, пользуясь функцией `Open...` из меню `File`, можно, пользуясь им, повторно устанавливать в систему методы и определения классов. Именно эта возможность используется при восстановлении системы, если она разрушилась.

В журнале также автоматически регистрируются все важные для системы события. Каждый раз при сохранении образа в файл `change.log` записывается сообщение с указанием даты и времени операции. Каждое выражение, которое выполняется с помощью пунктов `Do It` или `Show It`, также регистрируется. Каждый раз, когда из класса удаляется метод, регистрируется соответствующее сообщение. И так далее. Поэтому, если по каким-либо причинам образ не был сохранен, можно восстановить происходившие с системой события, пользуясь записями из файла журнала. Не сохранять образ иногда полезно, например, когда в ходе разработки были сделаны изменения, которые хотелось бы отбросить.

Поскольку в процессе работы размер файла `change.log` постоянно растет, его периодически необходимо сжимать — уменьшать до последней копии каждого нового или измененного метода. Сжатие журнала следует проводить периодически, по завершении разработки и отладки класса или большей его части. Для корректного проведения операции сжатия на диске должно быть достаточно места и для нового, и для старого файла `change.log`.

Чтобы сжать файл журнала системы, в любой текстовой панели надо выполнить выражение `Smalltalk compressChanges`.

Когда все новые классы и методы отлажены и постоянно используются, можно их интегрировать в систему, выполнив в любой текстовой панели выражение `Smalltalk compressSources`. В результате в файлах системе произойдут серьезные изменения.

- 1) Копия самой последней версии исходного текста каждого класса и метода, содержащаяся в файлах `change.log` или `sources.sml`, будет сохранена в файле `sources.sml` в сжатом формате. При этом изменятся соответствующие указатели в образе системы.
- 2) Будет создан новый пустой файл `change.log`.

Настоятельно рекомендуется делать резервные копии этих трех взаимосвязанных файлов перед каждым сжатием, чтобы можно было восстановить систему, если что-нибудь пойдет не так. А после успешного сжатия и проверки работоспособности системы следует заменить резервные копии вновь созданными файлами.

3.4.3. Выживание в случае краха системы

Система Смолток очень эластична. Но, к сожалению, могут случаться аварии, особенно, если производятся значительные изменения в самой системе. Характерная смолтоковская черта — автоматическая регистрация изменений — в большинстве случаев дает возможность восстановить работоспособность системы. Беды происходят не потому, что Смолток содержит в себе ошибки, а потому, что это — модифицируемая среда разработки программ. Смолток позволяет изменить почти все, даже если это наносит вред среде. С системой можно экспериментировать, чтобы научиться с ней работать во всех ситуациях, но прежде обязательно надо предпринять меры предосторожности.

- Создать резервные копии `v.exe`, `change.log` и `sources.sml`. Эти файлы используются вместе и должны копироваться вместе. Смешивание их старых и более новых версий может вести к невозможности обращения к исходному тексту некоторых методов.
- Перед тем как пытаться сделать то, что может разрушить систему, пользуясь пунктом `Save Image` из меню `File`, сохранить образ системы. Выполнение этого правила спасет всю проделанную работу. Если произошел сбой, всегда можно перезапустить Смолток, используя только что сохраненный образ. Если окажется, что использовать пункт `Save Image` невозможно, и произошел сбой, можно использовать пункт `Open...` из меню `File` и исследовать файл `change.log`.
- Если проводились эксперименты с системой Смолток и есть причины считать, что система повреждена, или есть изменения, от которых стоит избавиться, не сохраняйте образ. Помните, большинство сделанных изменений находятся в файле `change.log`, и то, что нужно, можно установить снова.

Если система Смолток разрушилась, не паникуйте. Перед тем как что-либо предпринять, сделайте копии рабочих файлов. Чтобы восстановить ранее сделанную работу, надо действовать по следующей схеме.

- Проверить, запускается ли система. Если не запускается, сохранить где-либо ее текущий файл `change.log`, взять последнюю резервную копию файлов `v.exe`, `change.log`, `sources.sml` и запустить систему.
- Теперь, когда есть работающая система, используя пункт **Open...** из меню **File**, надо просмотреть тот `change.log`, который использовался в момент краха системы. Он содержит все изменения и все выполнявшиеся выражения. Найти в нем запись о том, когда сохранялся тот образ, который сейчас выполняется. Ведь каждый раз, когда образ сохраняется, Смолток записывает в файл `change.log` комментарий, содержащий время и дату сохранения образа.
- Когда запись найдена, известно, что потерянная работа хранится от этой записи и до конца файла `change.log`. Чтобы восстановить работу, надо выбрать одну или несколько последовательных порций текста, а затем выбрать пункт **File It In** из меню **Smalltalk**. Каждая порция — выражение, которое будет выполнено. Если порция — определение метода, он будет перетранслирован и установлен в систему. Выбирая порции, будьте очень внимательны: одна из них содержит ошибку, которая разрушила смолтоковскую систему.

Когда восстановление завершено, не забудьте, используя пункт **Save Image** из меню **File**, сохранить новый образ.

3.4.4. Словарь системы

В Смолтоке есть класс **SystemDictionary**, определяющий в системе единственный экземпляр — словарь системы с именем **Smalltalk**. Словарь системы хранит все глобальные имена системы и определяет поведение, ориентированное на систему в целом. Именно к словарю системы мы обращались в выражениях, производящих сжатие файлов `change.log` и `sources.sml`.

Поскольку словарь **Smalltalk** хранит все глобальные имена, когда пользователь определяет новые классы, глобальные переменные или пулы, их имена автоматически добавляются в словарь системы. Для классов, представленных в словаре, ключ — имя класса, значение — сам класс. Для глобальных переменных ключ — имя переменной, значение — объект. Если переменная была определена, но не инициализирована, ее значение — `nil`. Для пулов ключ — имя пула, значение — словарь.

Перечислим некоторые из наиболее важных глобальных переменных системы и дадим краткое описание каждой из них.

- Clipboard** — единственный экземпляр класса **ClipboardManager**, который представляет в *Smalltalk Express* буфер обмена Windows, позволяющий передавать данные между окнами и приложениями.
- CurrentProcess** — процесс, который выполняется в настоящее время.
- Cursor** — экземпляр класса **CursorManager**, который используется для того, чтобы управлять курсорами приложения.
- Disk** — экземпляр класса **Directory**, представляющий тот каталог, из которого была запущена система.
- Display** — единственный экземпляр класса **Screen** (Экран), позволяющий работать непосредственно с доступным системе экраном монитора.
- KeyboardSemaphore** — экземпляр класса **Semaphore** (Семафор), который сообщает о появлении прерывания от мыши или клавиатуры.
- Notifier** — объект, играющий ключевую роль в обработке сообщений операционной среды Windows системой *Smalltalk Express*; единственный экземпляр класса **NotificationManager**.
- Processor** — процессор, единственный экземпляр класса **ProcessScheduler** (см. гл. 9).
- Sources** — массив, содержащий два файловых потока для доступа к исходным текстам, то есть к файлам `sources.sml` и `change.log`.
- SysFont** — шрифт по умолчанию, используемый для отображения текстовой информации.
- Terminal** — экземпляр класса **Pen** (Перо), связанный с объектом **Display**; используется для того, чтобы произвести звуковой сигнал, записать текст или вывести графику на экран.
- Transcript** — экземпляр **TextWindow**, представляющий окно, используемое для вывода сообщений системы; может использоваться для отображения необходимой информации (например, при отладке).
- Существует много полезных сообщений, которые можно посылать словари системы. Вот несколько из наиболее часто используемых:
- at: key put: anObject** — создает новую глобальную переменную с именем **key** и значением **anObject**; если глобальная переменная с таким именем существует, изменяет ее значение на **anObject**.
- implementorsOf: aSymbol** — возвращает окно **MethodsBrowser**, содержащее все классы, определяющие метод с именем **aSymbol**. Эквивалентно выбору функции **Implementors** из меню **Methods** окна просмотра иерархии классов.
- keys** — возвращает все ключи, определенные в словаре системы.

sendersOf: aSymbol — возвращает окно **MethodsBrowser**, содержащее все методы, и соответствующие им имена классов, которые посылают сообщение с именем **aSymbol**, что эквивалентно выбору функции **Senders** из меню **Methods** окна просмотра иерархии классов.

Словарь, содержащийся в общей переменной, может использоваться как пул. Создаются новые пулы при выполнении выражения типа

Smalltalk at: #MyPool put: Dictionary new.

После чего можно добавлять в этот словарь новые элементы (переменные пула), выполняя выражения вида **MyPool at: 'Age' put: 40.**

В базовой среде Смолток много пулов. Приведем описание тех, которые часто используются.

CharacterConstants — связывает имена со специальными символами: **Cr** (возврат каретки), **Lf** (перевод строки), **Esc** (отказ), **Ff** (перевод страницы) и т.д.; ключ словаря — строка, значения — связанный со строкой символ.

WinConstants — хранит константы операционной системы **Windows**.

ColorConstants — связывает имя цвета с его представлением в **Windows**.

CursorConstants — связывает имена курсоров с соответствующими экземплярами класса **CursorManager**.

FileConstants — хранит константы режимов открытия файлов.

ЧАСТЬ II

БИБЛИОТЕКА КЛАССОВ

Пред ним roast-beef окровавленный,
И трюфли, роскошь юных лет,
Французской кухни лучший цвет,
И Страсбурга пирог нетленный
Меж сыром лимбургским живым
И ананасом золотым. . .

А. Пушкин.
«Евгений Онегин», ст. XVI

ГЛАВА 4

ПРОТОКОЛ КЛАССА ОБЪЕКТ

Начнем изучение классов Смолтока с самого «главного» класса — класса `Object`. Он «главный» потому, что описывает поведение, общее для всех объектов системы. Это тот фундамент, опираясь на который, можно создавать новые виды объектов, добавлять новые сообщения и модифицировать уже существующие. Напомним, что класс `Object` — единственный класс иерархии, не имеющий суперкласса; он определяется следующим образом:

```
nil subclass: #Object
  instanceVariableNames: ' '
  classVariableNames: 'RecursionInError Dependents RecursiveSet'
  poolDictionaries: ' '
```

Определяемые в нем переменные класса используются всеми объектами системы. Переменные `RecursionInError` (логическая) и `RecursiveSet` (экземпляр класса `Set`) используются для работы с рекурсивными структурами данных. Переменная `Dependents` (экземпляр класса `IdentityDictionary`) представляет зависимости между объектами. Механизм зависимости, поддерживаемый этой переменной, мы рассмотрим в разд. 4.7.

4.1. Проверка функциональности объекта

Функциональность каждого объекта определяется его классом и проявляется в информации о том, является ли данный объект экземпляром указанного класса и может ли данный объект отвечать на указанное сообщение. Это отражает два типа отношений между экземплярами различных классов: в терминах иерархии «суперкласс/класс/подкласс» и в терминах совместно используемых объектами протоколов сообщений.

Класс *Object* Протокол экземпляра

`class` Возвращает объект, который является классом получателя.

`isKindOf: aClass` Возвращает `true`, если аргумент `aClass` — класс или суперкласс получателя. Иначе возвращает `false`.

`isMemberOf: aClass` Возвращает `true`, если класс получателя совпадает с `aClass`. Иначе возвращает `false`.

`respondsTo: aSymbol` Возвращает `true`, если получатель понимает сообщение с селектором (именем) `aSymbol`, и `false` в противном случае.

Вот несколько примеров таких сообщений и возвращаемых в ответ результатов:

```
5 class           → SmallInteger
$R class          → Character
'one' class       → String
#(2 4 6) class    → Array
Object class      → Object class
#(2 4 6) isKindOf: Collection → true
#(2 4 6) isMemberOf: Collection → false
#(2 4 6) isMemberOf: Array → true
Set new isKindOf: Collection → true
3.14 isKindOf: Collection → false
$R respondsTo: #isKindOf: → true
#(2 4 6) respondsTo: #at: → true
$R respondsTo: #at: → false
```

4.2. Равенство объектов

В классе *Object* определены два сообщения сравнения: проверка на идентичность и проверка на равенство. Идентичность `'=='` — это проверка того, являются ли два данных объекта одним и тем же объектом (то есть занимают ли они в памяти одно и то же место).

Идентичность двух одинаковых литеральных объектов, вообще говоря, зависит от реализации. Так, результат сравнения на идентичность двух литерально заданных символов, системных имен, коротких целых чисел во всех реализациях будет одинаков и равен `true`. А сравнение на идентичность двух литерально заданных строк или массивов в одних реализациях вернет `true` (*VisualAge for Smalltalk*), а в других — `false` (*Smalltalk Express*, *VisualWorks*).

Равенство `'='` — это проверка того, являются ли два объекта равными, то есть взаимозаменяемыми объектами, представляющими одну и ту же сущность предметной области. Решение о том, что значит «представлять взаимозаменяемые объекты», принимает получатель сообщения. Класс `Object` определяет равенство как идентичность. Каждый новый класс может переопределять метод с именем `#=`¹ для того, чтобы установить, как происходит проверка на равенство двух его экземпляров. Например, при проверке равенства двух массивов сначала сравниваются размеры массивов и, в случае их равенства, последовательно проверяются на равенство элементы массивов, соответствующие одинаковым индексам.

Кроме сообщений для проверки идентичности и равенства объектов есть сообщения для проверки неидентичности `'~='` и неравенства `'<='` объектов. Еще два сообщения — `isNil`, `notNil` — используются для проверки, является ли получатель сообщения объектом, равным `nil`, или нет. Приведем несколько примеров:

```
$R == $R           → true
#(2 4 6) class == Array → true
#(2 4 6) class ~~ Array → false
#(2 4 6) class ~~ Set  → true
#(2 4 6) = #(2 4 6)    → true
#(2 4 6) = #(6 4 2)    → false
3 = (6 / 2)           → true
'one' = 'two'         → false
'one' ~= 'two'        → true
nil isNil             → true
nil notNil            → false
true notNil           → true
3.14 isNil            → false
3.14 notNil           → true
```

¹ Если переопределяется метод `#=`, то *должен* быть переопределен и метод с именем `#hash`, возвращающий целое, так что для равных объектов возвращаются равные значения.

С понятием идентичности и равенства тесно связано сообщение `copy` (копировать), возвращающее новый объект, равный данному. Как понимать копирование, зависит от реализации. Например, в системе *Smalltalk Express* существуют два способа копирования каждого объекта. Различие состоит в том, надо ли копировать значения переменных объекта. Если не надо, то достаточно сообщения `copy` или `shallowCopy` — поверхностное копирование, которое возвращает копию получателя, совместно с ним использующую значения его переменных. Если же надо копировать структурные элементы объекта, то необходимо сообщение `deepCopy` — глубокое (или полное) копирование, которое возвращает объект, значения переменных которого являются копиями. Но в любой реализации и в любом случае копия объекта — это совершенно другой объект, поэтому замена значения переменной в копии не изменяет значения данной переменной в оригинале. Некоторое представление о различии между `shallowCopy` и `deepCopy` дает рис. 4.1.

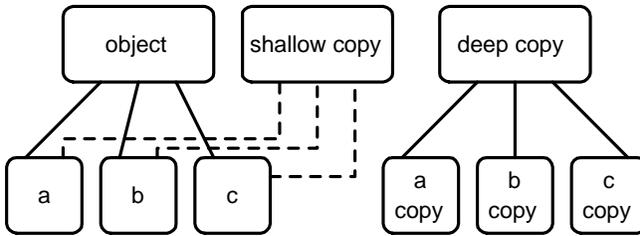


Рис. 4.1. Различие между поверхностной и полной копиями

В тех классах системы, в которых копирование должно дать специальную комбинацию совместно используемых и не используемых значений переменных, чаще всего переопределяется метод `copy`.

```

a := #'f' 'g' 'h')      → ('f' 'g' 'h')
b := a copy             → ('f' 'g' 'h')
a = b                   → true
a == b                  → false
(a at: 1) == (b at: 1) → true
c := Set new           → Set ()
d := c copy            → Set ()
e := c                 → Set ()
c = d                   → true
c == d                  → false
c = e                   → true
  
```

4.3. Индексированные переменные

Как мы знаем, в системе Смолток существует два типа переменных экземпляра: именованные переменные и индексированные переменные. Число именованных переменных у экземпляров одного и того же класса всегда одно и то же, число индексированных переменных у экземпляров одного и того же класса может быть разным. Поэтому классы, экземпляры которых имеют только именованные переменные, часто называют классами с *фиксированной структурой*, а классы, экземпляры которых имеют индексированные переменные, часто называют классами с *переменной структурой*. К именованной переменной можно обращаться по ее имени, а к индексированной — по индексу, который является положительным целым числом. Класс `Object` содержит сообщения, предназначенные для работы с индексированными переменными объекта.

Класс `Object`

Протокол экземпляра

`at: index` Возвращает значение индексированной переменной получателя сообщения с индексом, равным аргументу `index`; если получатель не имеет индексированных переменных или если аргумент больше, чем число индексированных переменных, сообщает об ошибке.

`at: index put: anObject` Сохраняет аргумент `anObject` как значение индексированной переменной получателя, индекс которой равен аргументу `index`; возвращает аргумент `anObject`. Если получатель не имеет индексированных переменных или если аргумент больше, чем число индексированных переменных, сообщает об ошибке.

`size` Возвращает число индексированных переменных получателя; эта величина совпадает с величиной наибольшего допустимого индекса. Если объект не имеет индексированных переменных, то возвращается `0`.

В этом же протоколе есть сообщения `basicAt:`, `basicSize`, `basicAt:put:`, каждое из которых аналогично соответствующему сообщению, перечисленному выше, однако эти сообщения не могут переопределяться ни в одном из классов смолтоковской системы.

Приведем пример. Класс `Array` — класс с переменной структурой, его разные экземпляры могут иметь разное число индексированных переменных. Пусть объект с именем `letters` — экземпляр класса `Array` вида `#$a $b $c $d $e $f $g $h`, тогда:

```

letters size           → 8
letters at: 3         → $c
letters at: 8 put: $H → $H
letters               → #($a $b $c $d $e $f $g $H)

```

4.4. Печать и сохранение объектов

Существуют разные способы описания объекта. Описание может давать информацию, необходимую только для того, чтобы узнать объект. Или описание может давать информацию, достаточную для полного восстановления объекта. Класс `Object` обеспечивает сообщения для обоих этих случаев. Ради эффективности и учета особенностей конкретных классов большинство классов системы эти сообщения переопределяют.

Класс `Object`

Протокол экземпляра

`printString` Возвращает строку, описывающую получателя.

`printOn: aStream` Добавляет в поток `aStream` (экземпляр класса `Stream`) строку, описывающую получателя.

`storeString` Возвращает строку, содержащую смолтоковское выражение, выполнение которого восстановит получателя.

`storeOn: aStream` Добавляет в поток `aStream` строку, содержащую смолтоковское выражение, выполнение которого восстановит получателя.

Например, экземпляр класса `Set`, состоящий из трех символов `$a`, `$b`, `$c`, будет печататься в виде `Set($a $b $c)`, в то же время храниться он будет в виде `(Set new add: $a; add: $b; add: $c)`.

Литеральные объекты, такие как символы, числа, строки, могут использовать одно и то же представление и для печати, и для хранения. Например, строка `'hello'` печатается и сохраняется как `'hello'`. Экземпляры класса `Symbol` часто печатаются без префикса `#`, а сохраняются с префиксом.

Чтобы лучше понять, как работают сообщения сохранения и печати, посмотрите их код и поэкспериментируйте с ними (но учтите, что ключевые сообщения из этого протокола переопределяются многими классами).

4.5. Обработка ошибочных ситуаций

Поскольку все изменения в системе происходят как реакция на посылку сообщений объектам, вполне понятно, что в системе может произойти ошибка следующего вида: сообщение посылается объекту, но метода, соответствующего сообщению, найти не удастся. В этом случае интерпретатор системы посылает первоначальному объекту сообщение `doesNotUnderstand: aMessage`. Аргумент `aMessage` — это непонятое получателем сообщение. Как мы уже знаем, сообщение об ошибке представляется пользователю через окно уведомлений `Walkback`, заголовок которого описывает,

в чем состоит проблема, а само окно представляет пользователю список контекстов выполнения в момент ошибки.

Это, по-видимому, одно из самых распространенных сообщений об ошибке, с которым вы будете сталкиваться при появлении окна уведомлений `Walkback`. Часто встречается ситуация, в которой правильное сообщение посылается не тому объекту; например, когда пытаются послать сообщение объекту `nil`. Обычно подобное происходит тогда, когда что-то забыли определить, инициализировать или присвоить значение переменной. Но такое может происходить и в том случае, если предыдущее сообщение вместо правильного объекта возвращает непредусмотренный, неправильный объект (в том числе и `nil`). Чтобы отследить подобную ситуацию, надо использовать стек вызовов, отображаемый окнами отладки.

Именно такая ошибка возникает, когда в создаваемом методе неправильно поставлен или вообще пропущен оператор возврата объекта, то есть пропущен символ `^`. Потери только одного оператора возврата достаточно, чтобы испортить длинную цепочку вызовов. И если вы получили сообщение об ошибке `doesNotUnderstand:`, проверьте, правильные ли объекты возвращаются используемыми методами.

Такая ошибка особенно неприятна в методах создания экземпляра. Правильный метод создания экземпляра должен выглядеть примерно так:

new

```
^ super new myInitialize
```

причем метод `myInitialize` (пере)определяется в создаваемом классе так, чтобы должным образом инициализировать переменные экземпляра. Если вы забыли вернуть значение, то есть написали нечто похожее на `super new myInitialize`, при вызове `new` будет возвращен не вновь созданный экземпляр класса, а сам класс, и, когда вы попытаетесь послать первое же сообщение вашему «новому» объекту, это немедленно вызовет исключительную ситуацию `doesNotUnderstand:` с сообщением вида:

```
MyClass class (Object)>>doesNotUnderstand:
```

Заметьте, что это сообщение отличается от сообщения, которое появляется, когда новый экземпляр действительно не понимает сообщения:

```
MyClass (Object)>>doesNotUnderstand:
```

Если вновь созданные объекты не понимают некоторых сообщений и вы не можете разобраться, почему подобное происходит, проверьте, не используете ли вы класс вместо экземпляра.

Неправильное сообщение, посланное правильному объекту, отследить сложнее. Вполне возможно, что, набирая выражение, вы просто ошиблись и исказили то сообщение, которое хотели напечатать. Такая же ошибка довольно часто возникает и из-за неправильной расстановки скобок, особенно

в сложных выражениях, и такую «опечатку» будет трудно отыскать. Еще один источник подобной ошибки — арифметические вычисления. Часто забывают, что все арифметические операции в Смолтоке имеют одинаковый приоритет.

Кроме того, многие методы системы используют механизм оповещения об ошибке в тех случаях, когда проверяются переданные данные, и оказывается, что дальнейшие вычисления с этими данными недопустимы. В таких случаях сам метод может определять представляемый программисту комментарий происшедшего. Сообщение, используемое в этой ситуации, — `error: aString` с аргументом, содержащим сообщение об ошибке. Такое сообщение создает в среде *Smalltalk Express* окно уведомлений Walkback, заголовок которого для описания ошибки использует аргумент `aString`. С помощью сообщения `error:` в классе *Object* реализовано сообщение `doesNotUnderstand:` и другие специальные сообщения об ошибке, описанные ниже:

Класс *Object*

Протокол экземпляра

`primitiveFailed` Сообщает пользователю, что метод, реализованный как системный примитив, при выполнении потерпел неудачу.

`implementedBySubclass` Сообщает пользователю, что метод, определенный в классе получателя, должен быть реализован в его подклассе. Такое сообщение хотя бы в одном методе класса указывает на абстрактный класс.

`invalidMessage` Сообщает пользователю, что для получателя посланное сообщение недопустимо (хотя оно является допустимым для экземпляров некоторых его суперклассов).

Для примера приведем реализацию одного из этих сообщений (все остальные аналогичны):

implementedBySubclass

```
^self error: 'my subclass should have implemented this message'.
```

Стандартные обработчики ошибок применимы только на этапе разработки программы, а в независимой прикладной программе они способны только испугать пользователя. Поэтому при разработке программы надо стараться в тексте методов предусмотреть реакцию на возможную ошибку (например, при вводе информации пользователь вместо числа ввел нечто другое) и, по возможности, переопределить методы обработки ошибок, так чтобы они выдавали менее пугающие сообщения и сохраняли где-либо сведения об ошибочной ситуации.

Другие реализации (*VisualWorks*, *VisualAge for Smalltalk*) используют механизм *исключений* для обработки сообщения об ошибках и для восстановления программы после возникновения ошибки. Таких средств в *Small-*

talk Express нет, но переопределение стандартных обработчиков ошибок и динамическая посылка сообщений (см. ниже) все же иногда позволяют вернуть программу к нормальному выполнению после возникшей ошибки.

4.6. Обработка сообщений

Поскольку вся работа в системе Смолток выполняется посредством посылки сообщений, в некоторых ситуациях бывает полезно не определять заранее посылаемое сообщение, а по ходу дела вычислять его. Например, из списка имен при вычислении может выбираться одно из них (присваиваемое переменной с именем `selector`) и посылаться объекту, называемому `receiver`. Для этого нельзя просто написать выражение `receiver selector`, поскольку такая запись означала бы посылку объекту, на который ссылается переменная `receiver`, унарного сообщения `selector`. Протокол класса `Object` включает методы для передачи объекту вычисляемых сообщений.

Класс `Object`

Протокол экземпляра

-
- `perform: aSymbol` Посылает получателю унарное сообщение с именем `aSymbol`; сообщает об ошибке, если получатель не понял это сообщение.
 - `perform: aSymbol with: anObject` Посылает получателю бинарное или ключевое сообщение с именем `aSymbol` и аргументом `anObject`; сообщает об ошибке, если число ожидаемых аргументов не равно 1.
 - `perform: aSymbol with: anObject1 with: anObject2` Посылает получателю ключевое сообщение с именем `aSymbol` и двумя аргументами `anObject1`, `anObject2`; сообщает об ошибке, если число ожидаемых аргументов не равно 2.
 - `perform: aSymbol with: anObject1 with: anObject2 with: anObject3` Посылает получателю ключевое сообщение с именем `aSymbol` и тремя аргументами `anObject1`, `anObject2` и `anObject3`; сообщает об ошибке, если число ожидаемых аргументов не равно 3.
 - `perform: aSymbol withArguments: anArray` Посылает получателю ключевое сообщение с именем `aSymbol` и аргументами, которые в должном порядке содержатся в массиве `anArray`; сообщает об ошибке, если число ожидаемых аргументов не равно размеру массива `anArray`.
-

Одной из задач, в которых эти методы могут использоваться, является расшифровка (декодирование) команд пользователя, например, при построении модели простого калькулятора, в которой операнды предшествуют определяемым пользователем операторам (см. [5, гл. 14]). Мы же ограничимся следующими простыми примерами, в которых пары выражений выполняют одинаковые вычисления:

1.7 squared	→ 2.89
1.7 perform: #squared	→ 2.89
1.7 raisedTo: 3	→ 4.913
1.7 perform: #raisedTo: with: 3	→ 4.913
Association key: 'Index'	
value: 344017	→ 'Index' ==> 344017
Association perform: #key:value:	
with: 'Index'	
with: 344017	→ 'Index' ==> 344017
#(a b c d e) copyReplaceFrom: 3	
to: 5	
with: #(C D E)	→ (a b C D E)
#(a b c d e)	
perform: #copyReplaceFrom:to:with:	
withArguments: #(3 5 (C D E))	→ (a b C D E)

4.7. Механизм зависимости

Вся информация в системе Смолток представляется объектами, которые взаимодействуют друг с другом в разных формах. Переменные, представляющие структурные единицы объектов, ссылаются на другие объекты; в этом смысле объекты устанавливают связи или зависимости друг с другом. Классы имеют определенные отношения со своими суперклассами и метаклассами, совместно используют внешние и внутренние описания структуры и поведения и тем самым зависят друга от друга. Эти две формы зависимости сконцентрированы в семантике языка Smalltalk.

Но существует еще один вид зависимости между объектами, который определяется и поддерживается классом *Object*. Его цель — координация действий между разными объектами. Конкретнее, цель состоит в том, чтобы дать возможность связать один объект, скажем *A*, с другим или многими другими объектами, скажем *B*, так, чтобы *B* мог получить некоторую информацию, если объект *A* каким-либо образом изменится. Получив информацию об изменении *A* и о характере происшедших изменений, *B* может предпринять некоторые действия по изменению собственного состояния.

Этот механизм реализуется через переменную класса с именем *Dependents*, определяемую классом *Object*. Поскольку любой класс системы есть, в конечном итоге, подкласс класса *Object*, все объекты системы имеют доступ к этой переменной, которая, напомним, представляет собой словарь —

экземпляр класса `IdentityDictionary`, ключами в котором являются объекты, а значениями — множество объектов, зависящих от объекта-ключа.

Механизм зависимости широко используется при поддержке объектов, имеющих графические представления. Каждый графический образ объекта зависит от объекта, который называется его моделью, в том смысле, что, если произошли изменения объекта-модели, образ объекта должен получить информацию об этих изменениях, а затем решить, воздействуют ли произошедшие изменения на отображаемую информацию и, если воздействуют, то произвести соответствующие действия. В частности, интерфейс пользователя любой системы Смолток широко использует этот механизм для отправки предупреждений о происходящих в объектах изменениях, часто добавляя собственные, более специальные механизмы зависимости.

Протокол класса `Object`, обеспечивающий механизм зависимости, невелик, и мы его приведем почти полностью.

Класс `Object`**Протокол экземпляра**

addDependent: anObject Добавляет аргумент `anObject` как один из объектов, зависящих от получателя сообщения.

dependents Возвращает набор (экземпляр класса `OrderedCollection`), содержащий все объекты, зависящие от получателя сообщения.

dependsOn: anObject Добавляет получателя сообщения в множество объектов, зависящих от `anObject`.

release Уничтожает зависимость объектов от получателя сообщения (это сообщение переопределяется подклассами).

changed Информировывает все зависящие от получателя объекты о его изменении, посылая каждому сообщение `update:`, с параметром равным получателю.

changed: aParameter Информировывает все зависящие от получателя объекты о его изменении, посылая каждому сообщение `update: aParameter`; характер произошедших изменений описывается аргументом `aParameter`. По умолчанию в качестве аргумента используется сам получатель сообщения.

update: aParameter Объект, от которого зависит получатель этого сообщения, изменился; поэтому получатель изменяет соответствующим образом свое состояние (по умолчанию ничего не происходит); информация о произошедших изменениях передается через аргумент `aParameter`.

broadcast: aSymbol Посылает аргумент `aSymbol` (как унарное сообщение) всем объектам, зависящим от получателя сообщения.

broadcast: aSymbol with: anObject Посылает аргумент `aSymbol` (как ключевое сообщение с аргументом `anObject`) всем объектам, зависящим от получателя сообщения.

Суть механизма зависимости в следующем. Всякий раз, когда объекту посылается сообщение `changed`, он автоматически посылает всем своим «иждивенцам» сообщение `update`. Другими словами, если вы изменяете объект и хотите, чтобы все зависимые от него объекты узнали об этом, измененному объекту должно быть послано сообщение `changed`. Иногда такая посылка происходит в наследуемом коде, а иногда это надо делать самим. Далее, каждый зависимый объект получает сообщение `update`. Если зависимые объекты что-то должны делать, когда объект, от которого они зависят, изменился, необходимо в классе зависимых объектов реализовать соответствующие «методы обновления». В классе `Object` есть реализация таких методов по умолчанию. Но эти методы ничего не делают и присутствуют только для того, чтобы гарантировать, что вы не получите сообщение об ошибке, если пошлете сообщение `change` объекту, который имеет такие зависимые от него объекты, для которых «метод модификации» не реализован.

Но почему не сделать так: пусть объект, который изменился, посылает сообщение непосредственно другому объекту, сообщая ему о произошедших изменениях? Ответ состоит в том, что во время разработки класса нельзя знать, каковы будут объекты, которые будут зависеть от создаваемого объекта. Последнее означает, что невозможно посылать явные сообщения, говоря всем зависимым объектам «Я изменился». Вместо этого просто объявляется, что объект изменился (посылкой себе сообщения `changed`), и запускается механизм зависимости, сообщая об изменении (с помощью посылки сообщения `update`) всем тем объектам, которые заинтересованы в получении такой информации. Таким образом, механизм зависимости может оказаться более динамичным, чем сложно программируемые явные сообщения между объектами.

Мы не станем здесь приводить конкретные примеры. Позже, когда мы больше узнаем об основных классах системы, мы еще вернемся к механизму зависимости.

ГЛАВА 5

УПРАВЛЯЮЩИЕ СТРУКТУРЫ

Управляющие структуры определяют порядок выполнения операций. В языке Смолток используется последовательное выполнение следующих друг за другом выражений языка. Но есть управляющие структуры, которые позволяют нарушить эту последовательность. Они чаще всего реализованы с помощью блоков.

В этой главе мы рассмотрим те библиотечные классы, которые используются при создании условных переходов, итераций и циклов. В конце главы мы поговорим о рекурсии в языке Смолток.

5.1. Условные выражения

С помощью блоков в Смолтоке реализованы две общие управляющие структуры: условный выбор и условное повторение. Условный выбор похож на оператор `if-then-else`, а условное повторение на операторы `while` и `until` в алголоподобных языках.

5.1.1. Условный выбор

Условный выбор обеспечивается сообщениями к логическим объектам `true` и `false`, и потому методы для них реализованы в классе `Boolean` и его подклассах `True` и `False`. Во всех следующих сообщениях блоки-аргументы должны быть блоками без переменных.

Класс `Boolean`

Протокол экземпляра

`ifTrue: trueBlock ifFalse: falseBlock` Если получатель сообщения — объект `true`, возвращает результат вычисления блока `trueBlock`; если получатель сообщения — объект `false`, возвращает результат вычисления блока `falseBlock`.

`ifFalse: falseBlock ifTrue: trueBlock` Выполняется аналогично предыдущему.

`ifTrue: trueBlock` Если получатель — `true`, возвращает результат вычисления аргумента `trueBlock`; если получатель — `false`, возвращает `nil` (это сообщение возвращает тот же результат, что и сообщение `ifTrue:ifFalse:`, когда его второй аргумент — пустой блок, поскольку значение пустого блока равно `nil`).

`ifFalse: falseBlock` Если получатель — `false`, возвращает результат вычисления аргумента `falseBlock`; если получатель — `true`, возвращает `nil` (это сообщение возвращает тот же результат, что и сообщение `ifFalse:ifTrue:`, когда его второй блок-аргумент — пустой блок).

Например, следующее выражение присваивает переменной `rem` значение 0 или 1 в зависимости от того, делится на 2 значение переменной `number` или нет (бинарное сообщение `\|` вычисляет остаток от деления целых чисел):

```
(number \| 2) = 0
  ifTrue: [rem := 0]
  ifFalse: [rem := 1]
```

Можно записать и по-другому:

```
rem := (number \| 2) = 0 ifTrue: [0] ifFalse: [1]
```

Рассмотрим еще один пример:

```
| max a b |
a := 5 squared.
b := 4 factorial.
a < b
  ifTrue: [max := b]
  ifFalse: [max := a].
^ max
```

После определения переменных `a` и `b` сравниваются их значения. Затем выражение `a < b` возвращает логический объект (в данном случае `false`), который становится получателем сообщения `ifTrue:ifFalse:.` В результате выполняется блок `[max := a]`.

5.1.2. Простое и условное повторение

Сообщение `timesRepeat: aBlock` (разПовторить: блок) из протокола класса `Integer` позволяет провести простое повторение вычислений. Посланное положительному целому числу, оно выполняет блок без параметров указанное число раз и возвращает получателя сообщения. Посланное отрицательному целому числу — ничего не делает. Выполним в рабочем окне следующую последовательность выражений:

```
| a |
a := 1.
10 timesRepeat: [a := a + a].
```

Последним значением переменной `a` будет число 1024, то есть 2^{10} .

В следующем примере воспользуемся сообщениями простого повторения и условного выбора для преобразования строки, при котором все гласные буквы превращаются в прописные, а все остальные — в строчные. Здесь используется тестирующее сообщение `isVowel` (этоГласная) из протокола класса `Character`, возвращающее `true`, только если его получатель —

гласная буква латинского алфавита, и преобразующие сообщения из этого же протокола `asUpperCase` (какПрописная) и `asLowerCase` (какСтрочная).

```
| string index c |
string := 'Now is the time'.
index := 1.
string size timesRepeat: [
    c := string at: index.
    string at: index put: (c isVowel ifTrue: [c asUpperCase]
                          ifFalse: [c asLowerCase]).

    index := index + 1].
^ string
```

Условное повторение некоторой последовательности операций обеспечивается сообщениями `whileTrue: aBlock` (покаИстина: блок) и `whileFalse: aBlock` (покаЛожь: блок) из протокола класса `Context`, в которых аргумент `aBlock` является блоком без параметров. В этих сообщениях блоку, который является получателем сообщения, посылается сообщение `value` и, если при этом возвращается объект `true` (соответственно, `false`), сообщение `value` посылается блоку-аргументу `aBlock`, после чего все начинается сначала: получателю посылается сообщение `value` и т.д. Когда блок-получатель в ответ на сообщение `value` возвратит `false` (соответственно, `true`), вычисления прекращаются, и возвращается `nil`. Например, условное повторение может быть использовано для инициализации всех элементов массива `list`:

```
| index list |
index := 1.
list := Array new: 5.
[index <= list size] whileTrue: [list at: index put: 0.
                                index := index + 1].
^ list
```

В качестве примера использования сообщения `whileFalse`: рассмотрим операцию копирования одного файла в другой. С системой *Smalltalk Express* поставляется учебник (каталог `tutorial`), главы которого (файлы `chapter.*`) стоит тщательно изучить. В качестве файла, который мы будем копировать, используем файл из этого учебника.

```
| input output |
input := File pathName: 'tutorial\chapter.2'.
output := File pathName: 'tutorial\copychap.2'.
[input atEnd] whileFalse: [output nextPut: input next].
input close.
output close
```

Дадим некоторые пояснения. Метод `pathName`: класса `File` открывает файл с указанным именем и создает файловый поток (экземпляр класса `FileStream`) для него (см. гл. 8). Сообщение `atEnd` возвращает `true` только тогда, когда в файловом потоке `input` больше нечего читать. Чтение информации из файлового потока `input` производит сообщение `next`, которое возвращает следующий элемент файлового потока. Запись информации в файловый поток `output` производит сообщение `nextPut`., которое записывает в файловый поток свой аргумент. Сообщение `close` закрывает файл.

Сообщения `whileTrue` и `whileFalse` из класса `Context` эквивалентные, соответственно, сообщениям `whileTrue: []` и `whileFalse: []`.

Очень часто, используя сообщения условного повторения, которые, как отмечалось, определены в классе `Context`, по ошибке их посылают логическому объекту (экземпляру класса `True` или `False`). Обратите внимание на следующий пример. Его первое выражение порождает ошибку; второе же выражение — правильное:

```
(myCount > 10) whileFalse: [myCount := myCount + 1].
(myCount > 10) whileFalse: [myCount := myCount + 1].
```

5.2. Итерации

5.2.1. Простой цикл

В большинстве алголоподобных процедурных языков программирования простой цикл строится с помощью выражения:

```
for i := nn step s to mm do операторы
```

которое в «переводе» на язык Смолток надо записать в виде выражения

```
nn to: mm by: s do: [:i | операторы].
```

Простые циклы на языке Смолток строятся с помощью следующих сообщений.

Класс `Number`

Протокол экземпляра

to: stop do: aBlock Выполняет одноаргументный блок `aBlock` для всех значений аргумента блока, заключенных между получателем и аргументом `stop` включительно, начиная с получателя, где каждое следующее число на единицу больше предыдущего.

to: stop by: step do: aBlock Выполняет одноаргументный блок `aBlock` для всех чисел между получателем и аргументом `stop`, начиная с получателя, где каждое следующее число равно предыдущему плюс величина шага `step`.

Вычислим в качестве примера сумму чисел $1/2$, $5/8$, $3/4$, $7/8$, 1 . Для этого достаточно выполнить выражения:

```
| sum |
sum := 0.
1/2 to: 1 by: 1/8 do: [:i | sum := sum + i ].
^ sum                               → 15/4
```

Чтобы увеличить на 1 каждый элемент массива с нечетным индексом, надо выполнить выражения:

```
| array |
array := #(1 2 3 4 5 6).
1 to: array size by: 2 do: [:i | array at: i put: ((array at: i) + 1)].
^ array                               → #(2 2 4 4 6 6)
```

5.2.2. Итерации по наборам

Протокол класса `Collection` содержит сообщения для более сложных итерационных вычислений, в которых в качестве аргумента используется блок с одним параметром.

Класс `Collection`

Протокол экземпляра

- do: aBlock** Выполняет блок `aBlock` по одному разу для каждого элемента из набора-получателя, подставляя каждый элемент набора в блок вместо его параметра.
- collect: aBlock** Выполняет блок `aBlock` по одному разу для каждого элемента из набора-получателя, собирает в новый набор, подобный получателю, результаты, возвращаемые блоком при каждом его выполнении, и возвращает новый набор.
- select: aBlock** Выполняет блок `aBlock` по одному разу для каждого элемента из набора-получателя и возвращает новый набор, подобный получателю, из тех его элементов, для которых блок вернул `true`.
- reject: aBlock** Выполняет блок `aBlock` по одному разу для каждого элемента из набора-получателя и возвращает новый набор, подобный получателю, из тех его элементов, для которых блок вернул `false`.
- detect: aBlock** Выполняет блок `aBlock` по одному разу для элементов из набора-получателя, возвращая первый элемент, при котором блок возвращает значение `true`; если такого элемента нет, сообщает об ошибке.

Поясним сказанное простыми примерами.

```
#(1 2 3 4 5 6 7) select: [:c | c odd]      → (1 3 5 7)
#(1 2 3 4 5 6 7) reject: [:c | c odd]   → (2 4 6)
#(1 2 3 4 5 6 7) collect: [:c | c \\ 2] → (1 0 1 0 1 0 1)
```

```
#(1 2 3 4 5 6 7) detect: [:i |
    i factorial > (i * i * i)] → 6
```

Пусть переменная `letters` содержит строку `'I live in Rostov-Don.'`. Надо определить, сколько раз в этой строке встречается буква `i` (как прописная, так и строчная). Поставленную задачу решает любое из следующих выражений:

```
(letters select: [:ch | ch asLowerCase == $i]) size → 3
```

```
(letters reject: [:ch | ch asLowerCase ~ $i]) size → 3
```

Того же можно добиться и с помощью сообщения `do:`

```
| count |
count := 0.
letters do: [:ch | ch asLowerCase == $i
    ifTrue: [count := count + 1]].
^ count → 3
```

В классе `Collection` есть два более сложных итерационных сообщения.

Класс <code>Collection</code>	Протокол экземпляра
-------------------------------	---------------------

detect: aBlock ifNone: exceptionBlock Выполняет блок `aBlock`, который является блоком с одним аргументом, по одному разу для каждого элемента из получателя, возвращая первый элемент, при котором блок возвращает значение `true`. Если такого элемента нет, выполняет блок `exceptionBlock`, который должен быть блоком без параметров, и возвращает полученный при его выполнении результат.

inject: thisValue into: binaryBlock Блок `binaryBlock` должен иметь два параметра; он выполняется по одному разу для каждого элемента из набора-получателя, который подставляется в блок вместо его второго параметра; вместо первого параметра подставляется результат предыдущего выполнения блока, а при первом выполнении — аргумент `thisValue`; возвращается результат последнего выполнения блока.

Как пример использования сообщения `inject:into:`, приведем другую реализацию задачи подсчета числа вхождений буквы `i` в строку `letters`. В этой реализации, в отличие от случая с использованием сообщения `do:`, нам не потребуется вводить дополнительные переменные:

```
letters
  inject: 0
  into: [:count :ch | count + (ch asLowerCase == $i
    ifTrue: [1] ifFalse: [0])] → 3
```

Все итерационные сообщения, кроме сообщения `do:`, которое реализовано в подклассах класса `Collection`, реализованы в самом классе `Collection`. Приведем в качестве примера определение метода для сообщения `collect:`

collect: aBlock

```
| newCollection |
newCollection := self species new.
self do: [ :each | newCollection add: (aBlock value: each)].
^ newCollection
```

Сообщение `add:`, используемое здесь, просто добавляет в набор новый элемент. Но в этом методе, а также в методах, связанных с сообщениями `select:` и `reject:`, используется сообщение `species`. Именно на это сообщение мы хотим обратить ваше внимание. Сообщение `species` реализовано в классе `Object` и возвращает класс объекта-получателя:

species

```
^ self class
```

Это сообщение используется для создания «подобных наборов» в операциях преобразования и копирования сложных структур данных. Если создание точного «подобия» исходного набора по каким-либо причинам невозможно или нежелательно, в таком классе этот метод нужно переопределить, создавая экземпляр другого класса.

Приведенные итерационные сообщения весьма эффективны, позволяют коротко и просто записывать сложные итерационные вычисления. Но такими сообщениями надо правильно пользоваться. Соблазнительно, например, пытаться менять содержимое набора во время выполнения над набором некоторых итерационных вычислений. Но так делать нельзя. Во время такой итерации некоторые элементы набора могут пропускаться или обрабатываться дважды. Вместо этого надо сделать копию набора и над ней производить итерацию, то есть написать нечто вроде:

```
aCollection copy do: [:each | ... aCollection remove: each. ...].
```

Если в большой программе в подобном выражении опустить сообщение `copy`, то поиск ответа на вопрос “Почему все работает не так?” может потребовать много сил и времени.

Но совершенно безопасно можно изменять характеристики объектов того набора, над которым производится итерация. Например, следующий код правилен и безопасен. Он проверяет размер каждой строки из набора `myCollection` и, если он равен 4, изменяет в этой строке первую букву:

```
| myCollection |
myCollection := #('one' 'two' 'three' 'four' 'five').
myCollection do: [:i | (i size = 4) ifTrue: [ i at: 1 put: $F]].
^ myCollection → #('one' 'two' 'three' 'Four' 'Five')
```

В последующих главах книги мы еще встретимся с управляющими структурами. А пока отвлечемся от общих управляющих структур и рассмотрим другой метод организации повторных вычислений — рекурсию.

5.3. Рекурсия

Как и во многих других языках программирования, итерационные методы можно переписать в виде рекурсивных методов. Метод называется рекурсивным, если в теле метода он вызывает сам себя.

Процедурная рекурсия может эффективно использоваться и в смолтовской среде. В этом можно убедиться, рассматривая в классе `Integer` реализацию метода экземпляра `factorial`, вычисляющего произведение чисел от 1 до получателя сообщения. Еще один пример — рекурсивный метод `fibonacci1`, который надо поместить в класс `Integer` и который вычисляет число Фибоначчи¹ с индексом, равным получателю.

factorial

```
"Возвращает факториал получателя."
self > 1 ifTrue: [^self * (self - 1) factorial].
self < 0 ifTrue: [^self error: 'negative factorial'].
^1
```

fibonacci1

```
"Возвращает n-е число Фибоначчи, где n — получатель
сообщения."
self <= 0 ifTrue: [^self error: 'negative fibonacci'].
^self < 3 ifTrue: [1]
  ifFalse: [(self - 1) fibonacci1 + (self - 2) fibonacci1]
```

Реализация этих методов точно следует математическим определениям, и написать такой код достаточно просто. Но эти два метода отличаются друг от друга. В методе `factorial` используется один рекурсивный вызов, а метод `fibonacci1` дважды вызывает сам себя, однако (что очень важно) эти вызовы не зависят друг от друга. Такого рода методы часто называют простыми рекурсивными методами с многократной рекурсией.

Если внимательно посмотреть на то, как в последнем методе происходят вычисления, то легко заметить его неэффективность: если $n > 3$, то для каждого числа m от 3 до $n - 1$, выражение `m fibonacci1` многократно вычисляется. Следующая реализация определяет вспомогательный рекурсивный метод с двумя дополнительными параметрами, в котором производится только один рекурсивный вызов, что позволяет избежать неэффективности предыдущей реализации:

¹ Напомним, что для положительного числа n , не меньшего 3, n -е число Фибоначчи вычисляется как сумма $(n - 1)$ -го и $(n - 2)$ -го чисел Фибоначчи, а первое и второе числа Фибоначчи равны 1.

fibHelp: f1 and: f2

```
"Частный вспомогательный метод для вычисления числа
Фибоначчи."
self < 2 ifTrue: [^ f2]
      ifFalse: [^(self - 1) fibHelp: f2 and: (f1 + f2)]
```

fibonacci2

```
"Возвращает n-е число Фибоначчи, где n — получатель
сообщения."
self <= 0 ifTrue: [^ self error: 'negative fibonacci'].
^ self fibHelp: 0 and: 1
```

Различие в эффективности этих двух методов вычисления чисел Фибоначчи огромно. Если попытаться выполнить выражение `45 fibonacci1`, то надо ждать результата довольно долго, а выражение `700 fibonacci2` выполнится мгновенно.

Чтобы увидеть другую проблему, связанную с применением рекурсии, воспользуемся идеей метода `fibonacci2` и напишем итеративный метод для вычисления числа Фибоначчи:

fibonacci3

```
| temp1 temp2 temp count |
count := self.
temp1 := 1. temp2 := 1.
self <= 0 ifTrue: [^ self error: 'negative fibonacci'].
[count < 3] whileFalse: [ temp := temp1. temp1 := temp2.
      temp2 := temp1 + temp.
      count := count - 1].
^ temp2
```

Скорости выполнения методов `fibonacci2` и `fibonacci3` сравнимы, но возможности применения метода `fibonacci2` ограничены размером используемого стека². Мы можем еще выполнить выражение `776 fibonacci2`, но уже не можем выполнить выражение `777 fibonacci2`. Поскольку в итерации стек не используется, его возможности ограничены только возможностью представления получаемого большого положительного целого числа. На выполнение выражения `5000 fibonacci3` Pentium-166 затратил менее двух секунд.

Эти примеры показывают достоинства и «подводные камни» рекурсии. Но есть задачи, решить которые можно только с помощью рекурсии. К та-

² В классических Смолток-системах, восходящих к реализации Smalltalk-80, контексты методов не использовали аппаратный стек, а были связаны в списки контекстов. Этот подход вместе с более глубокой оптимизацией байткода позволял широко использовать рекурсию в системе.

ким относится, например, задача о вычислении чисел Аккермана, которые определяются для целых неотрицательных m и n следующим образом:

$$A(m, n) = \begin{cases} n + 1, & \text{когда } m = 0, \\ A(m - 1, 1), & \text{когда } n = 0, \\ A(m - 1, A(m, n - 1)). & \end{cases}$$

Подводя итог, можно сказать, что процедурной рекурсией в среде Смолток пользоваться можно, только надо иметь в виду присущие ей ограничения. Но, говоря о рекурсии в смолтоковской среде, надо сказать, что здесь рекурсия — это один из основных принципов организации вычислений, основанный на рекурсивности многих структур данных в системе (например, иерархии классов). Когда мы сказали, что рекурсивный метод «вызывает сам себя», мы не стали уточнять, что это значит. Кроме привычного процедурного смысла, здесь возможен и другой, когда сообщение, посланное объекту, представляющему собой рекурсивную структуру данных, передается ему своим структурным единицам.

Из-за присущего объектно-ориентированным системам полиморфизма в них могут возникать (и возникают) сложные цепочки рекурсивных вызовов. Такой механизм вычислений называют *объектно-ориентированной рекурсией*. При этих вычислениях объект, получив сообщение, обрабатывает часть запроса (в рамках своей компетенции) и передает запрос другим объектам, которые сами вправе поступить точно так же или выполнить оставшуюся работу самостоятельно. Таким образом, вся обработка сообщения распределяется между множеством обработчиков.

В процедурной рекурсии функция каждый раз вызывается с разными, но однотипными аргументами. В конечном счете, аргумент принимает основное значение (условие выхода из рекурсии), при котором функция вычисляется очень просто, а затем рекурсия развертывается в обратном направлении до первоначального обращения. В объектно-ориентированной рекурсии метод полиморфно посылает сообщение другому получателю, который может быть другим экземпляром того же или другого класса. В конечном счете, вызывается метод, реализующий основной случай, который достаточно просто выполняет поставленную задачу, а затем рекурсия развертывается в обратном направлении до первого посланного сообщения.

Как правило, объектно-ориентированная рекурсия реализуется вдоль рекурсивной древовидной структуры и может использовать передачу сообщений по дереву как вверх, так и вниз. При этом уровень сложности таких структур заранее не известен. Ни один узел структуры не знает, что собой представляют другие ее узлы. Но зато каждый узел знает, каковы предыдущие и последующие узлы, так что, в конечном счете, все узлы можно перечислить один за другим.

Чтобы рассмотреть все особенности и схемы объектно-ориентированной рекурсии, нужна если не книга, то, по крайней мере, большая глава. Мы же ограничимся только двумя очень простыми, но весьма показательными примерами.

Рекурсия на дереве иерархии классов

Напомним, что иерархия классов смолтоковской системы представляет собой дерево. Начнем с примера объектно-ориентированной рекурсии, реализованной на иерархии классов: с реализации сообщения `initialize`. Предположим, что класс `D` — подкласс класса `C`, который является подклассом `B`, а последний — подкласс класса `A`, и все четыре класса реализуют сообщение `initialize`, производя разумную реализацию определяемых каждым классом переменных экземпляра. Тогда каждая реализация, кроме реализации в классе `A`, должна будет содержать выражение `super initialize`:

```
A class >> new                "Это метод класса в классе A."
  ^ self basicNew initialize

A >> initialize                "Это метод экземпляра в классе A."
  ...
  ^ self

B >> initialize                "Это метод экземпляра в классе B."
  super initialize.
  ...
  ^ self

C >> initialize                "Это метод экземпляра в классе C."
  super initialize.
  ...
  ^ self

D >> initialize                "Это метод экземпляра в классе D."
  super initialize.
  ...
  ^ self
```

Когда какой-либо объект, назовем его клиентом, создает экземпляр класса `D` и метод `new` посылает сообщение `initialize`, метод `initialize` класса `D` будет запускать аналогичный метод класса `C`, который, в свою очередь, потребует метода из класса `B`, который, в свою очередь, выполнит метод `initialize` из класса `A`. Таким образом, каждый класс берет на себя ответственность за инициализацию собственной части объекта.

Равенство и копирование

К общим сообщениям, реализованным в смолтоковских системах через объектно-ориентированную рекурсию, относятся сообщения `=` и `copy`. Мы уже знаем, что по умолчанию сообщение `=` в классе `Object` реализовано как идентичность (`==`). Однако классы, определяющие структурно сложные экземпляры, часто реализуют `=` намного более интересным способом. Давайте посмотрим, как бы мог реализовать сообщение `=` класс `Person` (Человек). Для примера, определим, что два экземпляра класса `Person` равны, если равны хранимые в переменной `name` значения:

```
Object subclass: #Person
```

```
instanceVariableNames: 'name address phoneNumber'
classVariableNames: ''
poolDictionaries:''
```

```
Person >> = aPerson
```

```
"Два экземпляра класса Person равны, если равны хранимые в
переменных name значения."
```

```
(aPerson isKindOfClass: Person) ifFalse: [^ false].
```

```
^ self name = aPerson name
```

```
Person >> hash
```

```
^ self name hash
```

```
Object subclass: #PersonName
```

```
instanceVariableNames: 'firstName lastName'
classVariableNames: ''
poolDictionaries:''
```

```
PersonName >> = aPersonName
```

```
"Два экземпляра класса PersonName равны, если равны имена
и фамилии, представленные строками."
```

```
(aPersonName isKindOfClass: PersonName) ifFalse: [^ false].
```

```
^ (self firstName = aPerson firstName)
```

```
and: [self lastName = aPerson lastName]
```

```
PersonName >> hash
```

```
^ self firstName hash + self lastName hash
```

Напомним, что две строки равны, если последовательно равны составляющие их символы. Приведенный текст показывает, что два экземпляра класса `Person` равны, если равны два экземпляра класса `PersonName`, хранящиеся в их переменных с именем `name`. Два экземпляра класса `Person-`

Name равны, если равны имена (`firstName`) и фамилии (`lastName`); строки равны, когда последовательно совпадают все их символы. Такая передача сообщения = (от `Person` к `PersonName`, затем к `String` и, наконец, к `Character`) — рекурсия. Каждый объект берет на себя ответственность за часть вычислений, а затем делегирует остальную часть ответственности своим непосредственным компонентам.

Алгоритм объектно-ориентированной рекурсии, применяемый объектом для создания полной (то есть полностью независимой) копии самого себя, очевиден. Он очень похож на предыдущий и состоит в следующем: сделать копию самого объекта и всех его составляющих, которые, в свою очередь, должны сделать копии всех своих составляющих, и так далее. Найдите в своей системе сообщения `copy` (`deepCopy`) и разберитесь с их реализацией самостоятельно.

5.4. Задания для самостоятельной работы

1. Используя идею введения дополнительного параметра, примененную в методе `fibHelp: f1 and: f2`, напишите более эффективный вариант рекурсивного метода `factorial`.
2. Напишите итеративный вариант метода `factorial`. Протестируйте и сравните его возможности с рекурсивными версиями.
3. Для вычисления чисел Аккермана напишите в классе `Integer` рекурсивный метод экземпляра `accerman: anInteger`. Поэкспериментируйте с ним, вычисляя выражения вида `n accerman: m` (даже на очень мощном компьютере не стоит выбирать значения для m и n вне первого десятка).
4. Напишите в классе `Integer` рекурсивный метод экземпляра `factors`, который вычисляет и возвращает в виде упорядоченного набора все простые делители получателя. Для реализации такого метода, возможно, придется построить частный вспомогательный метод. Если на этом этапе написать такой метод не удастся, вернитесь к этой задаче после того, как прочитаете главу о наборах. Напишите итеративный метод для решения этой же задачи. Сравните возможности методов.
5. Напишите в классе `Collection` рекурсивный вариант метода `do: aBlock`.

ГЛАВА 6

ВЕЛИЧИНЫ

6.1. Класс Magnitude

Протокол класса **Magnitude** позволяет работать с объектами, представляющими количества: даты, времена, числовые величины, символы. Общие свойства всех этих объектов проявляются в том, что их можно сравнивать и упорядочивать. Иерархия класса **Magnitude** в *Smalltalk Express* имеет следующий вид:

Magnitude	Величина
Association	АссоциативнаяПара
Character	Символ
Date	Дата
Number	Число
Float	СПлавающейТочкой
Fraction	РациональнаяДробь
Integer	Целое
LargeNegativeInteger	БольшоеПоложительноеЦелое
LargePositiveInteger	БольшоеОтрицательноеЦелое
SmallInteger	МалоеЦелое
Time	Время

Класс **Magnitude** — абстрактный класс, который обеспечивает протокол для сравнения и упорядочения, наследуемый всеми его подклассами, но предполагает, что его подклассы самостоятельно реализуют методы для отношений порядка и сравнений: $=$, \leq , \geq , $<$, $>$, \sim . Опираясь на эти методы, класс **Magnitude** определяет общие методы проверки на принадлежность сегменту и вычисления максимума-минимума:

46 min: 33	→ 33
46 max: 33	→ 46
2 between: 0 and: 1	→ false

6.2. Класс Character

Класс **Character** включен как подкласс в класс **Magnitude** потому, что его экземпляры образуют линейно упорядоченное множество, порядок в котором обеспечивается языковыми настройками операционной системы.

Для любых двух символов из этого множества всегда можно сказать, какой из символов предшествует (<) или следует (>) за другим. Ссылки на символы возможны двумя способами: литерально, непосредственно на сами символы или на целые числа, соответствующие их числовым кодам. Для преобразования числа в символ есть два сообщения. Сообщение `asCharacter` должно посылаться целому числу, а сообщение `value:` с аргументом, равным целому числу, должно посылаться классу `Character`. Следующая таблица относится к среде *Smalltalk Express*, где каждый символ связан со своим кодом — числом от 0 до 255:

Символ	Литера	Эквивалентное выражение
A	\$A	65 asCharacter
C	\$C	Character value: 67
<i>space</i>	\$	32 asCharacter
<i>line feed</i>		10 asCharacter
<i>tab</i>		Character value: 9

Подобно всем подклассам класса `Magnitude`, класс `Character` реализует методы для операций сравнения, устанавливая так называемый алфавитный порядок (с учетом настроек Windows). Проверка принадлежности сегменту и методы для вычисления минимума-максимума наследуются из класса `Magnitude`:

```
$a = $A           → false
$A < $B           → true
$E max: $A        → $E
$x between: $a and: $t → false
$d between: $a and: $t → true
```

Класс `Character` определяет много самых разнообразных сообщений. Приведем их небольшую часть в виде примеров с комментариями:

```
$a isUpperCase   → false   "Является ли прописной?"
$a isLowerCase   → true    "Является ли строчной?"
$a asUpperCase   → $A      "Представить как прописную."
$? asLowerCase   → $?     "Представить как строчную."
$e isVowel       → true    "Это гласная латинская буква?"
$+ isLetter      → false   "Это буква?"
$9 isDigit       → true    "Это цифра?"
$A asInteger     → 65     "Код символа."
```

Как пример применения сообщений, сравнивающих символы, создадим метод экземпляра для класса `String` с именем `min:`, который будет определять из двух строк (получателя и аргумента сообщения) ту, которая стоит

раньше в соответствии с алфавитным порядком. Строка — экземпляр класса `String`, представляющий индексированный набор из символов, который отвечает на сообщение `at: i`, возвращая символ с индексом `i`. Таким образом, выражение `'abcde' at: 2` возвращает `$b`.

min: aString

1 to self size do:

[:index |

(index > aString size) ifTrue: [^ aString].

(self at: index) > (aString at: index) ifTrue: [^ aString].

(self at: index) < (aString at: index) ifTrue: [^ self]].

^ self

Алгоритм этого метода представляет собой итерацию по символам получателя сообщения. Итерация прекращается тогда, когда выполняется одно из следующих трех условий:

- в аргументе `aString` больше нет символов для сравнения с очередным символом получателя (`index > aString size`);
- следующий символ в получателе расположен после соответствующего символа в аргументе `aString` (`(self at: index) > (aString at: index)`);
- следующий символ получателя расположен до соответствующего символа аргумента `aString` (`(self at: index) < (aString at: index)`).

Приведем несколько примеров работы созданного метода:

```
'love' min: 'live'    → 'live'
'apple' min: 'pear'   → 'apple'
'butt' min: 'butter'  → 'butt'
'event' min: 'even'  → 'even'
```

Арифметические операции символами не поддерживаются.

6.3. Дата и время

Экземпляр класса `Date` представляет конкретную дату, например, 1 января 1999, 9 мая 1945 или 21 июля 2048. Класс `Date` «знает» и «умеет» использовать следующую информацию:

- в неделе 7 дней, каждый день имеет имя и индекс 1, 2, ..., 7 (индекс 1 соответствует понедельнику);
- в году 12 месяцев, каждый месяц имеет имя и индекс 1, ..., 12;
- в месяце может быть 28, 29, 30 или 31 день;
- год может быть високосным.

Экземпляр класса `Time` представляет конкретное время дня. Считается, что день начинается в полночь.

Даты и время создаются посылкой сообщений соответствующим классам. Вот некоторые примеры:

```
Time now                → 16:59:53
Date today              → 20 January 1999
Date newDay: 14 month: #Aug year: 1998 → 14 August 1998
Date newDay: 202 year: 1999            → 21 July 1999
Time hours: 12 minutes: 0 seconds: 0    → 12:00:00
Time fromSeconds: 27000                 → 07:30:00
```

Еще один способ создать дату — послать экземпляру класса `String`, имеющему специальный вид, сообщение `asDate`:

```
'14 August 1998' asDate → 14 August 1998
```

Следующие выражения создают два экземпляра класса `Date`, три экземпляра класса `Time` и помещают их в соответствующие глобальные переменные системного словаря `Smalltalk`:

```
Smalltalk at: #BirthdayMyDaughter put: '29 Nov 1980' asDate.
Smalltalk at: #BirthdayMySon put: '28 April 1973' asDate.
Smalltalk at: #LunchTime
  put: (Time hours: 12 minutes: 0 seconds: 0).
Smalltalk at: #DinnerTime
  put: (Time hours: 18 minutes: 45 seconds: 0).
Smalltalk at: #BreakfastTime
  put: (Time hours: 7 minutes: 30 seconds: 0).
```

Теперь можно работать с этими объектами, посылая им сообщения:

```
BirthdayMyDaughter year                → 1980
BirthdayMySon dayName                  → Saturday
BirthdayMySon > Date today             → false
BirthdayMySon min: Date today          → Apr 28, 1973
BirthdayMyDaughter daysInYear         → 365
BirthdayMySon monthName                → April
BirthdayMyDaughter subtractDate: BirthdayMySon → 1311
LunchTime between: BreakfastTime
  and: DinnerTime                      → true
LunchTime min: DinnerTime              → 12:00:00
DinnerTime hours                       → 18
BreakfastTime minutes                  → 30
LunchTime < BreakfastTime             → false
LunchTime subtractTime: BreakfastTime  → 04:30:00
```

Мы уже упоминали сообщение `millisecondsToRun: aBlock`, которое возвращает число миллисекунд, необходимых для выполнения на данной машине блока `aBlock`. Например,

```
Time millisecondsToRun: [700 factorial] → 60
```

Ввиду того, что при выполнении любого сообщения существуют некоторые накладные расходы, возвращаемый результат не является точным временем выполнения блока.

6.4. Числа: большие, малые и всякие

Как и все остальное, числа — тоже объекты. Каждый вид числовых величин представляется отдельным классом. Числовые классы реализованы таким образом, что все числа ведут себя так, как если бы они принадлежали наиболее общему числовому типу. Поэтому все эти классы являются подклассами абстрактного класса `Number`, который определяет общее поведение, поддерживает смешанную арифметику для различных видов чисел и обеспечивает много полезных функций различного назначения. Еще раз подчеркнем, что в системе Смолток все бинарные сообщения имеют одинаковый приоритет, что не соответствует правилам, принятым в математике. Поэтому для достижения принятого в математике порядка выполнения арифметических операций необходимо использовать круглые скобки.

Смолток поддерживает три вида чисел¹, реализованных в соответствующих подклассах: целые числа (абстрактный класс `Integer` и его подклассы), рациональные числа (класс `Fraction`) — отношение двух целых чисел `numerator` (числитель) и `denominator` (знаменатель), числа с плавающей точкой (класс `Float`), которые позволяют аппроксимировать вещественные числа. Диапазон возможных чисел с плавающей точкой зависит от реализации. *Smalltalk Express* дает возможность представлять числа в диапазоне от $\pm 4.19 \times 10^{-307}$ до $\pm 1.67 \times 10^{308}$. Целые числа и числа с плавающей точкой задаются литерально. Дроби создаются, когда сообщение (`/`) посылается целому числу с аргументом, равным целому числу; при этом дробь автоматически приводится к несократимой дроби (с взаимно простыми числителем и знаменателем). Если результат имеет знаменатель, равный 1, дробь представляется целым числом.

¹ Современные реализации языка Смолток включают еще и четвертый тип чисел: числа с фиксированной запятой, позволяющие реализовывать корректные финансовые вычисления. Это дополнение включено в стандарт языка и позволяет Смолтоку успешно конкурировать с Коболом за рынок финансовых и деловых приложений.

Целые числа в *Smalltalk Express* представляются экземплярами классов `LargeNegativeInteger`, `LargePositiveInteger` и `SmallInteger`. В других реализациях набор подклассов может быть несколько иным. Экземпляры класса `SmallInteger` располагаются в диапазоне, представимом одним машинным словом в данной реализации (для *Smalltalk Express* от -32767 до 32767). Они эффективны как по скорости вычисления, так и по объему используемой памяти. Классы больших целых чисел могут представлять числа неограниченного размера (реально ограниченного только памятью компьютера). Преобразования между классами целых чисел происходят автоматически. Выполнение типичных выражений — самый лучший способ понять правила таких преобразований:

$1 + 2$	\rightarrow	3	
$5.1 - 3$	\rightarrow	2.1	
$2 * -4.0$	\rightarrow	-8.0	
$2/4$	\rightarrow	$1/2$	Приведенная рациональная дробь
$1/2 + 1$	\rightarrow	$3/2$	Возвращает рациональную дробь
$1/2 + 1.0$	\rightarrow	1.5	Возвращает число с плавающей точкой
$4/2$	\rightarrow	2	Выполняет целочисленное деление
$2 + 3 * 4$	\rightarrow	20	Вычисления проводятся слева направо

Класс `Number` реализует много математических функций, наследуемых его подклассами, например: `exp`, `sin`, `cos`, `arcSin`, `tan`, `ln`, `sqrt`, `floor`. Не забывайте, что имена этих математических функций — селекторы сообщений, которые посылаются числам и записываются, согласно синтаксису Смолтока, после числа. Поэтому запись выглядит несколько непривычно:

$2.3 \sin$	\rightarrow	$7.45705212e-1$	$\sin 2.3$
$3.44 \cos$	\rightarrow	$-9.5580594e-1$	$\cos 2.3$
0.25arcTan	\rightarrow	$2.4497866e-1$	$\arctan 0.25$
$2.21 \ln$	\rightarrow	$7.92992516e-1$	$\ln 2.21$
$6 \log: 2$	\rightarrow	2.5849625	$\log_2 6$
4sqrt	\rightarrow	2.0	$\sqrt{4} = 2$
2exp	\rightarrow	7.3890561	e^2
$1.72 \text{raisedTo: } 2.3$	\rightarrow	3.48109141	$1.72^{2.3}$
$3/7 \text{raisedToInteger: } 3$	\rightarrow	$27/343$	$(3/7)^3$
$3/11 \text{reciprocal}$	\rightarrow	$11/3$	обратная величина
10negated	\rightarrow	-10	
$30 \text{degreesToRadians}$	\rightarrow	$5.2359877e-1$	градусы в радианы
-2.3abs	\rightarrow	2.3	$ -2.3 $
5.1ceiling	\rightarrow	6	$\lceil 5.1 \rceil$
-5.1ceiling	\rightarrow	-5	$\lceil -5.1 \rceil$
5.1floor	\rightarrow	5	$\lfloor 5.1 \rfloor$

-5.1 floor	→ -6	$\lfloor -5.1 \rfloor$
5.1 truncated	→ 5	$\lceil 5.1 \rceil$
-5.1 truncated	→ -5	$\lceil -5.1 \rceil$
5.1 rounded	→ 5	
5.1 truncateTo: 2.3	→ 4.6	усечение, кратное параметру
5.9 roundTo: 2.3	→ 6.9	

Арифметические сообщения, возвращающие целые частное или остаток, определены двумя способами. В одних сообщениях (`//`, `\|`) округление происходит в сторону $-\infty$, в других (`quo:`, `rem:`) — в сторону нуля. Для положительных чисел результаты одни и те же, так как 0 и $-\infty$ располагаются в одном направлении; для отрицательных — результаты будут разными. Результаты посылки сообщений `quo:`, `rem:` и `//` положительны, когда аргумент и получатель одного знака, и отрицательны, когда они разных знаков; сообщение `\|` всегда возвращает положительное число.

4 // 3	→ 1	частное — целое число,
-4 // 3	→ -2	усечение в сторону $-\infty$
4 \ 3	→ 1	целый остаток,
-4 \ 3	→ 2	усечение в сторону $-\infty$
-4 quo: 3	→ -1	частное, усечение к нулю
-4 rem: 3	→ -1	остаток, усечение к нулю

Класс `Number` реализует много методов тестирования.

2 odd	→ false
4 even	→ true
10 negative	→ false
0 positive	→ true
-0.1 positive	→ false
0 strictlyPositive	→ false
0.1 strictlyPositive	→ true
-4.32e-2 sign	→ -1

Кроме стандартных операций класс `Integer` содержит протокол сообщений для работы с числами как с последовательностями битов. Среди них отметим сообщения `bitAnd:`, `bitOr:`, `bitXor:`, `bitShift:`, `bitAt:`. Предлагаем читателю самому разобраться в том, что и как делают эти сообщения.

ГЛАВА 7

НАБОРЫ НА ЛЮБОЙ ВКУС

7.1. Протокол класса Collection

Набор — группа совместно используемых объектов, которые называются элементами набора. Например, массив `#('word' 3 5 $G (1 2))` — это набор, состоящий из пяти элементов: первый — строка `'word'`, второй и третий — числа 3 и 5, четвертый — символ `$G`, а пятый — массив из двух чисел. Строка `'word'` — тоже набор, состоящий из четырех символов. Наборы нужны для представления основных структур данных и служат хранилищами объектов. Приведем иерархию класса `Collection` в системе *Smalltalk Express*. И в этой, и в других реализациях, кроме указанных, в иерархии есть и другие, более специальные классы.

Collection	Набор
Bag	ПростойНабор
IndexedCollection	ИндексированныйНабор
FixedSizeCollection	НаборФиксированнойДлины
Array	Массив
ByteArray	МассивБайт
Interval	Интервал
String	Строка
Symbol	Имя
OrderedCollection	УпорядоченныйНабор
SortedCollection	СортируемыйНабор
Set	Множество
Dictionary	Словарь
IdentityDictionary	СловарьИдентичности
SystemDictionary	СловарьСистемы

Рис. 7.1. Иерархия наборов

Наборы, определяемые разными подклассами этой иерархии, отличаются по своим свойствам. Одно из отличий состоит в том, определен или нет порядок среди элементов набора. Другое отличие состоит в том, имеет набор фиксированный размер или размер набора при необходимости может меняться. Существуют наборы, доступ к элементам которых происходит по известным вне набора именам (так называемым *ключам*). Вид ключей определяет еще одно отличие между классами наборов. Элементы одних

наборов доступны по числовым натуральным индексам, которые определяют порядок элементов в наборе. В других используется явно связанные с элементами набора внешние объекты, служащие ключами поиска. Еще одно отличие состоит в ограничении того, какие объекты могут быть элементами набора.

Представим эти характеристики классов в виде табл. 7.1, в которой использованы такие сокращения: **⊠** — абстрактный класс, **да!** — набор упорядочен в соответствии с внутренними характеристиками, **о.п.** — свойство определяется подклассом.

Таблица 7.1. Характеристики наборов

Класс	Порядок	Размер	Дубли	Ключ	Элементы
Bag	нет	любой	да	нет	любой
IndexedCollection ⊠	да	о.п.	о.п.	целый	о.п.
FixedSizeCollection ⊠	да	фикс.	о.п.	целый	о.п.
Array	да	фикс.	да	целый	любой
ByteArray	да	фикс.	да	целый	0–255
Interval	да!	фикс.	нет	целый	Number
String	да	фикс.	да	целый	Character
Symbol	да	фикс.	да	целый	Character
OrderedCollection	да	перем.	да	целый	любой
SortedCollection	да!	перем.	да	целый	любой
Set	нет	перем.	нет	нет	любой
Dictionary	нет	перем.	нет	любой	любой
IdentityDictionary	нет	перем.	нет	любой	любой
SystemDictionary	нет	перем.	нет	Symbol	любой

Элементы в экземплярах классов **Bag** и **Set** не сортируются и не связываются ни с какими ключами. Дубликаты элементов возможны в экземплярах класса **Bag** и невозможны в экземплярах класса **Set**.

Все упорядоченные наборы — это экземпляры подклассов класса **IndexedCollection**. В них элементы доступны через целые положительные ключи-индексы. Подклассы класса **IndexedCollection** поддерживают разные способы упорядочивания своих элементов. Экземпляры подклассов класса **FixedSizeCollection** после создания не могут менять свой размер. Элементами экземпляров классов **Array** могут быть любые объекты, элементами экземпляров класса **ByteArray** — целые в диапазоне от 0 до 255, элементами экземпляров класса **String** — экземпляры класса **Character**.

Порядок элементов является внешним для экземпляров классов **OrderedCollection**, **FixedSizeCollection**. Порядок элементов в экземплярах **Orde-**

`redCollection` определяется последовательностью операций по их добавлению и удалению. Элементами экземпляров класса `OrderedCollection` могут быть любые объекты. Порядок среди элементов экземпляров классов `Interval` и `SortedCollection` определяется внутренними свойствами самих элементов и никакими средствами не может быть изменен извне. Элементы экземпляра класса `Interval` составляют арифметическую прогрессию, которая однозначно определяется во время инициализации экземпляра. Для класса `SortedCollection` порядок элементов определяется заданным для данного экземпляра блоком (функцией) сортировки.

Экземпляр класса `Dictionary` — несортированный набор с внешними ключами, которые могут быть любыми объектами системы, а сравнение ключей происходит с помощью равенства `=`. В его подклассах `IdentityDictionary`, `SystemDictionary` внешние ключи сравниваются с помощью операции тождества `==`, что происходит значительно быстрее.

Наша цель — коротко описать протокол самого класса `Collection` и все особенности протоколов сообщений его подклассов, а также привести простые поясняющие примеры.

Один из общих протоколов класса `Collection` нам уже знаком: это протокол, обеспечивающий итерации над наборами (его часто называют протоколом перечисления). Кроме того, класс `Collection` обеспечивает протоколы для создания новых наборов, преобразования наборов, добавления и удаления элементов, определения текущего состояния набора.

7.1.1. Создание новых наборов

В главе, посвященной синтаксису языка, приводились примеры литерального создания некоторых наборов (экземпляров классов `Array`, `String`, `Symbol`). Для создания новых наборов можно также использовать сообщения `new` и `new:`. Но протокол класса `Collection` содержит специальные сообщения класса, обеспечивающие создание нового набора с одним, двумя, тремя или четырьмя элементами (в зависимости от числа ключевых слов `with:` в ключевом сообщении). Например:

`Set with: $a with: $b with: $c.` \rightarrow `Set($a $b $c)`

Заметим, что основная причина, по которой обеспечиваются только эти четыре сообщения создания экземпляров, а не больше и не меньше, состоит в том, что именно такое их количество оказалось достаточным для решения всех тех задач, ради которых эти сообщения вставлялись разработчиками в систему. При желании их число можно увеличить. Например, создадим метод порождения нового экземпляра с элементами из массива-аргумента:

withElements: anArray

```
"Возвратить набор со всеми элементами из массива anArray."
| answer |
answer := self new.
anArray do: [:element | answer add: element].
^ answer
```

Теперь множество из предыдущего примера можно создать и так:

```
Set withElements: #($a $b $c).      → Set($a $b $c)
```

Но почему такой метод будет корректно работать для подклассов класса `Collection`, ведь они такие разные? Все дело в том, что существующие различия между подклассами проявляются только при добавлении, удалении и перечислении элементов набора. Класс `Collection` реализует сообщения `add: anObject` (добавить:), `remove: anObject ifAbsent: aBlock` (удалить:еслиОтсутствует:), `do: aBlock` (выполнить:) как `^self implementedBySubclass`. То есть эти сообщения должны реализовываться подклассами с учетом их особенностей. Остальной протокол класса `Collection` реализован через эти сообщения. Те сообщения, которые в процессе специализации становятся недопустимыми, реализуются в подклассах выражением `^self invalidMessage`. Например, сообщение `add:` в классе `FixedSizeCollection` становится невозможным, поскольку это набор фиксированной длины, и просто добавить в него новый элемент нельзя: новый элемент может только заменить один из существующих. Далее мы приведем некоторые методы из класса `Collection`, опирающиеся на три перечисленных выше основных сообщения, а некоторые реализации последних будем рассматривать в подклассах в качестве примеров.

7.1.2. Преобразование наборов

Так как разные классы наборов имеют разные характеристики, полезно иметь возможность преобразовывать набор одного вида в набор другого вида. Допустимость преобразования между различными видами наборов зависит от способов представления в системе экземпляра каждого из подклассов. Класс `Collection` определяет пять сообщений для преобразования наборов:

Класс <code>Collection</code>	Протокол экземпляра
-------------------------------	---------------------

`asArray` Возвращает экземпляр класса `Array` с элементами из получателя.

`asBag` Возвращает экземпляр класса `Bag` с элементами из получателя.

`asSet` Возвращает экземпляр класса `Set` с теми же элементами, что и у получателя, при этом дубликаты уничтожаются.

`asOrderedCollection` Возвращает экземпляр класса `OrderedCollection` с теми же элементами, что и у получателя; порядок элементов в новом наборе зависит от используемого алгоритма преобразования.

`asSortedCollection: aBlock` Аналогично предыдущему, но с элементами, отсортированными согласно блоку сортировки `aBlock`.

`asSortedCollection` Возвращает экземпляр класса `SortedCollection` с элементами из получателя, отсортированными по возрастанию, т.е. в соответствии с блоком сортировки `[:a :b | a <= b]`.

Реализация преобразующих сообщений практически одинакова, приведем только одно из них:

asArray

```
| answer index |
answer := Array new: self size.
index := 1.
self do: [:element | answer at: index put: element.
          index := index + 1].
^ answer
```

В большинстве случаев новый экземпляр имеет тот же самый размер, что и первоначальный набор. В случае преобразования набора в экземпляр класса `OrderedCollection` элементы добавляются в конец последовательности с помощью сообщения `addLast:` (добавитьПоследним:). Обратите внимание, что при преобразовании набора с неупорядоченными элементами в набор с упорядоченными элементами, порядок элементов в создаваемом наборе непредсказуем.

Пусть переменная `numbers` содержит массив `#(5 5 1 2 2 3 4)`. Тогда:

```
numbers asBag           → Bag(5 5 4 3 2 2 1)
numbers asSet           → Set(5 4 3 2 1)
numbers asOrderedCollection → OrderedCollection(5 5 1 2 2 3 4)
numbers asSortedCollection → SortedCollection(1 2 2 3 4 5 5)
numbers asSortedCollection: [:a :b | a >= b]
                        → SortedCollection(5 5 4 3 2 2 1)
```

Подводя итог, еще раз подчеркнем, что любой набор можно преобразовать в экземпляр классов `Bag`, `Set`, `OrderedCollection`, `SortedCollection`, а любой индексированный набор можно преобразовать в экземпляр класса `Array`. Как мы увидим далее, экземпляры классов `String` и `Symbol` могут преобразовываться друг в друга, но такая возможность не определяется общим протоколом класса `Collection`. Никакие наборы не могут преобразовываться в экземпляры класса `Interval`.

7.1.3. Добавление и удаление элементов

Класс `Collection` обеспечивает общий протокол, позволяющий единообразно манипулировать любыми наборами: добавлять в набор новые элементы и удалять существующие.

Класс `Collection` **Протокол экземпляра**

add: newObject Добавляет в получатель объект `newObject` как один из его элементов; возвращает объект `newObject`.

addAll: aCollection Добавляет в получатель все элементы из набора `aCollection` как его элементы; возвращает `aCollection`.

remove: oldObject ifAbsent: aBlock Удаляет из получателя элемент, равный объекту `oldObject`; если в наборе есть несколько элементов, равных `oldObject`, удаляет первый найденный и возвращает `oldObject`; если в получателе нет такого элемента, выполняет блок `aBlock` и возвращает результат его выполнения.

remove: oldObject Аналогично предыдущему, но если в получателе нет такого элемента, сообщает об ошибке.

removeAll: aCollection Удаляет из получателя все элементы, входящие в набор `aCollection`; если какой-либо элемент из набора `aCollection` не обнаружен, сообщает об ошибке.

Последние два сообщения реализуются через сообщение `remove:ifAbsent:`. Кроме того, в их реализации используется сообщение `errorAbsentObject`, которое сообщает об отсутствии нужного элемента:

remove: anObject

```
^ self remove: anObject ifAbsent: [self errorAbsentObject]
```

removeAll: aCollection

```
aCollection do: [:each | self remove: each].
```

```
^ aCollection
```

Обратите внимание, что для каждого набора, в который добавляется новый элемент с помощью сообщения `add:` или удаляется существующий элемент с помощью сообщения `remove:`, соответствующий метод возвращает аргумент сообщения, а не набор, получивший сообщение, как почему-то нередко предполагается. В результате часто ошибочно пишут что-то вроде (`myCollection add: x`) `remove: y`, тогда как правильно было бы написать:

```
myCollection add: x; remove: y
```

Если после добавления или удаления элементов необходимо обратиться к обновленному набору, то, завершая каскад сообщений, нужно послать набору сообщение `yourself`, реализованное в классе `Object` и возвращающее получателя сообщения:

```
Set new add: x; add: y; . . . ; yourself
```

7.1.4. Проверка состояния набора

Следующие сообщения позволяют исследовать состояние набора:

Класс *Collection*

Протокол экземпляра

`includes: anObject` Возвращает `true` только тогда, когда хотя бы один из элементов получателя равен аргументу `anObject`.

`isEmpty` Возвращает `true` только тогда, когда в получателе нет элементов.

`occurrencesOf: anObject` Возвращает целое число — количество элементов получателя, равных аргументу `anObject`.

Кроме этих сообщений все наборы понимают сообщение `size`, которое возвращает количество элементов в наборе. В самом классе *Collection* такое сообщение не определено; оно наследуется из класса *Object*. Посмотрим на реализацию последнего сообщения протокола, которое использует итерационное сообщение `inject:into:`.

`occurrencesOf: anObject`

```

^self inject: 0
  into: [:occurrences :each |
    occurrences + (anObject = each ifTrue: [1]
                  ifFalse: [0])]

```

В заключение приведем примеры использования сообщений, описанных в последних двух разделах, используя для этого множество `numbers := Set new`. В последнем столбце указывается значение переменной `numbers`, если сообщение привело к его изменению. Тогда:

Выражение	Результат	Значение <code>numbers</code>
<code>numbers add: 0</code>	0	Set(0)
<code>numbers addAll: #(1 2 3 4 5)</code>	(1 2 3 4 5)	Set(0 1 2 3 4 5)
<code>numbers addAll: #(6 7)</code>	(6 7)	Set(0 1 2 3 4 5 6 7)
<code>numbers remove: 0</code>	0	Set(1 2 3 4 5 6 7)
<code>numbers removeAll: #(7 8)</code>	ERROR	Set(6 5 4 3 2 1)
<code>numbers size</code>	6	
<code>numbers isEmpty</code>	false	
<code>numbers occurrencesOf: 5</code>	1	
<code>numbers includes: 0</code>	false	
<code>numbers includes: 3</code>	true	

7.2. Индексированные наборы

Абстрактный класс `IndexedCollection` определяет общий протокол для наборов, порядок элементов которых определяется с помощью целочисленных индексов. Его непосредственным подклассом является абстрактный класс `FixedSizeCollection`, который определяет индексированные наборы с фиксированным диапазоном изменения целочисленных индексов. Класс `OrderedCollection` и его подклассы определяют индексированные наборы, размеры которых могут динамически изменяться. Хотя эти классы во многом похожи, есть характеристики, по которым они сильно отличаются. Рассмотрим некоторые из особенностей протоколов и реализаций этих системных классов.

Говоря об экземплярах класса `IndexedCollection`, мы всегда имеем в виду экземпляры его подклассов, которые полностью поддерживают определяемый им общий протокол. Класс `IndexedCollection` наследует из класса `Object` сообщения для работы с индексированными элементами: `at:`, `at:put:`, `size`. Класс `IndexedCollection` также расширяет протоколы класса `Collection`, связанные с добавлением и удалением элементов и с копированием набора. Кроме этого, его подклассы определяют много новых сообщений, производящих замену элементов набора на элементы из другого набора.

Класс `IndexedCollection` Протокол экземпляра

`atAll: aCollection put: anObject` Заменяет каждый элемент получателя с индексом из `aCollection` на объект `anObject`; возвращает обновленный набор.

`atAllPut: anObject` Заменяет каждый элемент из получателя на объект `anObject`, возвращает обновленный набор.

`first` Возвращает первый элемент получателя; сообщает об ошибке, если получатель пуст.

`last` Возвращает последний элемент получателя; сообщает об ошибке, если получатель пуст.

`indexOf: anElement ifAbsent: aBlock` Возвращает индекс первого элемента получателя, равного аргументу `anElement`; если такого элемента нет, возвращает результат выполнения блока `aBlock`.

`indexOf: anElement` Аналогично предыдущему, но если нужного элемента нет, возвращает 0.

`replaceFrom: start to: stop with: aCollection`

Возвращает получателя, в котором, начиная с индекса `start` и заканчивая индексом `stop`, элементы заменены на элементы из аргумента `aCollection`; в наборе `aCollection` должно быть ровно `stop - start + 1` элементов, иначе сообщается об ошибке.

Рассмотрим примеры отправки таких сообщений экземплярам класса `String` (аналогичные примеры можно написать и для экземпляров других подклассов класса `IndexedCollection`):

```
'aaaaaaa' size → 8
'aaaaaaa' atAll: #(2 3 8) put: $b → 'abbaaaab'
'abababab' indexOf: $b → 2
'aaaaaaa' atAllPut: $b → 'bbbbbbbbb'
'This string' first → $T
'This string' last → $g
'ABCDEFGH' indexOf: $F → 6
'ABCDEFGH' indexOf: $M ifAbsent: [0] → 0
'ABCDEFGH' indexOf: $C ifAbsent: [0] → 3
'The cow eat' replaceFrom: 5 to: 7 with: 'dog' → 'The dog eat'
```

Иногда при преобразованиях надо сохранять набор-оригинал. Чтобы не делать предварительно его копии, проще во время выполнения операции создать новый набор — копию оригинала с необходимыми изменениями. Протокол копирования класса `IndexedCollection` включает довольно много сообщений такого рода. Вот некоторые из них:

Класс <code>IndexedCollection</code>	Протокол экземпляра
<code>anIndexedCollection</code>	Возвращает копию получателя, в которую по порядку добавлены все элементы из <code>anIndexedCollection</code> .
<code>copyFrom: start to: stop</code>	Возвращает копию части получателя, состоящую из его элементов с индексами от <code>start</code> до <code>stop</code> .
<code>copyReplaceFrom: start to: stop with: aCollection</code>	Возвращает копию получателя сообщения, в которой элементы от индекса <code>start</code> и до индекса <code>stop</code> заменены на элементы набора <code>aCollection</code> .
<code>copyWith: newElement</code>	Возвращает копию получателя, в конец которого добавлен объект <code>newElement</code> .
<code>copyWithout: oldElement</code>	Возвращает копию получателя, из которого удален первый встретившийся элемент, равный <code>oldElement</code> .

Вот два простых примера использования двух последних сообщений:

```
 #(one two three) copyWith: #four → (one two three four)
 #(one two three) copyWithout: #two → (one three)
```

Так как элементы в экземплярах класса `IndexedCollection` упорядочены (см. с. 120), то возможно их перечисление, начиная с первого и до последнего. Поэтому класс `IndexedCollection` переопределяет сообщение `do:`.

do: aBlock

"Возвращает получателя. Пользуясь каждым элементом получателя в качестве аргумента блока, выполняет `aBlock`."

```

| index size |
index := 1.
size := self size.
[index > size] whileFalse: [aBlock value: (self at: index).
                             index := index + 1]

```

Но тогда возможно и перечисление от последнего до первого, для чего определяется сообщение `reverseDo: aBlock`. Кроме того, протокол расширяется следующими полезными сообщениями:

Класс `IndexedCollection`

Протокол экземпляра

findFirst: aBlock Вычисляет одноаргументный блок `aBlock` с каждым элементом получателя в качестве аргумента, начиная с первого; возвращает индекс первого элемента, для которого выполнение блока завершается возвращением значения `true`. Если такого элемента нет, сообщает об ошибке.

findLast: aBlock Вычисляет одноаргументный блок `aBlock` с каждым элементом получателя в качестве аргумента, начиная с последнего; возвращает индекс последнего элемента, для которого выполнение блока завершается возвращением значения `true`. Если такого элемента нет, сообщает об ошибке.

with: anIndexedCollection do: aBlock Вычисляет блок `aBlock`, имеющий два параметра, для каждого элемента из получателя и элемента с тем же индексом из набора `anIndexedCollection`, подставляя их в качестве аргументов блока; размеры набора-получателя и набора-аргумента должны быть равны.

Поясним последнее сообщение примером. Реализуем в классе `IndexedCollection` метод, который проверяет, содержат ли получатель и аргумент хотя бы в одной позиции равные элементы.

isEqualElementsInIndex: anIndexedCollection

```

| length |
length := self size min: anIndexedCollection size.
(self copyFrom: 1 to: length)
  with: (anIndexedCollection copyFrom: 1 to: length)
  do: [:a :b | (a = b) ifTrue: [^ true]].
^ false

```

Теперь можно посылать сообщение `isEqualElementsInIndex: anIndexedCollection` любому индексированному набору.

```

'after' isEqualElementsInIndex: 'before'      → true
#(1 2 3 4) isEqualElementsInIndex: #(4 3 2 1) → false
#Volvo isEqualElementsInIndex: #Toto         → true

```

7.2.1. Наборы фиксированного размера

Класс `FixedSizeCollection` — абстрактный подкласс класса `Collection`, экземпляры подклассов которого являются наборами фиксированного размера, определяемого при создании. Ради эффективности здесь переопределяются несколько сообщений из суперклассов (например, `collect:` и `select:`).

Экземпляр класса `Array` в качестве элементов может содержать любые объекты и представляет собой простейшую структуру для хранения объектов, связывая их с индексами. Класс `Array` полностью реализует протоколы своих абстрактных суперклассов, практически ничего нового в него не добавляя, а лишь переопределяя сообщения `printOn:` и `storeOn:`.

Класс Interval

Экземпляры класса `Interval` (будем называть их интервалами) представляют конечную арифметическую прогрессию. Элементами могут быть любые числа: целые, рациональные числа с плавающей точкой или рациональные дроби. Хотя интервал содержит все числа внутри определенного числового диапазона с определенной разностью прогрессии (приращением между соседними числами), на самом деле он реализуется вычислением членов конечной арифметической прогрессии с помощью трех параметров: `beginning` — первый член прогрессии; `end` — граница для членов прогрессии; `increment` — разность прогрессии.

Кроме сообщений класса `Number`, создающих интервал, его можно создать, посылая сообщения непосредственно самому классу `Interval`:

```
Interval from: 2 to: 5           → Interval(2 3 4 5)
Interval from: 2 to: 7 by: 2     → Interval(2 4 6 )
10.2 to: 11 by: 0.3             → Interval(10.2 10.5 10.8)
```

Заметим, что все элементы экземпляра класса `Interval` «создаются» в момент инициализации экземпляра, поэтому элементы из экземпляра нельзя удалять, а новые элементы нельзя добавлять в него. `Interval` также переопределяет метод `at:`, а метод `at:put:` запрещает. В дополнение к наследуемым сообщениям протокол экземпляра класса `Interval` поддерживает сообщение для определения разности арифметической прогрессии (сообщение `increment`).

Поскольку при выполнении итераций над арифметической прогрессией нет никакой гарантии, что создаваемый новый набор снова будет арифметической прогрессией, класс `Interval` переопределяет сообщение `species`, необходимое в итерационных вычислениях (см. с. 105):

species

```
^ Array
```

Класс **String**

Со строками — экземплярами класса **String** мы уже много раз встречались. Систематизируем наши знания и отметим некоторые их новые свойства. Строка может содержать в качестве элементов только экземпляры класса **Character**. Экземпляры класса **String** понимают все сообщения, допустимые для индексированных наборов. Кроме того, класс **String** поддерживает дополнительный протокол лексикографического сравнения своих экземпляров (<, <=, >, >=) в соответствии с содержащимися в строках символами. Строку можно конвертировать только в строку прописных символов (**asUpperCase**) или только строчных (**asLowerCase**). При передаче строк в интерфейсы операционной системы используется сообщение **asAsciiZ**, которое возвращает Си-строку (в конец строки добавляет нулевой байт); обратное преобразование выполняет сообщение **trimNullTerminator**.

Приведем еще несколько примеров работы со строками:

```
'four' < 'two'           → true
'fifth' > 'second'       → false
'first string' asUpperCase → 'FIRST STRING'
'First String' asLowerCase → 'first string'
```

В качестве примера работы со строками как с индексированными наборами приведем реализацию сообщения **asLowerCase**:

asLowerCase

```
| answer size index aCharacter |
size := self size.
answer := String new: size.
index := 1.
[index <= size] whileTrue: [
    (aCharacter := self at: index) isUpperCase
        ifTrue: [aCharacter := aCharacter asLowerCase].
    answer at: index put: aCharacter.
    index := index + 1].
^ answer
```

Класс **Symbol**

Экземпляр класса **Symbol** (СистемноеИмя) — уникальная последовательность символов, которая используется в системе Смолток в качестве имени объекта или метода. Напомним, что экземпляр класса **Symbol** можно ввести литерально, используя символ '#' как префикс определяющей последовательности символов, но используется и печатается уже введенное имя без префикса. Класс **Symbol**, в отличие от класса **String**, не поддерживает сообщения **new**:

В классе `String` определен метод `asSymbol`, который преобразует строку, получившую это сообщение, в экземпляр класса `Symbol`.

Преимуществом экземпляров класса `Symbol` является то, что система заботится об их уникальности, и два литерально совпадающие экземпляра этого класса совпадают и как объекты. Такой способ хранения позволяет вместо проверки имен на равенство использовать более эффективную проверку на идентичность. Система также запрещает все изменения имени. Поэтому для редактирования имя сначала преобразуют в строку, которую и редактируют, после чего строка может быть преобразована в новое имя. При выполнении над именами итераций, создающих новый набор, этот набор создается как строка. Класс `Symbol`, подобно классу `Interval`, переопределяет сообщение `species`:

species

`^String`

Приведем несколько примеров, в которых используются имена.

```
#Volvo size           → 5
#Volvo asString       → 'Volvo'
#at:put: first        → $a
'ABCDEFGH' asSymbol   → ABCDEFGH
Symbol mustBeSymbol: 123 → ERROR
Symbol mustBeSymbol: #String → Symbol
Symbol intern: 'Alpha' → Alpha
```

В заключение отметим одну очень полезную операцию сразу со всеми символами системы *Smalltalk Express*. Поскольку экземпляры класса `Symbol` не собираются сборщиком мусора, неиспользуемые имена постепенно накапливаются и засоряют систему. При выполнении выражения `Symbol purgeUnusedSymbols` имена, которые больше не используются, удаляются.

7.2.2. Наборы, динамически меняющие размер

Класс OrderedCollection

Класс `OrderedCollection` описывает наборы, порядок в которых определяется той последовательностью, в которой элементы добавляются в набор и удаляются из него. При этом экземпляры класса `OrderedCollection` расширяемы с обоих концов. Поэтому экземпляр класса `OrderedCollection` может выступать и как стек, и как очередь. Напомним, что *стек* — это последовательный набор, в котором удаление и добавление элементов производится на одном и том же конце, называемом вершиной стека. Часто о стеке говорят, что это набор с порядком «последним пришел — первым ушел» (“last-in

first-out”) или сокращенно lifo-набор. *Очередь* — это последовательный набор, в котором удаление элементов производится в его начале, а добавление — в конце. Часто об очереди говорят, что это набор с порядком «первым пришел — первым ушел» (“first-in first-out”) или сокращенно fifo-набор.

Из определения класса `OrderedCollection`

```
IndexedCollection subclass: #OrderedCollection
  instanceVariableNames: 'startPosition endPosition contents '
  classVariableNames: ''
  poolDictionaries: ''
```

видно, что его экземпляры имеют три переменные. Переменные `startPosition` с начальным значением 1 и `endPosition` с начальным значением 0 указывают на первый и последний элементы экземпляра. Сами элементы доступны по внешним ключам-индексам и хранятся в массиве `contents`. Поэтому все сообщения к экземпляру `OrderedCollection`, связанные с доступом к элементам набора, вызывают обращение к переменной `contents`.

Рассмотрим реализацию в классе `OrderedCollection` двух основных сообщений — `add:` и `remove:ifAbsent:`. Сообщение `add:` реализуется так же, как и сообщение `addLast:`, и добавляет новый элемент в конец набора.

add: anObject

```
endPosition = contents size
  ifTrue: [self putSpaceAtEnd].
endPosition := endPosition + 1.
contents at: endPosition put: anObject.
^ anObject
```

remove: anObject ifAbsent: aBlock

```
| index |
index := startPosition.
[index <= endPosition]
  whileTrue: [anObject = (contents at: index)
    ifTrue: [self removeIndex: index.
      ^ anObject].
    index := index + 1].
^ aBlock value
```

Код этих методов нуждается в небольшом комментарии. Выражение `self putSpaceAtEnd` расширяет набор, добавляя в него место для размещения нового элемента в случае, если все ранее выделенное пространство исчерпано. Выражение `self removeIndex: index` реально производит удаление элемента, расположенного по индексу `index`, возвращая его; если `index` окажется вне допустимых границ, будет выдано сообщение об ошибке.

Также отметим, что в связи с таким определением класса многие сообщения, наследуемые из суперклассов, в целях достижения большей эффективности переопределяются. Приведем только один простой пример:

size

\wedge endPosition — (startPosition — 1)

В дополнение к сообщениям, наследуемым из своих суперклассов, этот класс определяет много новых сообщений, которые реализуют специфику именно этого класса и позволяют добавлять, удалять и обращаться к элементам в начале набора, в конце набора и в любом другом места набора, определяя предшествующий или последующий элемент.

Класс OrderedCollection**Протокол экземпляра**

after: oldObject Возвращает элемент из получателя, стоящий после элемента **oldObject**; если в получателе нет элемента **oldObject** или элемента после него, сообщает об ошибке.

before: oldObject Возвращает элемент, стоящий до элемента **oldObject**; если в получателе нет элемента **oldObject** или элемента до него, сообщает об ошибке.

add: newObject after: oldObject Добавляет после элемента **oldObject** элемент **newObject**, возвращает **newObject**; если элемента **oldObject** в получателе нет, сообщает об ошибке.

add: newObject before: oldObject Аналогично предыдущему, но объект **newObject** добавляется в получатель перед элементом **oldObject**.

addAllFirst: anOrderedCollection Добавляет каждый элемент из аргумента **anOrderedCollection** в начало получателя, возвращает **anOrderedCollection**.

addAllLast: anOrderedCollection Добавляет каждый элемент из аргумента **anOrderedCollection** в конец получателя, возвращает **anOrderedCollection**.

addFirst: newObject Добавляет элемент **newObject** в начало получателя, возвращает **newObject**.

addLast: newObject Добавляет элемент **newObject** в конец получателя, возвращает **newObject**.

removeFirst Удаляет первый элемент в получателе и возвращает его; если получатель пуст, сообщает об ошибке.

removeLast Удаляет последний элемент из получателя и возвращает его; если получатель пуст, сообщает об ошибке.

Класс SortedCollection

Класс **SortedCollection** — подкласс класса **OrderedCollection**. Элементы в экземпляре класса **SortedCollection** упорядочиваются с помощью функции двух переменных. Эта функция представляется блоком с двумя аргументами, называемым блоком сортировки. В такие наборы новый элемент

можно добавить только с помощью сообщения `add:`, которое этим классом переопределяется. Сообщения, точно указывающие место вставки нового элемента (типа `addFirst:`, `addLast:`), не могут посылаться экземплярам класса `SortedCollection`. Поэтому все такие сообщения переопределяются, и их реализации состоят из одного выражения `^ self invalidMessage`.

Чтобы создать новый экземпляр класса `SortedCollection`, в котором реально еще нет ни одного элемента, следует использовать одно из следующих пяти выражений.

```
SortedCollection new.
```

```
SortedCollection new: 10.
```

```
SortedCollection sortBlock: [:a :b | ...].
```

```
anyCollection asSortedCollection.
```

```
anyCollection asSortedCollection: [:a :b | ...].
```

Блок сортировки может иметь любую структуру, но последнее выражение блока должно обязательно возвращать `true` или `false`. Когда во время создания экземпляра блок сортировки не указывается, по умолчанию используется блок сортировки `[:a :b | a <= b]`. Поэтому элементы такого набора должны понимать сообщение сравнения `<=`. Блок сортировки уже созданного экземпляра может быть изменен в любое время посылкой экземпляру сообщения `sortBlock: newBlock`, что приведет к автоматической пересортировке набора в соответствии с блоком `newBlock`.

Обратите внимание, что одно и то же сообщение `sortBlock: aBlock` посылается классу `SortedCollection`, если необходимо создать новый экземпляр с указанным блоком сортировки, и его экземпляру, если в экземпляре надо изменить критерий сортировки элементов. Никакого недоразумения или неоднозначности здесь нет. Вспомните, как объект, пользуясь иерархией, определяет метод, необходимый для выполнения сообщения.

Реализация большинства сообщений наследуется из суперкласса, а сообщение `add:` переопределяется, чтобы реализовать сортировку:

add: anObject

```
| index element |
```

```
endPosition = contents size
```

```
  ifTrue: [self putSpaceAtEnd].
```

```
index := endPosition.
```

```
endPosition := endPosition + 1.
```

```
[index < startPosition]
```

```
  whileFalse: [
```

```
    element := contents at: index.
```

```
    (sortBlock value: anObject value: element)
```

```
      ifFalse: [^ contents at: index + 1 put: anObject].
```

```

contents at: index + 1 put: element.
index := index - 1].
^ contents at: index + 1 put: anObject

```

Рассмотрим небольшой пример. Предположим, что мы желаем создать упорядоченный по алфавиту список имен. Пусть `ch := SortedCollection new`. По умолчанию у набора `ch` используется блок `[:a :b | a <= b]`. Элементами нашего набора будут экземпляры класса `Symbol`, которые понимают сообщения сравнения. Приведем примеры выражений и отметим значение переменной `ch` после каждой операции.

```

ch add: #Joe           → Joe      (SortedCollection(Joe))
ch add: #Bill          → Bill     (SortedCollection(Bill Joe))
ch sortBlock: [:a :b | a > b] → SortedCollection(Joe Bill)

```

7.3. Неиндексированные наборы

Неиндексированные наборы определяются двумя классами `Set` и `Bag`, экземпляры которых не отвечают на сообщения `at:` и `at:put:`. Экземпляры этих классов ведут себя точно в соответствии с протоколом класса `Collection`, и поведение их внешне очень похоже. Единственное различие состоит в том, что экземпляры класса `Bag` могут содержать дубли, а экземпляры класса `Set` не могут.

7.3.1. Класс `Set`

Класс `Set` представляет в системе конечные множества. Как и в математических множествах, равные элементы не различаются и сохраняются только один раз. Элементом множества может быть любой объект системы, за исключением объекта `nil`. В системе также присутствует класс `IdentitySet`, который считает равными только идентичные (совпадающие) объекты, и класс `SymbolSet`, оптимизированный для хранения объектов, являющихся экземплярами класса `Symbol`. Реализация класса `Set` и его подклассов довольно сложна и основана на технике, использующей функцию расстановки (хеширования). Ограничимся простыми примерами.

```

mySet := Set new.           → Set()
mySet isEmpty.              → true
mySet add: 1; add: 1; add: 3; yourself. → Set(1 3)
mySet add: 2.               → 2
mySet add: 4; yourself.     → Set(1 2 3 4)
mySet isEmpty.              → false

```

Словари

Класс `Dictionary` представляет ассоциативные массивы (словари), хранящие значения в связи с некоторым уникальным (в пределах словаря) объектом — ключом. Доступ к значениям словаря осуществляется по ключам.

По традиции, восходящей к *Smalltalk-80*, `Dictionary` реализуется как множество, элементы которого — ассоциативные пары (экземпляры класса `Association`)¹. Природа ассоциативного массива находит отражение и в протоколе сообщений этого класса. Сообщения `includes:`, `do:`, `select:`, `occurrencesOf:` переопределяются так, чтобы иметь дело со значениями словарей, а не с ключами или самими парами. Напротив, сообщения `at:`, `at:put:` переопределяются так, чтобы обращаться к ключам ассоциативных пар. А сообщение `add:` переопределяется так, чтобы работать с самими ассоциативными парами. Так как при обращении к элементам словаря обязательна ссылка на ключ, основной метод `remove:ifAbsent:` определяется как недопустимый и его функции передаются методам `removeKey: ifAbsent: removeKey:`.

Кроме того, протокол работы по поиску, перечислению и удалению в словаре расширяется для взаимодействия как с самими парами, так и с их ключами и значениями. Обратите внимание на то, *что* возвращает каждый из перечисленных ниже методов.

Класс `Dictionary`

Протокол экземпляра

- `at: aKey ifAbsent: aBlock` Возвращает значение ассоциативной пары получателя с заданным ключом `aKey`; если пары с таким ключом нет, возвращает результат выполнения блока `aBlock`.
- `associationAt: aKey` Возвращает ассоциативную пару с заданным ключом `aKey`; если ключ не найден, сообщает об ошибке.
- `associationAt: aKey ifAbsent: aBlock` Возвращает ассоциативную пару с заданным ключом `aKey`; если ключ не найден, возвращает результат выполнения блока `aBlock`.
- `associationsDo: aBlock` Возвращает получателя. Для каждой ассоциативной пары выполняет одноаргументный блок `aBlock` с этой парой как аргументом.
- `includesKey: aKey` Возвращает `true`, когда получатель содержит ассоциативную пару с ключом, равным аргументу `aKey`; иначе возвращает `false`.
- `keyAtValue: value` Возвращает первый ключ, найденный по значению `value`; если такового нет, возвращает `nil`.

¹ Ассоциативные пары — это деталь, зависящая от реализации, и некоторые новые реализации содержат словари, использующие другие техники хранения. Там весь протокол, имеющий дело с ассоциативными парами, отсутствует.

keyAtValue: value ifAbsent: aBlock Возвращает первый ключ, найденный по значению *value*; если такового нет, возвращает результат выполнения блока *aBlock*.

keys Возвращает экземпляр класса **Set** (поскольку ключи уникальны), содержащий все ключи получателя.

keysDo: aBlock Возвращает получателя. Для каждого ключа получателя выполняет блок *aBlock* с этим ключом как аргументом.

removeKey: aKey Удаляет из получателя пару с ключом, равным *aKey*, возвращает измененного получателя; если такой пары нет, сообщает об ошибке.

removeAssociation: anAssociation Удаляет из получателя пару, равную аргументу *anAssociation*, возвращает измененного получателя; если такой пары нет, сообщает об ошибке.

values Возвращает экземпляр класса **Bag** (поскольку значения не уникальны и могут повторяться), содержащий все значения словаря.

Для демонстрации возможностей словарей создадим словарь с именем **opposites**, состоящий из слов-ключей и их антонимов-значений. При этом для добавления в словарь элементов используем два разных сообщения.

```
| opposites |
opposites:= Dictionary new.
opposites at: #hot put: #cold.
opposites at: #push put: #pull.
opposites at: #stop put: #go.
opposites at: #come put: #go.
opposites add: (Association key: #front value: #back).
opposites add: (Association key: #top value: #bottom).
```

ключ	значение
hot	cold
push	pull
stop	go
come	go
front	back
top	bottom

Обратите внимание на еще одно различие между **at:put:** и **add:**. В случае **at:put:** возвращается *значение* (аргумент ключевого слова **put:**); а в случае **add:** — *ассоциативная пара*.

Теперь словарь **opposites** состоит из пар «ключ → значение», приведенных в таблице. Не забывайте, что одно и то же значение может быть связано с любым числом ключей, в то время как ключи уникальны.

В следующих примерах, применяя сообщения для проверки, посмотрим, есть ли в словаре **opposites** некоторые значения:

```
opposites size           → 6
opposites includes: #cold → true
opposites includes: #hot  → false
opposites occurrencesOf: #go → 2
opposites at: #stop put: #start → start
```

Поскольку каждый ключ в словаре уникален и может появиться только один раз, в последнем выражении для уже существовавшего в словаре ключа `stop` определяется новое значение `start`, которое заменяет прежнее значение `go`. Аналогичным образом можно проверить наличие в словаре `opposites` некоторых ассоциативных пар и ключей.

```
opposites includes: (Association key: #come value: #go) → true
opposites includesKey: #come → true
opposites includesKey: #hot → true
opposites includesKey: #go → false
```

Экземпляры класса `IdentityDictionary` по понимаемым ими сообщениям ничем не отличаются от экземпляров его суперкласса `Dictionary`. Но класс `IdentityDictionary` почти все наследуемые методы переопределяет так, чтобы проверять идентичность ключей, а не их равенство (см. с. 88).

Класс `SystemDictionary` — подкласс `IdentityDictionary`; он имеет в системе только один экземпляр — системный словарь с именем `Smalltalk`. Его ключи — системные имена. Дополнительно класс `SystemDictionary` содержит сообщения для управления самой системой через свой единственный экземпляр (см. раздел 3.4.4).

7.3.2. Класс `Bag`

Экземпляр класса `Bag` (простой набор, или «мешок») может содержать дубли, ведет себя согласно протоколу для всех наборов, и его поведение похоже на поведение экземпляров класса `Set`,

Возможность хранения дублей подчеркивается наличием сообщения `add: newObject withOccurrences: anInteger`, которое добавляет объект `newObject` в получатель в количестве `anInteger` и возвращает `newObject`.

В качестве примера рассмотрим код, который вычисляет частоту букв английского алфавита в некотором текстовом файле, например, с именем `packing.txt`.

```
| input frequency output ch |
input := File pathName: 'packing.txt'.
output := File pathName: 'frqltrs.txt'.
frequency := Bag new.
[input atEnd] whileFalse: [
    ch := input next.
    (ch isLetter) ifTrue: [frequency add: ch asLowerCase]].
frequency asSet asSortedCollection do: [:ch |
    output nextPutAll: ch printString;
```

```
nextPutAll: ' -> '  
nextPutAll: (frequency occurrencesOf: ch) printString;  
cr].
```

```
output close.
```

```
input close.
```

В результате выполнения этой последовательности выражений будет создан новый файл с именем `frq1trs.txt`, в который в алфавитном порядке будут записаны строки (`cr` — переход на новую строку). Каждая строка состоит из английской буквы (поскольку это символ, ему предшествует `$`), стрелки вида `'->'` и числа, возникшего в результате вычисления того, сколько раз встречается данная буква в текстовом файле `packing.txt`. В этом примере переменная `frequency` содержала экземпляр класса `Bag`, `ch` — следующий символ из `input`, а переменные `input` и `output` — потоки. В данном примере используется сообщение `nextPutAll:`, записывающее в поток все объекты из набора, переданного в качестве аргумента.

ГЛАВА 8

ПОТОКИ И ФАЙЛЫ

С потоками мы уже встречались, когда обращались к файлам на диске и использовали экземпляры класса `FileStream`. Система Смолток поддерживает много различных потоков, которые используются для доступа к файлам, внешним устройствам и внутренним объектам как к последовательности объектов. Причем поток позволяет нам как использовать последовательность, так и создавать ее. В отличие от наборов, поток не поддерживает ни непосредственное обращение к конкретному элементу с некоторой характеристикой (`at:`, `last`, `first`), ни работу сразу со всем набором. Поток предоставляет доступ одновременно только к одному (текущему) элементу последовательности, следующее обращение к потоку сделает текущим уже другой элемент последовательности. Поток некоторым образом сам «помнит», какой элемент был использован последним. Это запоминаемое потоком положение называется *указателем позиции* (или просто указателем). Мы будем использовать выражение «поток над набором», понимая под этим доступность элементов набора через указатель позиции, производя при этом чтение или запись элемента (одного за одно обращение), и, возможно, смешивая эти операции. Создавая несколько потоков над одним и тем же набором, можно поддерживать несколько указателей.

Основные потоки представлены в системе *Smalltalk Express* экземплярами классов из иерархии класса `Stream`.

`Stream`

`ReadStream`

`WriteStream`

`ReadWriteStream`

`FileStream`

Поток

ПотокЧтения

ПотокЗаписи

ПотокЧтенияЗаписи

ФайловыйПоток

Есть разные способы поддержки указателя. Наиболее распространенный подход использует в качестве указателя целочисленный индекс, который увеличивается каждый раз, когда поток обращается к элементу. Такой подход может применяться и для любого индексированного набора, и для внешних файловых потоков.

Второй способ получить доступ к последовательности однородных элементов состоит в том, чтобы использовать для образования новых объектов некий *итератор* (или *генератор*), который будет находить (или, соответственно, создавать) следующий элемент набора. Таким образом, можно, например, создавать потоки над наборами, не являющимися после-

довательностями (например, графами). Примером такого потока является класс `Random`, генерирующий последовательность псевдослучайных чисел, который мы рассмотрим в конце главы.

8.1. Протокол класса **Stream**

Класс `Stream` является абстрактным классом и определяет общий протокол потока. Экземпляры классов из потоковой иерархии создаются при отправке классу сообщения `on: aCollection`, в котором аргумент `aCollection` определяет набор доступных создаваемому потоку элементов. Обратите внимание, что поток не может создаваться отправкой классу сообщения `new`, потому что поток должен быть проинформирован о том, какой набор ему доступен и каков начальный указатель позиции.

Класс `Stream` — первый встретившийся нам класс, в котором определен общий пул, в данном случае это пул с именем `CharacterConstants`, содержащий часто используемые символьные константы, такие как `Space` — пробел, `Lf` — перевод строки, `Tab` — табуляция и т.д.

Протокол класса `Stream` включает общие сообщения для чтения элементов из потока, записи элементов в поток, а также сообщения для работы с указателем позиции. Но не все потоки поддерживают операции чтения и записи одновременно. Класс `ReadStream` определяет потоки только для чтения и сообщение `next`, а класс `WriteStream` — только для записи и сообщение `nextPut: anObject`. Поэтому сообщения для чтения нельзя отправлять экземплярам класса `WriteStream` (но можно отправлять экземплярам тех его подклассов, которые его определяют сами), а сообщения для записи нельзя отправлять экземплярам класса `ReadStream`. Укажем здесь только основные сообщения для чтения и записи.

Класс `Stream`

Протокол экземпляра

`next` Читает следующий доступный элемент из получателя, т.е. возвращает следующий доступный элемент и продвигает указатель позиции вперед на один элемент; сообщает об ошибке, если указатель стоит в конце получателя.

`next: anInteger` Читает следующие `anInteger` доступных элементов из получателя и возвращает их в наборе (обычно это набор того же самого класса, что и тот, над которым определен поток).

`nextMatchFor: anObject` Читает следующий доступный элемент из потока и возвращает результат сравнения его с `anObject` (`true` или `false`).

`peek` Возвращает следующий доступный элемент потока (как и сообщение `next`), но не продвигает указатель позиции («подсматривает» следующий элемент); возвращает `nil`, если указатель в конце потока.

-
- peekFor: anObject** «Подсматривает» следующий элемент потока (как сообщение **peek**), при этом, если этот элемент равен **anObject**, то продвигает указатель на следующую позицию и возвращает **true**, иначе возвращает **false** и не меняет указатель.
- upTo: anObject** Возвращает набор элементов, доступных получателю, начиная от следующего доступного элемента и до элемента **anObject**, не включая его (если **anObject** отсутствует в наборе, то — до конца набора); устанавливает указатель позиции за элемент **anObject**.
- nextPut: anObject** Сохраняет аргумент **anObject**, как следующий элемент получателя, и продвигает указатель за **anObject**; возвращает **anObject**.
- nextPutAll: aCollection** Сохраняет элементы из аргумента **aCollection**, как следующие элементы получателя; возвращает **aCollection**.
- next: anInteger put: anObject** Сохраняет аргумент **anObject**, как следующий элемент получателя **anInteger** раз; возвращает **anObject**.
-

Чтобы определить, возможен ли доступ к потоку, каково состояние потока и его переменных, протокол класса **Stream** определяет сообщения доступа и проверки состояния.

Класс **Stream**

Протокол экземпляра

-
- atEnd** Сообщает, имеет ли получатель доступ к какому-либо элементу.
- isEmpty** Возвращает **true**, если набор, доступный получателю, не имеет элементов, иначе возвращает **false**.
- contents** Возвращает все элементы из набора, доступного получателю, в виде набора, аналогичного тому, над которым струится поток.
- do: aBlock** Вычисляет блок **aBlock** для каждого доступного элемента получателя от текущей позиции и до конца потока.
-

Реализация сообщения **do**: основывается на сообщениях **atEnd** и **next**.

do: aBlock

[self atEnd] whileFalse: [aBlock value: self next].

Поэтому **do**: нельзя посылать экземплярам класса **WriteStream**.

Так как экземпляру класса **Stream** «известно», как хранится указатель позиции, класс поддерживает сообщения для работы с ним. Но все сообщения из этого протокола, кроме первого, возвращают не значение переменной **position**, а сам поток.

Класс **Stream**

Протокол экземпляра

-
- position** Возвращает текущий указатель позиции потока.
- position: anInteger** Устанавливает указатель на позицию, равную **anInteger**; если аргумент выходит за допустимые границы, сообщает об ошибке.
- reset** Устанавливает указатель в начало потока.

`setToEnd` Устанавливает указатель в конец потока.

`skip: anInteger` Устанавливает указатель позиции потока в позицию, равную сумме текущей позиции и аргумента `anInteger`, но так, чтобы оставаться в допустимых границах.

`skipTo: anObject` Устанавливает указатель позиции потока на следующий за `anObject` элемент и возвращает `true`; если такого элемента нет, возвращает `false` и устанавливает указатель в конец потока.

Рассмотрим пример, в котором проиллюстрируем позиционирование и протокол чтения, используя экземпляр класса `ReadStream`, определение которого не добавляет в структуру экземпляра ничего нового по сравнению с суперклассом, но впервые определяет сообщение `next`, нереализованное в классе `Stream`, а также некоторые другие частные сообщения, необходимые для корректного создания нового экземпляра.

```
Colors := ReadStream on:
    #(red blue green yellow pink cyan magenta brown).
```

Тогда

```
Colors position      → 0
Colors next          → red
Colors next: 3      → (blue green yellow)
Colors peek         → pink
Colors peekFor: #blue → false
Colors upTo: #magenta → (pink cyan)
Colors skip: -4     → aReadStream
Colors position     → 3
Colors skipTo: #pink → true
Colors upTo: #red   → (cyan magenta brown)
Colors atEnd        → true
```

8.1.1. Особенности класса `WriteStream`

Подкласс `WriteStream`, как следует из его имени, предоставляет доступ к потоку только для записи.

Именно в этом классе переопределяется основное сообщение для записи `nextPut:`, используемое системой Смолток в методах печати и сохранения любого объекта. Напомним, что каждый объект в системе может отвечать на сообщения `printString`, `printOn: aStream` и `storeString`, `storeOn: aStream` из протокола класса `Object`. Методы, реализующие эти сообщения, состоят из последовательности сообщений к аргументу, который является экземпляром класса `WriteStream`.

Следующие сообщения поддерживаются классом **WriteStream** для того, чтобы позволить использовать краткие выражения для разделителей в потоках над строками.

Класс **WriteStream**

Протокол экземпляра

cr Сохраняет символы **Cr** — возврат каретки (carriage return) и **Lf** — перевод строки (line feed) как следующие два элемента потока.

space Сохраняет символ пробела как следующий элемент потока.

tab Сохраняет символ табуляции как следующий элемент потока.

Класс **ReadWriteStream** — подкласс **WriteStream**, экземпляры которого поддерживают протоколы и класса **ReadStream**, и класса **WriteStream**, поэтому в доступный такому потоку набор можно как записывать, так и читать информацию. Все методы для записи информации в поток этот класс наследует из класса **WriteStream**, а все методы чтения приходится определять заново. Это недостаток одиночного наследования, используемого в смолтоковских системах.

8.2. Особенности класса **FileStream**

Класс **FileStream** — подкласс класса **ReadWriteStream** — обеспечивает интерфейс высокого уровня с файловой системой¹. Все обращения к внешним файлам выполняются через экземпляр класса **FileStream**, используя объект класса **File** (Файл). Класс **Directory** (Каталог) обеспечивает доступ к дискам и каталогам.

Файловые потоки допускают использование всех ранее описанных сообщений из протокола потоков. Наиболее эффективный способ считывания информации из файлового потока — посылка ему сообщения **next**; а записи в файловый поток — посылка сообщения **nextPutAll**; которые, как и некоторые другие сообщения, переопределяются классом **FileStream**. Большая часть обращений к файлам происходит на уровне, обеспеченном протоколом потоков.

Файловые потоки создаются путем посылки сообщения **pathName**: классу **File**, в котором частично или полностью задается путь доступа к файлу:

¹ В других реализациях иерархия потоковых классов для работы с внешними объектами будет другой. Другими будут и классы для работы с файлами. Причина отличий в том, что система *Smalltalk Express* предназначена для работы только в очень похожих *Windows* и *OS/2*.

```
File pathName: 'c:\smalltalk\chapter.1'
```

```
File pathName: 'chapter.1'
```

В первом выражении задано полное имя файла. Во втором выражении имя определено не полностью. В таких случаях недостающие части пути к файлу извлекаются из глобальной переменной *Disk*, хранящей экземпляр класса *Directory*. Эта переменная представляет тот каталог, из которого был запущен Смолток (в данном случае предполагается, что переменная *Disk* содержит каталог 'c:\smalltalk\'). Посылка классу *File* сообщения *pathName:* всегда открывает файл для записи и чтения. Чтобы открывать файл только для чтения, используется сообщение *pathNameReadOnly:*.

Другим способом создания файлового потока является посылка существующему экземпляру класса *Directory* сообщения, задающего имя конкретного файла в каталоге. В следующих примерах файловый поток создается сообщением, посылаемым объекту *Disk*:

```
Disk file: 'chapter.1'.    Disk newFile: 'jink.fil'
```

В таких выражениях аргументом может быть только имя файла. Путь к файлу не указывается, поскольку сообщение посылается каталогу, в котором и будет располагаться файл. Оба сообщения создают файл в каталоге, если он не существовал ранее. Различие между сообщениями *file:* и *newFile:* заключается в том, что при использовании второго сообщения всегда создается новый файл, удаляя, если необходимо, ранее существовавший файл с тем же именем.

При изменении содержимого файлового потока немедленного обновления информации в файле на диске не происходит. Для принудительного сохранения информации существуют два сообщения, посылаемые файловым потокам: *close* и *flush*. Различие между этими сообщениями заключается в том, что сообщение *close* после записи на диск всех изменений, сделанных в потоке, закрывает файл, делая невозможным дальнейший доступ к нему. Сообщение *flush* заставляет систему действительно выполнить все операции записи, обновляя файл на диске, но файловый поток остается доступным для дальнейшей работы.

8.2.1. Краткий обзор классов поддержки

Приведем краткий обзор классов, используемых для работы с файлами.

Класс Directory

Класс *Directory* представляет каталоги файловой системы. С глобальной переменной системы *Disk*, содержащей тот каталог, из которого запускается система *Smalltalk Express*, мы уже познакомились. С помощью выражений,

подобных приведенным ниже, можно создавать произвольные объекты-каталоги: `SampleDir := Directory new drive: $a; pathName: '\dirname'`.

Необходимо отметить, что создание объекта, представляющего каталог, *не означает* создания каталога на диске. Для создания реального каталога на диске такому объекту необходимо послать сообщение `create`. Например, `SampleDir create`. Экземплярам класса `Directory` можно посылать сообщения, возвращающие все подкаталоги данного каталога (`subdirectories`), все файлы в каталоге (`filesNamed: '*.*'`), а также сообщения, создающие новый файл в каталоге, новый подкаталог и т.д.

Класс File

Класс `File` представляет в Смолтоке дисковые файлы (то есть именованные наборы данных на диске) и обеспечивает работу с ними.

Следующие методы класса позволяют управлять файловой системой.

Класс File	Протокол класса
<code>copy: oldFile to: newFile</code>	Копирует файл с именем <code>oldFile</code> в файл с именем <code>newFile</code> .
<code>open: aString in: aDirectory</code>	Возвращает экземпляр класса <code>File</code> , открытый на файле с именем <code>aString</code> в каталоге <code>aDirectory</code> .
<code>pathName: aString</code>	Возвращает файловый поток, связанный с файлом, задаваемым аргументом <code>aString</code> .
<code>pathName: aString in: aDirectory</code>	Возвращает файловый поток, связанный с файлом, задаваемым аргументами сообщения.
<code>remove: aString</code>	Удаляет (стирает) файл с именем <code>aString</code> .
<code>rename: oldString to: newString</code>	Файл с именем <code>oldString</code> переименовывает в <code>newString</code> .

Много полезных сообщений класса `File` содержит протокол экземпляра. Вот только некоторые из них.

Класс File	Протокол экземпляра
<code>close</code>	Закрывает файл.
<code>directory</code>	Возвращает строку с именем каталога, содержащего файл.
<code>name</code>	Возвращает строку, содержащую имя файла.
<code>open</code>	Открывает файл.
<code>getDate</code>	Возвращает массив, содержащий время и дату файла.
<code>size</code>	Возвращает размер файла в байтах.

Класс *ObjectFiler*

Для сохранения объекта и последующего восстановления сохраненного объекта система *Smalltalk Express* предоставляет специальный механизм, связанный с классом *ObjectFiler* (ХранительОбъекта). Этот класс, в частности, позволяет:

- создать или переписать конкретный файл с именем *aPathName*, хранящий объект *anObject*, выполняя выражение вида
`ObjectFiler dump: anObject newFile: aPathName.`
- открыть в интерактивном режиме диалоговое окно для определения имени загружаемого в образ файла, выполняя выражение вида
`ObjectFiler load.`
- загрузить в образ конкретный файл, хранящий нужный объект, выполняя выражение вида
`ObjectFiler loadFromPathName: aPathName.`

Если во время загрузки произойдет фатальная ошибка, класс *ObjectFiler* возвратит *nil* и отобразит информацию о возникшей проблеме.

8.3. Поток генерируемых элементов

Создадим в системе *Smalltalk Express* класс *Random* (подкласс класса *Stream*), экземпляр которого — генератор псевдослучайных чисел. Есть несколько алгоритмов создания такого генератора. Мы используем тот, который основан на конгруэнтном методе Лемера². Датчик Лемера псевдослучайных чисел строится по правилу $X_{n+1} = (a \cdot X_n + c) \bmod m$ и определяет свои элементы, отправляясь от некоторого начального значения X_0 (в нашем классе — *seed*). Теория, которая описывает поведение такой последовательности чисел, изложена в [32, § 3.2.1]. Выбор конкретных значений для коэффициентов a, c, m в методе *next* обусловлен этой теорией.

Так как поток является генерируемым и можно вычислить сколь угодно много элементов, экземпляры класса *Random* не должны отвечать на сообщение *contents*, а сообщение *atEnd* всегда должно возвращать *false*. Хотя класс *Random* и поддерживает сообщение *do:*, наследуемое из класса *Stream*, но соответствующий метод, однажды начавшись, никогда не закончится без целенаправленного вмешательства программиста. Можно послать экземпляру класса *Random* и сообщение *next: anInteger*, чтобы получить упорядоченный набор из *anInteger* случайных чисел.

² В других реализациях языка Смолток класс *Random* может входить в поставляемую библиотеку классов, и в его основе могут лежать другие алгоритмы.

Приводимую ниже реализацию класса `Random` читатель должен встроить в свою систему *Smalltalk Express*, поскольку этот класс нам потребуется в примерах. Напомним, что для введения в систему нового класса надо воспользоваться окном просмотра иерархии классов (см. раздел 3.2.2).

Перед выходом из системы Смолток следует сохранить образ системы, который теперь содержит класс `Random`. Всегда следует сохранять и сам новый класс в текстовом файле; для этого надо выбрать его в панели классов и воспользоваться пунктом `File Out...` из меню `Classes`. Из текстового файла класс можно установить в систему, воспользовавшись пунктом `Install...` из меню `File`. В текстовом файле определение класса `Random` будет таким:

```
Stream subclass: #Random
  instanceVariableNames:
    'seed '
  classVariableNames: ''
  poolDictionaries: '' !

!Random class methods !

new
  ^self basicNew initialize!

!Random methods !

initialize
  self setSeed.
  collection := Array new.
  position := readLimit := 0.
  ^self!

atEnd
  "Генерируемый поток бесконечен."
  ^false!

contents
  "Генерируемый поток бесконечен."
  ^self invalidMessage!

position: anInteger
  "Перемещение позиции чтения в генерируемом
  бесконечном потоке бессмысленно."
  ^self invalidMessage!
```

next

```
"Линейный конгруэнтный метод Лемера. Возвращает
псевдослучайное число из интервала (0, 1)."
| temp |
[ seed := 13849 + (27181*seed) bitAnd: 8r177777.
  temp := seed/65536.0 .
  temp = 0 ] whileTrue: [~].
^temp!
```

setSeed

```
"Частный. Для определения псевдослучайного начального
значения взять время системных часов компьютера.
Это большое положительное целое число, поэтому нужны
только младшие 16 битов."
seed := Time millisecondClockValue bitAnd: 8r177777! !
```

Мы записали только содержательную часть определения класса. Следует также переопределить все недопустимые наследуемые методы, используя выражение `^self invalidMessage` (как это сделано выше в методах `contents` и `position`:). Таких методов достаточно много.

Вернемся к экземпляру класса `Random`. Генератор с именем `rand`, каждый раз порождающий равномерно распределенные случайные числа из интервала $(0, 1)$, можно теперь создать с помощью выражения

```
rand := Random new.
```

Теперь, когда потребуется случайное число, объекту `rand` надо послать сообщение `next`.

ГЛАВА 9

НЕЗАВИСИМЫЕ ПРОЦЕССЫ

Процессом называется последовательность действий, описанная выражениями языка и выполняемая виртуальной машиной системы Смолток. Есть процессы, контролирующие асинхронные устройства, например, клавиатуру, мышшь, часы реального времени или доступную память системы. Для пользователя наиболее важными являются процессы, выполняющие действия, непосредственно им определяемые, например, редактирование текста или изображения, определение класса. Такие процессы должны общаться с процессами, контролирующими клавиатуру и мышшь, чтобы определить, что делает пользователь. Процессы могут добавляться в систему, например, для работы с объектом.

Все вычисления в объектно-ориентированной системе происходят последовательно в рамках текущего (*активного*) процесса, как результат посылки объектами друг другу сообщений. Объект всегда ждет, пока не получит ответа на посланное им сообщение. Так, в процедурном языке в любой момент времени есть стек не завершенных процедурных вызовов и есть единственная процедура, которая является активной. В системах Смолток тоже есть стек посланных, но не завершенных сообщений, и есть единственный метод, который является активным.

Смолтоковские системы, работающие на компьютерах с одним процессором, моделируют параллельную обработку, определяя не один, а множество стеков посланных, но не завершенных сообщений. Каждый стек, описывая текущее состояние вычислений в отдельном процессе, представляется объектом — экземпляром класса **Process** (Процесс). В любой момент времени выполняется только один процесс. Но в системе могут существовать несколько процессов, готовых к выполнению. Существуют точно определенные условия, при которых происходит переключение с одного процесса на другой. При определенных условиях остановленный, но не завершенный процесс может быть возобновлен.

Кроме уже упомянутого класса **Process** — подкласса класса **OrderedCollection** — для поддержки множества независимых процессов система *Smalltalk Express* использует еще два класса — **ProcessScheduler** (Планировщик-Процессов) и **Semaphore** (Семафор). Экземпляр класса **Process** определяет последовательность действий, которая может выполняться независимо от действий, определяемых другими экземплярами этого класса. Единственный экземпляр класса **ProcessScheduler** с именем **Processor** (Процессор) планирует использование виртуальной машины, которая реально выполня-

ет все действия, представленные в системе экземплярами класса `Process`. В системе может существовать большое количество процессов, готовых к выполнению, и именно `Processor` определяет, какой из них виртуальная машина будет выполнять в данное время. Экземпляр класса `Semaphore` позволяет независимым процессам согласовывать (синхронизировать) свои действия друг с другом, посылая семафору всего два сообщения: `signal` (сигнал), который разрешает продолжить работу, и `wait` (ждать), который останавливает процессы. Обеспечивая простую форму связи между процессами, семафоры могут применяться для создания более сложных механизмов взаимодействия.

9.1. Процессы и управление ими

Новый процесс создается или посылкой блоку унарного сообщения `fork`, или посылкой объекту `Processor` ключевого сообщения `fork: aBlock`. В результате будет создан новый экземпляр класса `Process`, который будет включен в список процессов и выполнен в соответствии с существующими правилами. Например,

```
[Transcript show: 'Hello'; cr] fork. Transcript show: 'Bye'; cr.
```

В данном примере создается отдельный процесс для выполнения кода, содержащегося внутри блока, а затем продолжается выполнение текущего активного процесса (выполняется код, записанный после блока). Действия, выполняемые новым процессом, описываются выражениями блока. Сообщение `fork` имеет тот же эффект воздействия на эти выражения, что и сообщение `value`, но отличается от него способом возвращения результата сообщения. Когда блок получает сообщение `value`, он ждет выполнения всех своих выражений, пока не вернет результат. Когда же блок получает сообщение `fork`, он возвращает управление после создания нового процесса, не дожидаясь выполнения своих выражений. Это позволяет выполнять выражения, следующие за сообщением `fork`, независимо от выполнения выражений в блоке. Так как выражения блока могут оставаться не вычисленными, когда происходит выход из `fork`, то значение `fork` не должно зависеть от значений выражений блока. Поэтому как результат выполнения сообщения `fork` блок возвращает себя. При выполнении примера в окне `Transcript` отобразятся две фразы:

```
Hello  
Bye
```

Вывод в окне `Transcript` показывает, что новый процесс инициализируется прежде, чем будет продолжен текущий процесс.

Виртуальной машине доступен только один процессор, способный выполнять последовательности действий, представленные экземплярами класса `Process`. Следовательно, когда создается экземпляр класса `Process`, входящие в него операции могут и не начать выполняться немедленно. Процесс, ожидающий выполнения, называется *пассивным*. За порядком выполнения процессов следит объект `Processor`. Обычно процессы используют процессор, опираясь на простое правило: «первым пришел, первым обслужили».

Чтобы лучше управлять порядком выполнения процессов, используется механизм *приоритетов*. Приоритет — целое число. Процесс с высоким приоритетом получит в свое распоряжение процессор раньше любого другого процесса с низким приоритетом, независимо от порядка, в котором они создавались. Когда процесс создается без явного указания приоритета (как в наших первых примерах), он получает тот же приоритет, что и экземпляр класса `Process`, создавший его. Таким образом, порядок активизации процессов зависит от приоритета процесса, а среди процессов равного приоритета — от порядка, в котором они находятся в списке готовых к исполнению процессов.

Чтобы задать приоритет создаваемого процесса, надо или послать сообщение `forkAt: anInteger` блоку, или сообщение `fork: aBlock at: anInteger` объекту `Processor`. Кроме того, приоритет процесса всегда можно изменить, посылая ему сообщение `priority: anInteger`, в котором аргумент `anInteger` задает новый уровень приоритета. Рассмотрим следующий пример:

```
Processor fork: [Processor fork:
  [Transcript show: 'world!'; cr] at: 2.
  Transcript show: 'Hello '] at: 3.
```

В нем создаются два новых процесса, один с приоритетом, равным двум, а другой с приоритетом, равным трем. Так как процесс с более высоким приоритетом всегда планируется на исполнение первым, вывод в окно `Transcript` имеет вид `Hello world!`.

Всего в *Smalltalk Express* существует семь приоритетных уровней:

- 7 — `topPriority` (высший приоритет),
- 6 — `realTimePriority` (приоритет реального времени),
- 5 — `highPriority` (высокий приоритет),
- 4 — `userPriority` (приоритет пользовательского интерфейса),
- 3 — `lowPriority` (низкий приоритет),
- 2 — `backgroundPriority` (приоритет фоновых операций),
- 1 — `idleTaskPriority` (приоритет простоя).

Например, рассмотрим следующие выражения, предполагая, что они выполняются в рамках текущего процесса с приоритетом 4:

```
[ [Transcript show: 'now '] forkAt: 6.  
  [Transcript show: 'is '] fork.  
  Transcript show: 'the '  
] forkAt: 5.  
Transcript show: 'time '  
Transcript show: (Time now) asString.
```

В результате в окне Transcript появится текст: `now is the time 11:25:36`.

При реальном программировании в смолтоковской системе приоритеты не определяют с помощью целых чисел. Нужный приоритет всегда получают, посылая соответствующее сообщение объекту `Processor` (эти сообщения определены в протоколе класса `ProcessScheduler`). Поэтому приведенный ранее пример, выводящий в окно Transcript фразу `Hello world!`, должен быть записан в виде

```
Processor fork: [Processor fork:  
  [Transcript show: 'world!'; cr] at: Processor backgroundPriority.  
  Transcript show: 'Hello '] at: Processor lowPriority.
```

Кроме того, для объекта `Processor` предусмотрено сообщение `yield`, которое приостанавливает выполнение активного процесса и помещает его в конец списка ожидающих выполнения процессов с тем же приоритетом. При этом первый процесс из этого списка становится активным процессом. Если других процессов с таким же приоритетом в списке нет, сообщение `yield` ничего не делает.

В начале главы мы уже отмечали, что процессы могут находиться в системе в разных состояниях: активного (исполняемого в данный момент) или пассивного (неисполняемого) процесса. В свою очередь, пассивные процессы подразделяются на готовые к выполнению (`ready`), заблокированные (`blocked`) и мертвые (`dead`). Таким образом, любой процесс в каждый момент времени может находиться в одном из четырех состояний:

`active` — единственный процесс, выполняемый системой в данный момент; этот процесс — значение глобальной переменной `CurrentProcess`;

`ready` — созданный процесс, который система могла бы выполнить в любое время;

`blocked` — процесс, выполнение которого прервано (блокировано) в результате остановки (на семафоре) или в результате ошибки.

`dead` — процесс, который уже никогда не будет выполняться; сборщик мусора удалит его, если в системе на него не будет ссылок.

Переходами процессов из одного состояния в другое управляет объект `Processor`. Переходы происходят по следующим правилам:

- вновь созданный процесс устанавливается в состояние **ready**;
- процесс из состояния **ready** переходит в состояние **active**, если среди процессов с равным приоритетом он дольше всех находится в состоянии **ready**, нет процессов с более высоким приоритетом, и:
 - (1) активный процесс переходит в состояние **blocked** или **dead**;
 - (2) активный процесс имеет более низкий приоритет, чем процесс, находящийся в состоянии готовности;
 - (3) процесс, находящийся в состоянии готовности, имеет приоритет активного процесса, а **Processor** получает сообщение **yield**;
- активный процесс переходит в состояние **ready**, когда он заменяется процессом, находящимся в состоянии готовности, при условиях, описанных выше в пунктах (2, 3);
- активный процесс переходит в состояние **blocked**, когда семафору, который не имеет лишних сообщений **signal**, процесс посылает сообщения **wait**;
- заблокированный процесс переходит в состояние **ready**, когда он становится первым в очереди процессов, ожидающих у семафора, и семафору посылается сообщение **signal**;
- активный процесс переходит в состояние **dead** или когда достигается конец блока, вызвавшего создание этого процесса, или когда выполняется выражение **Processor terminateActive**.

Процессы и семафоры используются также, если пользователь вводит и обрабатывает необходимую для него информацию. Единственным процессом, который в системе *Smalltalk Express* отвечает на события, возникающие при нажатии клавиш на клавиатуре или кнопок мыши, является процесс пользовательского интерфейса (**UserInterfaceProcess**). В этом процессе, сменяя друг друга, происходит либо ответ на событие ввода, либо ожидание следующего ввода посредством посылки глобальной переменной **KeyboardSemaphore** сообщения **wait**.

Когда никаких действий по вводу не происходит, могут выполняться другие процессы, имеющие более низкие приоритеты. Объект **Processor** гарантирует существование неактивного процесса с самым низким приоритетом (**idleTaskPriority**), который выполняется тогда, когда нет никаких других неактивных процессов.

Когда при работе пользователя с системой возникает ошибка, открывается окно отладчика. Отладчик открывается всегда, независимо от того, в каком процессе произошла ошибка. Когда сообщение **error**: посылается в процессе пользовательского интерфейса, текущий процесс приостанавливается (становится заблокированным), а для отладчика создается новый от-

дельный процесс пользовательского интерфейса. Исследуя с помощью отладчика состояние процесса, в котором произошла ошибка, ошибку можно найти. Если сообщение `error`: посылается в процессе, который не является процессом пользовательского интерфейса (`non-user-interface`), этот процесс блокируется, а сообщение, вызывающее отладчик (вместе с информацией, описывающей ошибку) помещается в очередь `PendingEvents`. Очередь `PendingEvents` используется здесь для ускорения создания отладочного процесса пользовательского интерфейса, когда нет никаких других действий, связанных с вводом (когда больше нечего обрабатывать).

Для того чтобы лучше понять происходящее с процессами, надо поближе познакомиться с классом `Semaphore`.

9.2. Семафоры

Процессы могут функционировать независимо друг от друга. Некоторые процессы, независимые по существу, время от времени все же должны взаимодействовать. Они могут обращаться к одним и тем же устройствам, к одной и той же информации или могут передавать друг другу некоторую информацию. Экземпляр класса `Semaphore` (Семафор) обеспечивает простейшую форму связи между независимыми процессами. Однако наличие семафора позволяет программисту реализовывать любые другие механизмы взаимодействия между процессами.

Для организации связи семафор использует 1 бит информации — сигнал от одного процесса к другому. На одном конце связи процесс, ожидая некоторого события, посылает семафору сообщение `wait`. На другом конце связи процесс сообщает о том, что это событие произошло, посылая семафору сообщение `signal`. Не имеет значения, в каком порядке эти сообщения посылаются. Процесс из очереди `waitingProcesses`, ожидающий сигнала, не будет исполняться до тех пор, пока не будет послано сообщение `signal`. Семафор возвращает управление стольким процессам, пославшим сообщение `wait`, сколько он получил сообщений `signal`. Если одному и тому же семафору посылают одно сообщение `signal` и два сообщения `wait`, он вернет управление только для первого сообщения `wait`. Когда семафор получает сообщение `wait`, для которого еще нет соответствующего сообщения `signal`, он останавливает тот процесс, из которого было послано сообщение `wait`.

Все остановленные данным семафором процессы ставятся в одну очередь. Приоритет процесса принимается во внимание только объектом `Processor` при планировании использования процессора. Каждый процесс, ожидающий сигнала на семафоре, будет переводиться в состояние `ready` неза-

висимо от его приоритета в соответствии с принципом «первый пришел — первым обслужили». Рассмотрим простой пример.

```
| sem |
sem := Semaphore new.
Processor fork: [ Transcript show: '1' ].
Processor fork: [ Transcript show: '2' .
    sem wait. Transcript show: '3' ]
at: 3.
Processor fork: [ Transcript show: '4' .
    sem signal. Transcript show: '5'; cr ]
at: 2.
```

Пример создает три новых процесса и семафор. Как результат, в окне Transcript появится строка 12435, которая получается следующим образом:

- сообщение `fork: [. . .]` создает процесс, который отображает '1';
- сообщение `fork: [. . .] at: 3` создает процесс, который отображает '2', а затем блокируется посланным семафору `sem` сообщением `wait`;
- сообщение `fork: [. . .] at: 2` создает процесс, который отображает '4' и посылает семафору `sem` сообщение `signal`;
- активизируется остановленный процесс как процесс с более высоким приоритетом; он отображает '3' и завершается;
- активизируется процесс с приоритетом 2, отображает '5' и завершается.

Одно из основных преимуществ создания независимых процессов состоит в том, что если некоторый процесс затребует в данное время недоступный для него ресурс, то, пока он ждет освобождения требуемого ресурса, выполняются другие процессы. Примерами объектов, которые могут потребоваться процессу, но могут быть доступны или недоступны, являются устройства ввода-вывода, события пользователя, совместно используемые структуры данных.

9.2.1. Прерывания

Семафоры нужны еще и для того, чтобы устанавливать связи между внешними событиями и процессами. Для этих целей в системе *Smalltalk Express* используется механизм прерываний. Примеры таких внешних событий — ввод с клавиатуры, перемещение мыши, импульсы сигнала времени. Можно расширить множество событий, вызывающих прерывание.

Модель прерываний системы *Smalltalk Express* соответствует типичной архитектуре компьютерных аппаратных прерываний. Прерывания явно допускаются и блокируются с помощью сообщения `enableInterrupts:`, посы-

лаемого классу `Process`. Блокировку прерываний надо использовать очень осторожно, поскольку в то время, когда прерывания заблокированы, смолотовская система не может отвечать на внешние события. Перед тем, как что-либо предпринять, предыдущее состояние прерываний нужно сохранить, окружая критичную часть кода следующим образом:

```
| oldState |
oldState:= Process enableInterrupts: false.
... "Блокировать прерывания и сохранить предыдущее состояние."
... критичный код ...
Process enableInterrupts: oldState.
... "Восстановить предыдущее состояние."
```

9.2.2. Пример: совместное использование набора

Рассмотрим пример, в котором семафоры используются для того, чтобы гарантировать право на безопасное использование одной и той же структуры данных разными процессами¹. Возьмем простую очередь — структуру данных, в которой работа с элементами данных происходит по правилу «первым пришел — первым ушел» (FIFO — first in, first out).

Чтобы построить модель очереди, можно воспользоваться экземпляром класса `OrderedCollection`, который запоминает свое содержимое в массиве `contents` и поддерживает в нем два индекса: `endPosition` и `startPosition`. Новый элемент нужно добавлять после элемента, расположенного по индексу `endPosition`, а удалять можно только существующий элемент, расположенный по индексу `startPosition`. Такая модель не гарантирует безопасное использование очереди. Проблемы с посылкой ей сообщений из различных процессов связаны с тем, что одновременно несколько процессов могут потребовать выполнить метод, например, для сообщений `removeFirst` или `addLast: anObject`. Напомним реализацию этих методов:

removeFirst

```
"Удаляет и возвращает последний элемент получателя.
Если набор пуст, сообщает об ошибке."
| answer |
startPosition > endPosition ifTrue: [^ self errorAbsentElement].
answer := contents at: startPosition.
contents at: startPosition put: nil.
startPosition := startPosition + 1.
^ answer.
```

¹ Пример взят из книги [5] и немного изменен.

addLast: anObject

```
"Добавляет аргумент anObject в конец получателя и возвращает
его как результат."
endPosition = contents size
    ifTrue: [self putSpaceAtEnd].
endPosition := endPosition + 1.
contents at: endPosition
    put: anObject.
^ anObject.
```

Предположим, что такому экземпляру класса `OrderedCollection` послали сообщение `removeFirst` из одного процесса, и как только удаляемый элемент найден (то есть выполнено выражение `answer := contents at: startPosition`) «проснулся» процесс с более высоким приоритетом и послал этому же экземпляру еще одно сообщение `removeFirst`. Так как индекс `startPosition` еще не увеличился, второе выполнение метода для сообщения `removeFirst` свяжет тот же самый элемент набора с переменной `answer`. Процесс с более высоким приоритетом удалит элемент из `contents`, увеличит на 1 значение переменной `startPosition` и возвратит удаленный элемент. Когда процесс с более низким приоритетом снова получит управление, индекс `startPosition` будет увеличен и произойдет удаление следующего элемента из массива `contents`. Он удаляется непрочитанным, а оба отработавших сообщения `removeFirst` возвращают один и тот же элемент.

Чтобы гарантировать безопасное использование набора, каждый процесс должен ждать его освобождения, посылая семафору сообщение `wait` перед обращением к набору, а освободив ресурс, посылать семафору сообщение `signal`, позволяя любому другому процессу использовать набор. Экземпляр определяемого ниже класса `SimpleSharedQueue` обеспечивает такое использование, и с ним можно работать из различных процессов. Переменная `accessProtect` ссылается на семафор, используемый для организации безопасного доступа к набору. Семафор должен начинать свою работу с одним лишним сообщением `signal`, чтобы первый обратившийся к набору процесс смог начать вычисления.

```
OrderedCollection subclass: #SimpleSharedQueue
    instanceVariableName: 'accessProtect'
    classVariableNames: ''
    poolDictionaries: '' !
```

```
!SimpleSharedQueue instance methods !
```

```
removeFirst
  | answer |
  accessProtect wait.
  answer:= super removeFirst.
  accessProtect signal.
  ^ answer!

addLast: anObject
  accessProtect wait.
  super addLast: anObject.
  accessProtect signal.
  ^ anObject!

initPositions: size
  super initPositions: size.
  accessProtect := Semaphore new.
  accessProtect signal! !
```

Напомним, что метод экземпляра `initPositions:` из класса `OrderedCollection` определяет значения переменных экземпляра. Поскольку класс `SimpleSharedQueue` добавляет новую переменную экземпляра `accessProtect`, необходимо переопределить метод `initPositions:`, определяя ее значение при создании нового экземпляра. Остальные унаследованные сообщения удаления и добавления элементов класс `SimpleSharedQueue` должен переопределить как `self invalidMessage`.

ГЛАВА 10

ГРАФИКА В *SMALLTALK EXPRESS*

Графические классы различных реализаций Смолтока сильно отличаются друг от друга: графические библиотеки концептуально не совпадают. Мы ограничимся описанием классов графики только системы *Smalltalk Express*. Примеры, написанные в *Smalltalk Express*, нельзя, к сожалению, выполнить в другой смолтоковской системе.

Графические возможности *Smalltalk Express* можно оценить, если посмотреть демонстрационные примеры, поставляемые вместе с системой, воспользовавшись в окне Transcript пунктом **GraphicsDemo** из меню **File**.

Графика *Smalltalk Express* формируется на основе графических возможностей среды MS Windows. Основа языка графики в Windows — интерфейс графических устройств (GDI — Graphics Device Interface, см., например, [36]). Система вызывает GDI-функции для того, чтобы с их помощью реализовать графические операции языка Смолток. Все GDI-функции вывода требуют контекста устройства вывода, который содержит текущие характеристики графической среды. Например, чтобы отобразить текст, *Smalltalk Express* вызывает GDI-функцию **TextOut**, одним из параметров которой является контекст устройства. Но при таком обращении явно не указываются шрифт, цвет текста, фоновый цвет и правила выравнивания текста, поскольку эти и другие характеристики — часть контекста устройства.

Каждый графический класс имеет соответствующий набор переменных и множество методов, реализующих требуемые графические операции. Если возникнет необходимость выйти за пределы графических возможностей ядра системы *Smalltalk Express*, следует обращаться к классам, реализующим низкоуровневый программный интерфейс с системой Windows и использовать функции интерфейса прикладного программирования (API — Application Program Interface). В принципе, *Smalltalk Express* позволяет вызвать любую функцию. Но, как правило, стандартных возможностей достаточно для решения большинства задач, поэтому только классами, реализующими эти возможности, мы и ограничимся.

Smalltalk Express поддерживает и растровую, и векторную графику. Рисунок в векторной графике задается как набор объектов, характеризующихся цветом и математическим описанием их контуров. Любая графическая форма может представляться и преобразовываться в соответствии с ее геометрическим описанием, не без учета разрешающей способности конкретного графического устройства. Эта особенность векторной графики и позволяет строить изображения, не зависящие от графического устройства.

Основываясь на идее независимости от устройства, система *Smalltalk Express* делает достаточно легким вывод графики в графическую среду, которой обычно является экран дисплея, но может быть и принтер, и графопостроитель и любое другое записывающее устройство.

Некоторые устройства вывода (графопостроитель, векторный дисплей) реализуют векторную графику непосредственно, например, непрерывно перемещая перо из позиции *A* в позицию *B*. Но большинство устройств вывода создают изображения, составленные из *растровых точек*: линия состоит из включаемых в нее точек на пути от *A* до *B*. Следовательно, образы векторной графики часто представляются *растровым* устройством вывода.

В *Smalltalk Express* можно работать и прямо с растровыми точками или *пикселями* (pixel)¹, из которых формируется *растр* или *битовая карта* (матрица пикселей). Растры представляются экземплярами класса *Bitmap*. Чтобы сослаться на конкретный элемент растра, используются точки — экземпляры класса *Point*. Для представления прямоугольных областей в графических операциях используются экземпляры класса *Rectangle* (Прямоугольник). Классы *Point* и *Rectangle* во всех реализациях похожи.

Модель графики в *Smalltalk Express* подобна той, которая обычно используется при рисовании пером на бумаге. В *Smalltalk Express* «графические инструменты» — экземпляры класса *GraphicsTool* — аналогичны перу: это нечто, способное создавать изображение, а «графические среды» — экземпляры классов *GraphicsMedium*, *Window* — аналогичны бумаге или холсту: это нечто, способное к отображению или сохранению графики. Реально это может быть экран монитора, окно, принтер, файл или часть памяти.

Графические устройства должны иметь систему координат, налагаемую на устройство, причем начало координат (0,0) может отображаться в любом месте устройства. По умолчанию, начало совпадает с верхним левым углом устройства. Переменная *x* возрастает при движении слева направо, а переменная *y* — при движении сверху вниз (см. рис. 10.1). Единицы измерения системы координат можно выбирать, возможный выбор зависит от контекста устройства. Как правило, это будет пиксел.

В этой главе мы рассмотрим все вышеперечисленные классы графики системы *Smalltalk Express* и приведем соответствующие примеры.

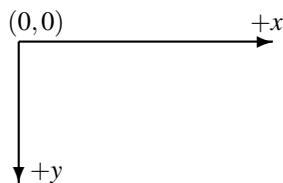


Рис. 10.1. Система координат в среде *Smalltalk Express*

¹ Это сокращение от английского выражения «picture element» — «элемент изображения».

10.1. Класс Point

Экземпляр класса `Point` имеет две переменные `x` и `y`, которые, как правило, обозначают положение пиксела в растре относительно левого верхнего угла растра (или другого заданного начала координат). Точка создается с помощью бинарного сообщения `@`, посылаемого числу. Аргументом сообщения также является число. Получатель задает `x`-координату точки, аргумент — `y`-координату. Например, результатом выражения `200 @ 150` будет точка с `x`- и `y`-координатами, равными соответственно 200 и 150.

Протокол экземпляра класса `Point` поддерживает следующие сообщения.

Класс `Point`

Протокол экземпляра

- `x` Возвращает `x`-координату получателя.
- `x: aNumber` Устанавливает `x`-координату получателя равной аргументу `aNumber`.
- `y` Возвращает `y`-координату получателя.
- `y: aNumber` Устанавливает `y`-координату получателя равной аргументу `aNumber`.
- `< aPoint` Возвращает `true`, если получатель лежит выше и левее аргумента `aPoint`, иначе возвращает `false`.
- `<= aPoint` Возвращает `true`, если получатель лежит не ниже и не правее аргумента `aPoint`, иначе возвращает `false`.
- `> aPoint` Возвращает `true`, если получатель лежит ниже и правее аргумента `aPoint`, иначе возвращает `false`.
- `>= aPoint` Возвращает `true`, если получатель лежит не выше и не левее аргумента `aPoint`, иначе возвращает `false`.
- `max: aPoint` Возвращает нижний правый угол прямоугольника, заданного получателем и аргументом `aPoint`.
- `min: aPoint` Возвращает верхний левый угол прямоугольника, заданного получателем и аргументом `aPoint`.
- `between: aPoint and: bPoint` Возвращает `true`, если получатель `>= aPoint` и получатель `<= bPoint`, иначе возвращает `false`.

Вот несколько примеров выражений, использующих этот протокол:

```
(10 @ 100) x           → 10
(10 @ 100) y           → 100
(10 @ 100) x: 50       → 50 @ 100
(10 @ 100) y: 50       → 10 @ 50
(45 @ 230) < (175 @ 270) → true
(45 @ 230) < (175 @ 200) → false
(45 @ 230) > (175 @ 200) → false
(175 @ 270) > (45 @ 230) → true
(45 @ 230) max: (175 @ 200) → 175 @ 230
1 @ 2 between: 0 @ 2 and: 2 @ 2 → true
```

Многие арифметические операции из класса **Number** переносятся на экземпляры класса **Point** как покоординатные операции. Их результатом является новый экземпляр класса **Point**, координаты которого получены применением указанной операции к соответствующим координатам точки, получившей сообщение, и точки, выступающей в качестве аргумента. Допускается также, чтобы в этих операциях в качестве аргумента использовалось число (экземпляр класса **Number**), которое в таких операциях рассматривается как точка с *x*- и *y*-координатами, равными заданному числу. Отсечение и округление, использующие те же имена сообщений, что и для класса **Number**, также включены в протокол экземпляра класса **Point**. Не станем приводить всего протокола, он достаточно прост и понятен почти без объяснений, а ограничимся несколькими примерами:

```
(45 @ 230) + (175 @ 300)    → 220 @ 530
(45 @ 230) + 5              → 50 @ 235
(45 @ 230) - (175 @ 300)   → -130 @ -70
(10 @ 20 ) * (3 @ 2)       → 30 @ 40
(160 @ 240) // 50          → 3 @ 4
(160 @ 240) // (50 @ 30)   → 3 @ 8
(160 @ 240) \\(50 @ 50)    → 10 @ 40
(-20 @ -30) abs            → 20 @ 30
(-2 @ 3) negated           → 2 @ -3
(2 @ 4) dotProduct: (5 @ 6) → 34  "Скалярное произведение"
(120.5 @ 220.7) rounded    → 121 @ 221
(120.5 @ 220.7) truncated  → 120 @ 220
(2 @ 4) transpose          → 4 @ 2
```

Если приведенного протокола недостаточно, его легко дополнить нужными методами. Например, можно определить метод, вычисляющий расстояние между двумя точками (внесите этот метод в класс **aPoint** своей системы):

distance: aPoint

"Возвращает расстояние между получателем и аргументом aPoint."

```
^ ((x - aPoint x) squared + (y - aPoint y) squared) sqrt
```

Тогда, выполняя выражение (0@0) distance: (3@4), получим 5.0.

10.2. Класс **Rectangle**

Прямоугольники представляют прямоугольные области пикселей. Как объект, каждый прямоугольник — экземпляр класса **Rectangle** и имеет две переменные экземпляра **leftTop** и **rightBottom**, которые задают две точки,

определяющие прямоугольник: первая переменная задает верхний левый угол прямоугольника (**origin**), вторая переменная задает нижний правый угол прямоугольника (**corner**). Ширина (**width**) и высота (**height**) прямоугольника могут быть найдены следующим образом:

width := **rightBottom** x – **leftTop** x.

height := **rightBottom** y – **leftTop** y.

Ширина и высота прямоугольника отражают количество пикселей в прямоугольной области по горизонтали и вертикали соответственно. Точка, задаваемая в виде **width @ height**, называется *размером* или *экстендом* (**extent**) прямоугольника. Размер прямоугольника можно вычислить следующим образом:

extent := **rightBottom** – **leftTop**.

Для поддержки графических операций точки и прямоугольники используются совместно. Поэтому создавать новый прямоугольник позволяют и протокол класса **Point**, и протокол класса **Rectangle**. В классе **Point** определены следующие два сообщения для создания прямоугольников:

Класс Point	Протокол экземпляра
--------------------	----------------------------

corner: **aPoint** Возвращает прямоугольник со значением **leftTop**, равным получателю, и значением **rightBottom**, равным аргументу **aPoint**.

extent: **aPoint** Возвращает прямоугольник со значением **leftTop**, равным получателю, и с шириной и высотой, равными координатам аргумента **aPoint**.

В качестве иллюстрации рассмотрим два выражения, создающих одинаковые прямоугольники:

1 @ 0 corner: 4 @ 3.

1 @ 0 extent: 3 @ 3.

Несколько сообщений для создания прямоугольников, использующих различные определяющие прямоугольник точки, содержит также класс **Rectangle**. Из этого обширного протокола приведем только два сообщения:

Класс Rectangle	Протокол класса
------------------------	------------------------

origin: **originPoint** **corner:** **cornerPoint** Возвращает прямоугольник с левым верхним и правым нижним углами, задаваемыми аргументами **originPoint** и **cornerPoint** соответственно.

origin: **originPoint** **extent:** **extentPoint** Возвращает прямоугольник с левой верхней точкой, равной аргументу **originPoint**, и с шириной и высотой, равными координатам точки **extentPoint**.

В качестве примера приведем еще два выражения, создающих прямоугольники, равные тем, которые созданы чуть выше с помощью сообщений из протокола класса `Point`:

```
Rectangle origin: 1@0 corner: 4@3.
```

```
Rectangle origin: 1@0 extent: 3@3.
```

Класс `Rectangle` поддерживает протокол для определения характерных точек прямоугольника.

Класс `Rectangle`

Протокол экземпляра

`bottom` Возвращает положение нижней стороны получателя (*y*-координату точки `rightBottom`).

`center` Возвращает точку, представляющую геометрический центр получателя.

`corner` Возвращает точку, представляющую правый нижний угол получателя.

`corner: aPoint` Изменяет в получателе точку, представляющую правый нижний угол, не изменяя при этом его размеров.

`extent` Возвращает размер получателя в виде точки `width @ height`.

`extent: aPoint` Изменяет размеры получателя в соответствии с аргументом, который рассматривается как `width @ height`; при этом положение левого верхнего угла прямоугольника не изменяется.

`height` Возвращает высоту получателя.

`height: anInteger` Изменяет высоту получателя, делая ее равной значению аргумента `anInteger`; при этом положение левого верхнего угла прямоугольника не изменяется.

`left` Возвращает положение левой стороны получателя (*x*-координату `leftTop`).

`origin` Возвращает точку, представляющую левый верхний угол получателя.

`origin: originPoint corner: cornerPoint` Изменяет левый верхний и правый нижний углы получателя, делая их равными аргументам `originPoint` и `cornerPoint` соответственно.

`origin: originPoint extent: extentPoint` Изменяет левый верхний угол получателя и его размеры, устанавливая левый верхний угол равным аргументу `originPoint`, а ширину и высоту равными координатам аргумента `extentPoint`.

`right` Возвращает положение правой стороны получателя.

`top` Возвращает положение верхней стороны получателя.

`width` Возвращает ширину получателя.

`width: anInteger` Изменяет ширину получателя, делая ее равной значению аргумента `anInteger`; при этом положение левого верхнего угла прямоугольника не изменяется.

Создадим три прямоугольника (общих переменных) `BoxA`, `BoxB`, `BoxC` и рассмотрим примеры отправки им различных сообщений.

```

BoxA := 50 @ 50 corner: 200 @ 200.
BoxB := 120 @ 120 corner: 260 @ 240.
BoxC := 100 @ 300 corner: 300 @ 400.
BoxA bottom    → 200 (y-координата точки rightBottom)
BoxB left      → 120 (x-координата точки leftTop)
BoxB right     → 260 (x-координата точки rightBottom)
BoxA center    → 125 @ 125 (геометрический центр)
BoxC width     → 200 (ширина прямоугольника)
BoxC height    → 100 (высота прямоугольника)
BoxC origin    → 100 @ 300 (верхний левый угол)
BoxC corner    → 300 @ 400 (нижний правый угол)

```

Следующий протокол класса `Rectangle` позволяет создавать новые прямоугольники, используя арифметические операции с точками, определяющими прямоугольник.

Класс `Rectangle`

Протокол экземпляра

```

expandBy: delta Возвращает прямоугольник, который больше получателя на аргумент delta (аргумент может быть прямоугольником, точкой или числом).
insetBy: delta Возвращает прямоугольник, который вложен в получатель с отступом от его границы на аргумент delta (аргумент может быть прямоугольником, точкой или числом).
intersect: aRectangle Возвращает прямоугольник, который является пересечением получателя и aRectangle.
nonIntersections: aRectangle Возвращает упорядоченный набор (экземпляр класса OrderedCollection) прямоугольников, объединение которых представляет область получателя, расположенную вне прямоугольника aRectangle.
merge: aRectangle Возвращает наименьший прямоугольник, содержащий и получателя, и аргумент aRectangle.

```

Вот примеры, использующие сообщения из этого протокола. Обратите внимания, что в системе *Smalltalk Express* прямоугольники печатаются в виде `originPoint rightBottom: cornerPoint`.

```

BoxC expandBy: 10 @ 20 → 90 @ 280 rightBottom: 310 @ 420
BoxC insetBy: 10 → 110 @ 310 rightBottom: 290 @ 390
BoxC insetBy: 10 @ 20 → 110 @ 320 rightBottom: 290 @ 380
BoxA intersect: BoxB → 120 @ 120 rightBottom: 200 @ 200
BoxB merge: BoxC → 100 @ 120 rightBottom: 300 @ 400
BoxA nonIntersections: BoxB → OrderedCollection(
    50 @ 50 rightBottom: 200 @ 120
    50 @ 120 rightBottom: 120 @ 200)

```

В классе *Rectangle* есть протокол проверки, который включает в себя сообщения, позволяющие определить, равны ли два прямоугольника (`=`), является ли получатель прямоугольником (`isRectangle`), содержится ли данная точка внутри границ прямоугольника (`containsPoint: aPoint`), пересекаются два прямоугольника или нет (`intersects: aRectangle`). Рассмотрим примеры с прямоугольниками `BoxA`, `BoxB`, `BoxC`, которые были определены выше.

```
$A isRectangle           → false
BoxA isRectangle        → true
BoxA=BoxA               → true
BoxC=BoxA               → false
BoxA containsPoint: 350 @ 150 → false
BoxC containsPoint: 200 @ 320 → true
BoxA intersects: BoxC   → false
BoxA intersects: BoxB   → true
```

Подобно координатам точки, координаты прямоугольника могут быть округлены (`rounded`) или усечены (`truncated`) до ближайшего целого числа. Прямоугольники можно перемещать и масштабировать. Из множества таких сообщений приведем только четыре, предлагая читателю с остальными познакомиться самостоятельно.

Класс *Rectangle*

Протокол экземпляра

`moveBy: aPoint` Сдвигает получатель на вектор `aPoint`.

`moveTo: aPoint` Сдвигает получатель так, чтобы его левый верхний угол совпал с точкой `aPoint`.

`scaleBy: scale` Возвращает прямоугольник, у которого переменные экземпляра равняются соответствующим переменным экземпляра получателя, умноженным на аргумент `scale`, являющийся точкой или числом.

`translateBy: delta` Возвращает прямоугольник, который сдвинут по отношению к получателю на аргумент `delta`, являющийся точкой или числом.

Применим эти сообщения к прямоугольникам `BoxA`, `BoxB` и `BoxC`. Обратите внимание, что в двух первых примерах происходит изменение структуры объекта: переменные экземпляра прямоугольника `BoxA` принимают новые значения. В остальных примерах создаются новые объекты (при этом прямоугольники `BoxB` и `BoxC` не изменяются).

```
BoxA moveBy: 50 @ 50      → 100 @ 100 rightBottom: 250 @ 250
BoxA moveTo: 200 @ 300   → 200 @ 300 rightBottom: 350 @ 450
BoxB scaleBy: 1/5        → 24 @ 24 rightBottom: 52 @ 48
BoxB scaleBy: 1@2        → 120 @ 240 rightBottom: 260 @ 480
BoxC translateBy: 100    → 200 @ 400 rightBottom: 400 @ 500
BoxC translateBy: 200 @ 300 → 300 @ 600 rightBottom: 500 @ 700
```

10.3. Графическая среда

Графические среды системы *Smalltalk Express* представляются абстрактным классом `GraphicsMedium` и его подклассами:

<code>GraphicsMedium</code>	ГрафическаяСреда
<code>Bitmap</code>	Растр
<code>Printer</code>	Принтер
<code>Screen</code>	Экран
<code>StoredPicture</code>	СохраняемыйРисунок

Класс `Bitmap` может иметь столько экземпляров, сколько необходимо. Классы `Printer` и `Screen` должны иметь по одному экземпляру для каждого реально существующего устройства. Например, класс `Printer` может иметь экземпляры для различных принтеров, установленных в системе. Глобальная переменная `Display` — экземпляр класса `Screen` для экрана монитора. Каждый экземпляр класса `StoredPicture` связывается с графическим метафайлом системы *Windows* (на диске или в оперативной памяти).

Класс `Window` и его многочисленные подклассы — также графические среды. Несмотря на то, что они не являются подклассами класса `GraphicsMedium`, все подклассы `Window` реализуют полный протокол класса `GraphicsMedium`. Класс `Window` был выделен в прямой подкласс класса `Object`, чтобы подчеркнуть фундаментальное значение объектов этого класса при программировании не только в задачах построения графических объектов, но и в задачах построения интерфейса пользователя, о которых мы подробно поговорим в четвертой части учебника. Экземпляр класса `Window` представляет окно в смысле системы MS Windows.

Каждый раз, когда создается новый экземпляр среды, создается и связывается со средой графический инструмент. Именно к этому графическому инструменту мы обращаемся, посылая графической среде сообщение `pen`:

```
Display pen.
Printer new pen.
(Bitmap width: 100 height: 100) pen.
```

10.3.1. Растры

Объект, который может содержать точечное графическое изображение или растр, — экземпляр класса `Bitmap`. Возможности растровой графики реализуются на основе массива битов, которые и хранят изображение.

Для растра, содержащего только черные и белые пиксели, каждый бит в растре соответствует пикселу. Для многоцветного растра нужно более

одного бита, чтобы представить на экране цветной пиксел. Есть два практических способа реализации цветных растров. Один из способов представления цвета в растре состоит в том, чтобы иметь множество битовых плоскостей. Каждая плоскость будет состоять из последовательных битов в памяти. Биты в той же самой позиции каждой плоскости воспринимаются вместе, формируя цвет пиксела в данной позиции. Таким образом, шестнадцатичетветный растр нуждается в четырех плоскостях. Другой способ реализации цветного растра состоит в представлении пиксела последовательными битами в памяти. В этом случае в шестнадцатичетветном растре каждый пиксел представляется четырьмя последовательными битами.

Выбор способа представления осуществляется операционной системой Windows в зависимости от видеоадаптера и его драйвера. Поэтому при программировании лучше ориентироваться на высокоуровневые графические операции, меньше зависящие от применяемого метода реализации цветов.

Растр размером 100×100 пикселов будет создан любым из следующих четырех выражений:

```
Bitmap width: 100 height: 100.  
Bitmap extent: 100 @ 100.  
Bitmap screenWidth: 100 height: 100.  
Bitmap screenExtent: 100 @ 100.
```

Первые два выражения создадут черно-белый растр. Следующие два выражения создадут растр, совместимый с экранным форматом. Все четыре выражения автоматически создадут перо (экземпляр класса Pen), связанное с экземпляром класса Bitmap, к которому всегда можно обратиться с помощью сообщения pen.

Рассмотрим несколько примеров.

```
| bitmap |  
bitmap := Bitmap screenExtent: 100 @ 100.  
bitmap pen home;  
fill: ClrRed; foreColor: ClrGreen;  
ellipse: 40 minor: 20; displayText: 'Hi'.  
bitmap displayAt: 10 @ 100 with: Display pen.  
bitmap release.
```

Здесь создается растр размером 100 на 100 пикселов с атрибутами используемой видеосистемы (Bitmap screenExtent: 100 @ 100). В этом растре пером в памяти создается рисунок, состоящий из эллипса и строки.

Созданному в памяти растру посылается сообщение displayAt:with:, пером экрана Display pen отображающее растр в точке 10 @ 100, с которой будет совпадать левый верхний угол растра. После чего (последняя строка метода) освобождается память, связанная с растром. Запомните, что память,

используемая под растры, принадлежит не *Smalltalk Express*, а *Windows*, и автоматически сборщиком мусора не освобождается. Ответственность за освобождение памяти растра всегда лежит на программисте.

Рассмотрим следующий пример:

```
| aBitmap |
aBitmap:= (Bitmap screenWidth: 100 height: 100).
aBitmap pen erase; place: 20 @ 20; box: 80 @ 80.
aBitmap displayAt: 0 @ 0 with: Display pen.
aBitmap release.
```

Здесь в памяти создается растр размером 100 на 100 пикселей, пером растра поле заполняется фоновым цветом экрана (сообщение `erase`), перо помещается в точку растра `20 @ 20` (сообщение `place: 20 @ 20`) и сообщение `box: 80 @ 80` рисует в поле растра контур прямоугольника `20 @ 20 corner: 80 @ 80`. Созданный растр пером экрана (`Display pen`) отображается в верхнем левом углу экрана, при этом левая верхняя точка растра помещается в точку экрана `0 @ 0` (сообщение `displayAt: 0@0 with: Display pen`). Последняя строка метода освобождает память, связанную с растром.

В классе `String` есть метод с именем `outputToPrinter`, который позволяет напечатать строку. Метод с таким же именем из класса `Bitmap` позволяет вывести на принтер созданный растр, рассматривая принтер системы как соответствующую графическую среду. Следующая программа напечатает на бумаге эллипс (имейте в виду, что начало координат `(0,0)` — верхний левый угол листа бумаги):

```
| bitmap |
bitmap := Bitmap extent: 100 @ 100.
bitmap pen home; ellipse: 40 minor: 20.
bitmap outputToPrinter.
bitmap release.
```

Растр всегда можно сохранить в файле (`outputToFile:`) с расширением `'bmp'`, а сохраненный ранее растр можно вновь загрузить в систему и использовать. Программа в следующем примере сначала создаст растр, затем запишет его в файл, после чего восстановит из файла в новый экземпляр класса `Bitmap` и выведет на экран:

```
| bitmap bitmap1 |
bitmap := Bitmap extent: 100 @ 100.
bitmap pen home; ellipse: 40 minor: 20.
bitmap outputToFile: 'c:\test.bmp'.
bitmap1 := Bitmap fromFile: 'c:\test.bmp'.
bitmap1 displayAt: 0@0 with: Display pen.
bitmap release. bitmap1 release.
```

Конечно, первые примеры — «плохие», поскольку прямой вывод на экран «портит» его внешний вид. Хотя объект `Display` — экземпляр класса `Screen` — делает существующий экран монитора доступным для графических операций и проявляет все графические возможности среды `GraphicsMedium`, он не предназначен для программирования графики. Объект `Display` нужен, прежде всего, для того, чтобы начать сеанс работы с системой *Smalltalk Express* и вести интерактивный диалог с пользователем. Графические операции следует выполнять внутри специальных окон (экземпляров класса `Window`), создавая внутри окна *графическую панель* (экземпляр класса `GraphPane`). Перо экрана (`Display pen`) используется тогда, когда необходимо переместить рисунок из одного окна в другое, поскольку перо окна не может рисовать вне своего окна, в то время как перо экрана рисует в любом месте экрана.

Продемонстрируем на примере перенос растра внутрь окна, добавляя небольшой кусочек «мерцающей» анимации. Сначала подготовим набор растров, которые поместим в глобальную переменную `Pictures`, чтобы им можно было воспользоваться позже в примере, связанном с анимацией.

```
| major minor |
Pictures := Array new: 8.
1 to: Pictures size do: [:i |
    Pictures at: i put: (Bitmap screenExtent: 100 @ 100).
    major := (i - 1 * 10) integerCos // 2.
    minor := (i - 1 * 10) integerSin // 2.
    (Pictures at: i) pen home;
        fill: ClrRed; foreColor: ClrGreen;
        ellipseFilled: major minor: minor].
```

Теперь приведем текст самого примера.

```
Window turtleWindow: 'Flexing Ellipses'.
10 timesRepeat: [ 1 to: Pictures size do: [:i |
    (Pictures at: i) displayAt: 10 @ 100 with: Turtle]].
"1 to: Pictures size do: [:i | (Pictures at: i) release]."
```

Рассмотрим по порядку, что происходит в этих двух кусках кода. Сначала создается массив `Pictures`, в котором будут содержаться восемь располагаемых в памяти растров, каждый из которых, используя собственное перо, создаст в растре рисунок, состоящий из зеленого контура эллипса и красного поля, размеры которого зависят от номера растра в массиве `Pictures`.

Затем создается окно с заголовком `'Flexing Ellipses'` ('Мерцающие эллипсы'), с графической панелью и графическим инструментом (пером панели) `Turtle` (все это делает сообщение `turtleWindow:`). В этом окне с помощью уже знакомого нам сообщения `displayAt:with:` растры последова-

тельно выводятся 10 раз подряд, создавая иллюзию «бьющегося сердца». В окне рисунок выполняется пером Turtle, а не пером экрана, показывая один за другим каждый из растров. Последнее выражение освобождает память, связанную с растрами. Мы его «закомментировали» по той причине, что его можно пока не выполнять, а выполнить только после того, как растры Pictures будут использованы в других примерах. Если кавычки вами удалены и эта строка выполняется, то перед тем, как выполнять пример, использующий переменную Pictures, придется снова выполнить выражения, инициализирующие эту переменную.

Перед тем как закрыть окно с именем 'Flexing Ellipses', сверните окно, а затем снова восстановите его на экране. Рисунок исчезнет. Не восстановится рисунок и после того, как вы перекроете это окно другим окном, а затем вытащите его наверх. Восстановления можно добиться, используя для этого «другие» перья, но об этом мы поговорим в разделе 10.4.3.

10.3.2. Класс StoredPicture

Класс **StoredPicture** используется для того, чтобы сохранять последовательность графических операций в метафайле Windows, дескриптор которого содержится в переменной экземпляра этого класса с именем **hMetaFile**. Метафайл — это не файл на диске, а инструмент фиксации действий программы по отношению к графическому контексту. Точнее, это особый графический контекст, в котором обращения к GDI не выполняют операций по созданию рисунка и записываются в сжатой, независимой от устройства форме. Потом можно выбрать один из действительных контекстов (окно, экран, принтер, ...) и послать метафайлу сообщение **play:**, выполняя в заданном контексте сохраненные обращения. В сущности, именно метафайл создает в Windows независимую от устройства графику. Почти вся созданная пером графика может сохраняться в метафайле. Экземпляр класса **StoredPicture** может, если это необходимо, сохранить метафайл в файле на диске и позже загрузить в систему для повторного использования или поместить его в буфер обмена. В OS/2 метафайл имеет расширение **met**, а в Windows — **wmf**. Следующая последовательность выражений показывает, как можно использовать экземпляр класса **StoredPicture**:

```
| meta meta1 |
meta := StoredPicture new.
meta create: nil.           "Создать метафайл."
meta pen place: 100 @ 100;
      box: 200 @ 200.
meta close.                "Закреть метафайл."
```

meta save: 'test.wmf'.	"Сохранить на диске."
meta release.	"Освободить ресурсы," "выделенные метафайлу."
meta1 := StoredPicture new.	
meta1 load: 'test.wmf'.	"Загрузить с диска."
Window turtleWindow: 'test'.	"Создать окно."
meta1 play: Turtle.	"Пользуясь пером Turtle," "вывести метафайл в окно."
meta1 release.	"Освободить ресурсы."

10.3.3. Принтер

При изучении растров уже отмечалось, что растры (и метафайлы тоже) можно печатать. Для этого предназначен класс `Printer`, который позволяет печатать на бумаге графику, используя те же самые сообщения, которые используются и в других графических средах. Таким образом, можно по-прежнему использовать перо (перо принтера, которое тоже является экземпляром класса `Pen`) для того, чтобы создать рисунок, а затем вывести его на принтер по правилу «что увидели бы в окне, то и получим на бумаге». Однако, так как физические характеристики и принципы действия дисплеев и принтеров различны, это правило выполняется не всегда. Для получения качественных документов следует вывод на принтер программировать отдельно, учитывая как его достоинства (прежде всего высокое разрешение), так и существенные ограничения.

Когда на принтер выводится графика, началом координат (0, 0) является левый верхний угол страницы. Обычно печатаемый объект выводится в очередь принтера, а связью между очередью и самим принтером управляет операционная система Windows.

Следующий пример использует ваш принтер по умолчанию так, как это определено в панели управления печатью графики системы Windows:

```
| printer |
printer:= Printer new.          "Создаем объект для принтера."
printer startPrintJob.         "Начинаем процесс печати."
printer pen
  place: 100 @ 100;
  mandala: 8 diameter: 90.
printer endPrintJob.           "Завершаем процесс печати."
```

К классу `Printer` мы еще вернемся после того, как рассмотрим классы графических инструментов и, в частности, класс `RecordingPen`.

10.4. Графические инструменты

Как видно из предыдущих примеров, рисовать можно только с помощью объектов, которые умеют создавать графические образы в графических средах. Таковыми являются экземпляры подклассов абстрактного класса `GraphicsTool`:

`GraphicsTool`

`TextTool`

`Pen`

`RecordingPen`

`Commander`

ГрафическийИнструмент

ТекстовыйИнструмент

Перо

ЗаписывающееПеро

НаборПерьев

Абстрактный класс `GraphicsTool` содержит переменные и методы, общие для всех графических инструментов.

Каждый подкласс `GraphicsTool` расширяет его функциональные возможности. Экземпляр класса `TextTool` работает подобно пишущей машинке и отображает только символы. Экземпляр класса `Pen` способен не только печатать, но и рисовать. Экземпляр класса `RecordingPen` имеет все возможности классов `TextTool` и `Pen`, но способен запоминать графические операции и последовательно выполнять их позже. Экземпляр класса `Commander` управляет несколькими перьями.

К графическому инструменту (перу) графической среды обращаются, посылая графической среде сообщение `pen`. Когда перу посылают некоторое рисующее сообщение, то результат отображается в связанной с пером графической среде. При желании можно разорвать связь между пером и его текущей графической средой и связать перо с другой средой. Следовательно, можно сначала связать перо с окном, создать нужное изображение, а потом связать перо, например, с принтером, чтобы создать окончательную твердую копию изображения.

Экземпляр класса `GraphicsTool` имеет доступ к пулам `WinConstants`, `ColorConstants` и к следующим переменным:

`deviceContext` — дескриптор контекста устройства, связанного с графическим инструментом;

`graphicsMedium` — экземпляр класса `GraphicsMedium`, связанный с графическим инструментом;

`width` — ширина страницы среды; для окна — ширина окна, для принтера — ширина листа бумаги;

`height` — высота страницы среды; для окна — высота окна, для принтера — высота листа бумаги;

`foreColor` — цвет, используемый для рисования;

`BackColor` — цвет фона (заливки);

`location` — текущая позиция, занимаемая графическим инструментом в связанной с ним графической среде;

`logicalTool` — дескриптор графического инструмента.

Посылая экземпляру одного из подклассов класса `GraphicsTool` сообщение из протокола доступа, селектор которого совпадает с именем переменной, всегда можно получить значение этой переменной. Можно и переопределить значение любой переменной, посылая экземпляру ключевое сообщение, представляющее собой имя переменной с конечным двоеточием (кроме переменной `deviceContext`, значение которой устанавливается посредством сообщения `handle`:).

Вся реальная работа по созданию графического образа выполняется GDI-функциями системы Windows, которые скрыты в методах, написанных на языке Смолток так, что пользователь может не заботиться о технических тонкостях операций. Наиболее мощная и эффективная из функций GDI — это, по-видимому, функция `BitBlt`, выполняющая операции переноса блока битов. В классе `GraphicsTool` функция `Bitblt` скрыта в методах `copy` (скопировать) и `fill` (заполнить). Например, выражение

```
Display pen copy: Display pen
    from: (0 @ 0 extent: 100 @ 100)
    to: (200 @ 150 extent: 200 @ 200)
```

скопирует часть экрана из прямоугольника `0@0 extent: 100@100` на экран в прямоугольник `200 @ 150 extent: 200@200`, используя функцию `BitBlt`. В этом примере прямоугольник-адресат больше прямоугольника-источника, поэтому прямоугольник-источник будет автоматически отмасштабирован до размеров адресата.

При выполнении выражения

```
Display pen place: 100 @ 100; ellipseFilled: 100 minor: 50
```

будет нарисован эллипс с центром в точке `100@100` с указанными размерами, заполненный цветом, хранящимся в переменной `BackColor`². Конечно, цвет изображения и фоновый цвет, как и в предыдущих примерах, можно установить с помощью сообщений `foreColor: aColor` и `BackColor: aColor`. До сих пор мы не задавались вопросом, что представляет собой аргумент подобных сообщений. Это имя цвета — константа из пула `ColorConstants` (чтобы с ними познакомиться, выполните, например, в рабочем окне выражение `ColorConstants inspect`). Имена всех таких констант начинаются с префикса `Clr`.

² По умолчанию — `nil`, системный цвет фона. Он же задается константой `Clr-Background` и зависит от выбранной цветовой схемы Windows.

Вообще говоря, в сообщениях `foreColor:` и `backColor:` аргументом должно быть 32-битовое целое число, позволяющее указывать как цвета из системной палитры (в безпалитровых видеорежимах — из цветовой схемы *Windows*), так и цвет в цветовой модели RGB. Число, содержащее RGB-значение, можно сформировать, посылая классу `GraphicsTool` сообщение `red:green:blue:` (красный:зеленый:синий:), где каждый аргумент — целое число от 0 до 255, описывающее «величину» соответствующей составляющей в создаваемом цвете. В константах из пула `ColorConstants` хранятся именно числа. Подробности об управлении цветами и использовании палитры см. в [36, с. 104–112], а мы ограничимся для пояснения двумя примерами, в одном из которых создадим RGB-значение. Выполним выражение

```
Display pen backColor: ClrRed;
      place: 100 @ 100; ellipseFilled: 100 minor: 50;
      backColor: nil. "Восстановить значение по умолчанию."
```

Мы получим эллипс из предыдущего примера, но красного цвета. Тот же результат получим при выполнении выражения

```
Display pen backColor: (GraphicsTool red: 255 green: 0 blue: 0);
      place: 100 @ 100; ellipseFilled: 100 minor: 50;
      backColor: nil.
```

10.4.1. Класс `TextTool`

Экземпляр класса `TextTool`, в дополнение к тому, что он наследует из класса `GraphicsTool` методы, позволяющие ему производить заполнение областей и перемещение блоков, еще может по переданной ему строке рисовать в графической среде *глифы* (изображения символов) из шрифтов, имеющих в *Windows*. Экземпляр `TextTool` обычно используется совместно с окнами, содержащими экземпляры класса `TextPane`, в которых невозможна графика. Наследуемая переменная `location` используется экземпляром `TextTool` как позиция первого глифа выводимой строки. Поэтому всегда можно послать сообщение, меняющее ее, или сделать запрос о текущей позиции размещения символа.

Следующий код открывает окно (экземпляр класса `TextWindow`) с заголовком `'TextTool Examples'`, которое содержит *текстовую панель* (объект, представляющий собой часть окна, специально предназначенного для вывода текста) и использует связанный с этой панелью экземпляр класса `TextTool` для того, чтобы в разных точках окна отобразить строки:

```
| aWindow textTool |
aWindow := TextWindow windowLabeled: 'TextTool Examples'
      frame: (100@100 extent: 400@200).    "Создать окно."
```

```

aWindow nextPutAll: 'Hi!'; cr.
textTool := aWindow pane pen.
textTool place: 10@10;
  displayText: 'Welcome';
  displayText: 'to' at: textTool location + (50@30);
  centerText: 'Smalltalk' at: textTool extent // 2;
  lineDisplay: 'Windows'
    at: (textTool width - 40) @ (textTool height - 20).

```

Перед тем как закрыть окно с именем 'TextTool Examples', сверните окно, а затем снова восстановите его на экране. Обратите внимание, что восстановится только строка 'Hi'. Эта строка единственная, записанная методом `nextPutAll:`, который добавляет аргумент сообщения так, чтобы поддерживать состояние текста в панели, когда в ней происходят некоторые события (в данном случае — событие `getContents`, происходящее всегда, когда окно отображается). Остальной текст был записан в панель пером текстовой панели — экземпляром класса `TextTool` — и не восстановился, поскольку такое перо не поддерживает операции восстановления созданного ранее состояния панели.

К тем переменным, которые класс `TextTool` наследует от суперкласса `GraphicsTool`, он добавляет собственную переменную экземпляра `font`. Переменная `font` хранит описание любого доступного шрифта и используется для вывода глифов в связанный с инструментом экземпляр класса `GraphicsMedium`. В качестве примера использования переменной `font` рассмотрим следующий код:

```

| font aWindow textTool |
aWindow := TextWindow windowLabeled: 'TextTool Examples'
  frame: (100 @ 100 extent: 400 @ 200).
textTool := aWindow pane pen.      "Получить доступ к перу."
font := Font chooseAFont: 'Выберите шрифт, пожалуйста.'.
font isNil ifTrue: [^ self].
textTool font: font;                "Установить шрифт."
  foreColor: ClrLightgray;
  displayText: 'Hello, World!'
    at: (60 @ (textTool height // 2));
  foreColor: ClrBlack;
  displayText: 'Hello, World!'
    at: (60 @ (textTool height // 2 + font height)).

```

Во время выполнения этого кода возникнет диалоговое окно выбора шрифта с заголовком 'Выберите шрифт, пожалуйста.'. Выберите шрифт и нажмите на кнопку ОК.

Несколько слов о шрифтах

Smalltalk Express представляет символы в строках, используя текущую кодовую страницу Windows. Чтобы отображать символы в графической среде, их коды должны преобразовываться в графические образы (глифы). Класс `TextTool` выполняет преобразование, вызывая GDI-функции Windows, а класс `Font` предоставляет информацию о том, какой шрифт используется в этом преобразовании. Сами шрифты обеспечиваются системой Windows. Табл. 10.1 перечисляет глобальные переменные системы, хранящие шрифты, используемые различными объектами. Воспользовавшись пунктом `Fonts...` системного меню, можно изменить значения первых двух переменных, после чего все открываемые окна с текстовыми и списковыми панелями будут использовать новый шрифт.

Таблица 10.1. Стандартные шрифты *Smalltalk Express*

Глобальная переменная	Какими объектами используется
<code>ListFont</code>	ListPane (СписковаяПанель)
<code>TextFont</code>	TextPane (ТекстоваяПанель)
<code>SysFont</code>	все другие

Приведем наиболее часто используемые методы класса из класса `Font`:

Класс Font Протокол класса

`allFonts` Возвращает массив всех доступных шрифтов.

`chooseAFont: aTitleString` Открывает диалоговое окно, которое позволяет пользователю выбрать шрифт; возвращает экземпляр класса `Font` со шрифтом, уже доступным для работы; `aTitleString` отображается как заголовок диалогового окна.

`new` Создает новый объект, представляющий шрифт, который хранит атрибуты шрифта графической оконной системы. Этот шрифт еще не доступен для работы, но станет доступным при послышке созданному экземпляру сообщения `makeFont` или `fontHandle`.

После создания нового шрифта его необходимо настроить, устанавливая его многочисленные атрибуты. Приведем только сообщения установки используемых в Windows атрибутов шрифта. Посылая экземпляру класса `Font` сообщения с именами без двоеточия, можно узнать значение соответствующего атрибута.

Класс Font Протокол экземпляра

`bold: aBoolean` Устанавливает полужирный шрифт, когда аргумент равен `true`.

`italic: aBoolean` Устанавливает курсивный шрифт, когда аргумент равен `true`.

charSize: **aPoint** Устанавливает размер шрифта: *x*-координата точки **aPoint** задает среднюю ширину символов шрифта, *y*-координата задает наибольшую высоту шрифта.

faceName: **aString** Устанавливает гарнитуру шрифта по ее имени; здесь требуется указать имя гарнитуры шрифта, установленной в системе или включенной в список подстановок шрифтов.

Как сказано выше, экземпляр класса **Font** может использоваться только после отправки ему сообщения **makeFont**, позволяя сделать данный шрифт (или наиболее схожий с ним шрифт) доступным в графической среде. Как обычно, выделенные ресурсы после их использования надо вернуть. Это делает метод **deleteFont**. Он освобождает ресурсы Windows, запрошенные для шрифта, но *не удаляет* экземпляр класса **Font**, который является обычным смолтоковским объектом. Поэтому, если какой-то шрифт нужен во многих операциях вывода, имеет смысл создать глобальную переменную с экземпляром класса **Font**, а доступ к самому шрифту получать только на время операций вывода в конкретной графической среде.

10.4.2. Класс **Pen**

Класс **Pen** наследует все возможности класса **TextTool**, а для рисования обеспечивает интерфейс «черепаший графика»: рисунок создается так, как будто идет черепаха, оставляя след «вымазанным чернилами» хвостом.

Помимо позиции (в наследуемой переменной **location**) и ширины пера (в наследуемой переменной **width**), экземпляр класса **Pen** хранит направление движения в переменной **direction** и состояние пера в переменной **downState**.

Переменная **location** сообщает перу, с какой позиции начать следующее перемещение. Когда перо создается, оно помещается в центр нового окна (что равносильно отправке перу сообщения **home**). Направление движения (переменная **direction**) — целое число от 0 до 359 градусов, с отсчетом по часовой стрелке; при этом направление на «восток» (к правому краю экрана) соответствует 0°, а на «юг» (к нижнему краю экрана) — 90°. По умолчанию перо устанавливается на «север» — 270°. Пользуясь переменной **direction**, перо вычисляет конечную точку перемещения, когда посылается сообщение **go**; в котором определяется только величина перемещения (единицей измерения всегда является пиксел). Если значение переменной **downState** равно **true**, то во время своего перемещения перо рисует (перо опущено и оставляет след); иначе оно перемещается, но не рисует (перо поднято). Состояние пера всегда можно установить с помощью следующих выражений: **aPen down** — перо опустить, **aPen up** — перо поднять.

Ширина пера (**width**) — это ширина кончика пера (маски) или, другими словами, ширина рисуемой линии. Чтобы установить ширину пера, можно использовать сообщения `setLineWidth:`, `defaultNib:`. Например, нарисуем в окне замкнутый контур из прямолинейных отрезков:

```
Window turtleWindow: 'Pen Examples'.
```

```
Turtle goto: 30 @ 30;
```

```
  box: Turtle extent — 30;
```

```
"Нарисовать прямоугольник."
```

```
  home;
```

```
"Перо — в центр панели."
```

```
  defaultNib: 4.
```

```
"Ширина пера — 4 пиксела."
```

```
10 timesRepeat: [
```

```
  4 timesRepeat:
```

```
    [Turtle go: 100; turn: 90].
```

```
    Turtle turn: 36 ].
```

```
Turtle defaultNib: 1
```

Здесь глобальная переменная `Turtle` — это перо графической панели окна, созданного в ответ на посылку сообщения `turtleWindow:` классу `Window`.

Можно рисовать не только прямоугольники и отрезки, но и эллипсы, круги, хорды, секторы. Код следующих выражений демонстрирует некоторые из возможностей, выполняя операции не в окне (как в предыдущем примере), а непосредственно на экране:

```
Display pen
```

```
  place: Display extent // 2;
```

```
  boxOfSize: 100 @ 100;
```

```
  circle: 100;
```

```
  ellipse: 100 minor: 50;
```

```
  chord: 200 minor: 100 angles: 0 @ 90;
```

```
  pie: 150 minor: 150 angles: 0 @ 135.
```

В предыдущих двух примерах рисовались только контуры. Приведем примеры сообщений, поддерживающих цвет заполнения и окраску линий.

```
Window turtleWindow: 'Pen Examples'.
```

```
Turtle fill: ClrYellow;
```

```
  home;
```

```
  foreColor: ClrDarkgray;
```

```
  defaultNib: 4;
```

```
  boxOfSize: 100 @ 100;
```

```
  home.
```

Обратите внимание, что класс `Pen` наследует из класса `TextTool` возможности по отображению текста. Но если свернуть, а затем восстановить окно 'Pen Examples', то, как и в случае с текстом, рисунок не восстановится.

Цветом заполнения области можно распорядиться и по-другому. GDI-функции, реально выполняющие графические операции, используют текущую кисть для заполнения внутренних областей замкнутых фигур: эллипсов, прямоугольников и т.д. Есть семь предопределенных кистей, которые могут быть выбраны в контексте устройства: `GrayBrush`, `LtgrayBrush`, `DkgrayBrush`, `BlackBrush`, `WhiteBrush`, `NullBrush`, `HollowBrush` (но можно создавать и собственные однородные или штриховые кисти). В следующем примере, чтобы заполнить внутреннюю часть прямоугольника, выбирается серая кисть.

```
| aPen |
```

```
aPen := Display pen.
```

```
aPen selectStockObject: GrayBrush; place: 0 @ 0; boxFilled: 100 @ 100.
```

10.4.3. Класс `RecordingPen`

Чтобы создать восстанавливаемое изображение, сравнимое с восстанавливаемым текстом в текстовых панелях, надо использовать экземпляры класса `RecordingPen` (ЗаписывающееПеро). Этот класс реализует все возможности своих суперклассов, но вводит новые переменные и методы для регистрации графики, что позволяет записывать рисунки в графический сегмент и позднее его воспроизводить. Графический сегмент — экземпляр класса `StoredPicture` (см. с. 172). Есть три способа работы с этим пером:

```
aPenRecorder retainPicture: aBlock.
```

```
aPenRecorder drawPicture: aBlock.
```

```
aPenRecorder drawRetainPicture: aBlock.
```

Каждое из этих выражений выполнит блок `aBlock`, содержащий графические операции. Метод `retainPicture`: только записывает изображение, метод `drawPicture`: только рисует изображение, но не записывает его, а метод `drawRetainPicture`: и создает, и записывает изображение.

С графической панелью в системе *Smalltalk Express* связывается именно экземпляр класса `RecordingPen`. Поэтому приложение может создавать графический образ, и всякий раз, когда графическую панель нужно повторно отобразить на экране, сегмент воспроизводится, восстанавливая содержимое графической панели. Когда графическая панель открывается первый раз, ее перо (экземпляр класса `RecordingPen`) создает начальный графический сегмент и выполняет метод, определяемый селектором, связанным с событием `#getContents`. Графика, создаваемая этим методом, регистрируется в начальном сегменте, позже приложение может добавить сюда любое количество сегментов. Когда графическая панель получает сообщение `WmPaint`, она воспроизводит все сегменты, принадлежащие ее перу.

Есть другой способ отображения содержимого графической панели. Он состоит в том, чтобы выполнять метод, определяемый селектором, связанным с событием `display`. Этот метод не использует регистрируемую графику. Метод, связанный с событием `#display` выполняется тогда, когда графическая панель вновь должна отображаться на экране. Приложение сможет отвечать на событие `#display`, если в метод из класса приложения, открывающий окно, добавить строчку `when: #display perform: #draw:`. Все это используется в примерах `GraphicsDemo`, `Dashboard`, `Puzzle15`, `FreeDrawing`, поставляемых с системой. Панели, события, сообщения и технику построения приложений мы подробно рассмотрим в четвертой части. А пока на примерах продемонстрируем некоторые из возможностей класса `RecordingPen`. В первом примере

```
Window turtleWindow: 'RecordingPen Examples'.
Turtle drawRetainPicture: [Turtle fill: ClrYellow;
                           home;
                           mandala: 16 diameter: 250]
```

в центре желтого окна будет нарисована 16-угольная мандала, а рисунок будет сохранен в сегменте, обеспечивая возможность его восстановления. Если теперь изменить размеры этого окна, свернуть, а потом восстановить его, перекрыть другим окном, а потом вновь сделать активным, первоначальное состояние окна восстановится.

В следующем примере дважды открывается графический сегмент, в сегменте создается графический образ и сегмент закрывается, тем самым графический образ создается в окне и запоминается. Всегда можно создать цепочку таких сегментов, а затем всю ее повторно воспроизвести. Поскольку используется экземпляр класса `RecordingPen`, записывающий графику, его можно использовать и для вывода графики на доступный по умолчанию принтер, выполняя выражение вида `Printer printWith: aPenRecorder`. Такое выражение есть в нашем примере, поэтому, выполняя его, не забудьте включить принтер.

```
Window turtleWindow: 'Test 1'.
Turtle openSegment;           "Начать новый сегмент."
  place: 0 @ 0;              "Нарисовать в сегменте квадрат."
  box: 100 @ 100;
  closeSegment.             "Закрыть первый сегмент."
Turtle openSegment;           "Начать новый сегмент."
  place: 100 @ 100;         "Нарисовать в сегменте круг."
  circle: 50;
  closeSegment.             "Закрыть второй сегмент."
Printer printWith: Turtle.   "Содержимое панели напечатать."
```

Заметим, что каждый сегмент имеет индекс, который можно использовать для отображения только этого сегмента, посылая перу сообщение `drawSegment: index`. В примере создано два сегмента с индексами 1 и 2.

10.4.4. Класс **Commander**

Класс **Commander** — подкласс класса **RecordingPen**. Экземпляр класса **Commander** состоит из массива перьев (экземпляров класса **RecordingPen**). Класс **Commander** предоставляет интерфейс для одновременного рисования всеми перьями: он переопределяет методы, связанные с сообщениями `place:`, `turn:`, `down`, `up`, `go:` и `goto:`, чтобы передавать соответствующие сообщения всем перьям экземпляра. В следующем примере рисуются пять расположенных веером драконовых кривых:

```
Window turtleWindow: 'Commander Dragons'.
(Commander pen: 5           "Создать экземпляр Commander."
  forDC: Turtle handle medium: Turtle graphicsMedium)
  up; home; fanOut;         "Установить перья веером."
  go: 60; down; dragon: 9   "Нарисовать драконовы кривые."
```

10.5. Работа с курсорами

Чтобы визуально указать на состояние системы, Смолток использует различные формы курсора. Например, форма курсора в виде песочных часов используется для того, чтобы указать на то, что машина «занята». Формы курсора — хороший способ передачи информации о приложениях.

Курсоры системы *Smalltalk Express* управляются классом **CursorManager**, который обеспечивает интерфейс между системой и мышью. Обычно, когда что-либо отображается поверх курсора, сначала нужно скрыть курсор; иначе курсор может вмешаться в происходящее. Чтобы упростить эту проблему, все примитивы системы, которые изменяют текущее состояние экрана, сначала скрывают курсор, а после сделанных изменений восстанавливают его. Выражение `Cursor hide` скрывает курсор, а `Cursor display` — отображает его. Эти два сообщения работают подобно круглым скобкам: они должны быть сбалансированы. Например, предположим, что курсор в настоящее время отображается; если его дважды скрыть, а затем один раз отобразить, курсор не будет виден, необходимо отобразить его еще раз. Если курсор не отображается и нет уверенности, в каком состоянии он находится в настоящее время, следует выполнить выражение `Cursor reset`. После этого восстановится равновесие скрывающих и отображающих сообщений, и курсор появится на экране.

Smalltalk Express включает несколько форм курсора (см. табл. 10.2), которые хранятся в пуле `CursorConstants`. Доступ к курсорам осуществляется с помощью сообщений, посылаемых классу `CursorManager`.

Таблица 10.2. Стандартные курсоры *Smalltalk Express*

Сообщение	Форма курсора
<code>arrow</code>	стрелка на северо-запад
<code>crossHair</code>	курсор-перекрестие
<code>execute</code>	песочные часы
<code>normal</code>	стрелка на северо-запад
<code>origin</code>	двунаправленная стрелка на юго-запад
<code>text</code>	карет (курсор, похожий на букву I)

Чтобы изменить форму курсора, надо просто послать курсору сообщение `change`, меняющее текущий курсор на курсор, получивший это сообщение. Например, выражение

```
CursorManager execute change
```

изменяет форму курсора на песочные часы (что на языке курсора означает «ждите, ведутся вычисления»). Если форму курсора надо изменить временно, например, поменять ее на форму `origin` только на время выполнения блока `aBlock`, можно выполнить выражение

```
CursorManager origin changeFor: aBlock.
```

Любую форму курсора, определяемую операционной системой, можно сделать доступной и в *Smalltalk Express*. Для этого сначала надо выполнить код, подобный следующему:

```
CursorConstants at: 'Cross'
put: (CursorManager new
handle: (CursorManager getWinCursor: ldcCross))
```

Затем создать метод класса в классе `CursorManager`:

cross

```
#addedByOSI. "Указание на то, что ресурс добавляется ОС."
^ CursorConstants at: 'Cross'.
```

Теперь в *Smalltalk Express* можно воспользоваться новым курсором `cross`, выполняя выражение

```
CursorManager cross change.
```

Имя ресурса `ldcCross` определяется в пуле `WinConstants`. Там же находятся имена и нескольких других курсоров `Windows`, идентифицированных приставкой `ldc`. Форму курсора можно всегда вывести на экран с помощью выражения вида

```
(CursorManager cross) displayAt: 200 @ 200 with: Display pen.
```

ЧАСТЬ III

ПОСТРОЕНИЕ НОВЫХ КЛАССОВ

Мы живем, точно в сне неразгаданном,
На одной из удобных планет . . .
Много есть, чего вовсе не надо нам,
А того, что нам хочется, нет . . .

Игорь Северянин

ГЛАВА II

ОСНОВЫ СТРОИТЕЛЬСТВА

Цель, которую мы будем преследовать не только в этой главе, но и во всей этой части, двоякая. Во-первых, мы создадим несколько новых классов, расширяя тем самым возможности системы. Среди этих классов будут такие, которые уточняют, специализируют поведение уже существующих классов, и те, которые определяют совершенно новые для системы объекты. Такая классификация имеет значение только для нас; для системы никаких различий между классами не существует. Начнем с объектов, довольно простых по своей структуре и понятных по поведению, а затем приведем более сложные примеры. По ходу дела мы будем решать и вторую задачу: продолжим знакомство с библиотекой классов, поставляемых с системой, и с протоколами этих классов.

11.1. Принципы построения нового класса

Напомним, что построение нового класса позволяет создавать новые, ранее отсутствовавшие в системе объекты. Разумеется, эти объекты нужны не сами по себе, а только потому, что они «решают» некоторую полезную для нас задачу. Можно сказать и по-другому. Когда необходимо решить некоторую задачу, в процедурном языке программирования мы создаем программу, которая, обрабатывая исходные данные, создает интере-

сующие нас результаты. В объектно-ориентированном языке программирования для тех же целей мы определяем класс или классы, позволяющие создавать объекты со структурой, хранящей данные, и поведением, которое проявляется в умении объекта обрабатывать эти данные, тем самым решая стоящие перед нами задачи. В отличие от процедурного языка, здесь данные и программы, обрабатывающие эти данные, существуют в рамках одного объекта.

Сам процесс разработки класса носит итеративный, а не строго последовательный характер. Такой характер программирования отвергает некоторые распространенные методологии разработки программ. Например, в процедурных языках часто используют *каскадный подход*. Программа полностью проектируется сверху вниз, а затем реализуется по восходящей — снизу вверх. При этом предполагается, что одна фаза процесса проектирования или разработки завершается прежде, чем начинается следующая. Для такого подхода есть серьезные основания — очень дорого стоит исправление допущенной ошибки.

Приемы, полезные при каскадном подходе, не могут использоваться при итеративной разработке, поскольку в этом случае периоды анализа решаемой задачи могут сопровождаться периодами проектирования, которые, в свою очередь, могут сопровождаться созданием некоторого кода, прежде чем будет предпринято дальнейшее проектирование или работа по анализу ситуации. Поэтому итеративная разработка (или, как часто говорят, быстрое прототипирование) является специфической возможностью программирования на объектно-ориентированном языке и, как показывает практика, позволяет создавать работающие программы значительно быстрее, чем на любом процедурном языке программирования.

Еще одно преимущество языка Смолток состоит в возможности многократного применения созданного ранее кода из уже существующей иерархии классов. Поэтому, создавая новый класс, надо помнить, что он создается не на пустом месте, а в рамках уже существующей системы, и определять классы, совместимые с существующей библиотекой классов, тем самым, расширяя возможности системы.

Когда разрабатывается программа на процедурном языке, обычно задаются вопросы, подобные следующим: «Какие процедуры нужны?», «Что они должны возвращать?», «Что должно передаваться в качестве параметра?». Когда разрабатываются новые классы, дополнительно надо задать и такие вопросы: «В каком классе должен быть этот метод?», «Нужно ли создать подкласс или разумнее все реализовать в одном классе?», «Какое место должен занимать создаваемый класс в иерархии?». Отвечая на эти и многие другие вопросы, надо придерживаться указанных ниже правил.

Рассматривать интерфейс отдельно от реализации. Интерфейс объекта составляет набор функциональных возможностей, предлагаемый им для использования. В системе Смолток — это набор методов, определенных программистом в созданном классе и наследуемых из суперклассов. Важно, чтобы этот интерфейс (или протокол) рассматривался отдельно от способов его реализации.

Стараться скрыть сложность. Это, в сущности, предыдущее правило, но сказанное другими словами. Если можно представить простой, общий интерфейс для класса и с его помощью скрыть сложные моменты в его реализации, стоит сделать именно так. Надо всегда стремиться сделать создаваемый класс более легким в использовании.

Стараться минимизировать зависимости между классами. Многие проблемы будут решаться значительно проще, если классы как можно меньше осведомлены друг о друге. Тогда, например, можно изменять один из классов, абсолютно не воздействуя на другие. Особенно важно подчеркнуть, что классы не должны ничего знать о реализации друг друга. Конечно, иногда будут нужны группы классов, которые хорошо «осведомлены» друг о друге. В этом случае подумайте, нет ли возможности построить такую группу в виде отдельной иерархии (посмотрите на класс `Collection`).

Тщательно продумывать структуру класса и его экземпляра. Хорошо продуманный (спроектированный) класс обладает рядом преимуществ: высокой надежностью, хорошими перспективами многократного использования созданного кода, более быстрой и эффективной реализацией, более широкими возможностями дальнейшего расширения. Хотя эти характеристики могут оказаться взаимно исключаящими (например, более быстрая реализация может подразумевать меньшие возможности многократного использования кода), поэтому всегда полезно знать, что в каждом конкретном случае важнее.

Выносить общий код за пределы сложных алгоритмов. Точно так же, как и в процедурном программировании, надо стремиться разбивать сложные алгоритмы на логически независимые части. Они не только будут проще, их будет легче проверять и отлаживать. Но это позволит создать куски кода, которые затем смогут многократно использоваться совершенно независимо. Однако, принимая во внимание предыдущие правила, будьте внимательны в вопросах распределения сложного алгоритма по нескольким классам. Бездумное следование последнему правилу может сделать такие классы зависимыми от информации о реализации друг друга.

Не создавать сложных (структурированных) переменных. Не «кодируйте» несколько характеристик состояния объекта в одной переменной.

Такая практика станет препятствием и в будущих разработках, и при многократном использовании кода. Используйте отдельную переменную для каждой «*атомарной*» характеристики состояния объекта.

Всегда помнить о значении класса для системы. Об этом мы уже упоминали, но повторим еще раз. Не забывайте, что, когда класс разрабатывается в системе Смолток, он разрабатывается не сам по себе. В действительности он расширяет уже существующую (и очень большую) библиотеку классов. Поэтому надо стараться шире использовать уже существующий код, искать среди существующих классов те, которые обеспечивают нужные функциональные возможности, и только не найдя их, разрабатывать и создавать собственные классы.

При реализации класса использовать наследование. Можно сказать, что это правило — уточнение предыдущего. Когда вы уже все знаете о создаваемом классе, перед тем как приниматься за его *реализацию*, вспомните о наследовании. Напомним, что согласно механизму наследования, каждый класс наследует из всех своих суперклассов структуру (переменные) и поведение (методы). При этом наследуемые методы можно переопределять.

Использование наследования — самый последний, но весьма важный момент, о котором следует позаботиться при создании класса. Но почему наследование надо рассматривать последним? Так происходит потому, что реально нельзя определить, *что и как* создаваемые классы должны наследовать из уже существующих классов и друг у друга до тех пор, пока не ясно, *что* они должны делать и *как* они собираются это делать. Конечно, как только настанет момент выбора места класса в иерархии, вполне возможно, придется возвратиться и изменить некоторые детали, делая их более «наследуемыми». Это важная часть процесса разработки. Стоит заметить, что из-за одиночного наследования место класса в иерархии может определяться неоднозначно.

Наследование «прививает» создаваемый класс на существующее дерево классов, во-первых, реально делая новый класс совместимым по форме и функциям с уже существующим кодом, и, во-вторых, создавая условия для разработки хорошего и многократно используемого кода.

С другой стороны, наследование нарушает инкапсуляцию и делает реализацию класса совместно используемой. Помните, что классы могут иметь общий протокол, находясь в разных ветвях иерархии.

Во всем стремиться к простоте. Очевидное, но стоящее напоминания правило. Продолжая итеративную работу над классом, старайтесь делать его все проще и проще, пока присущие ему функциональные возможности не станут наиболее общими и, следовательно, более мощными.

11.2. Многократное использование кода

Одно из основных преимуществ объектно-ориентированного языка программирования — возможность многократно использовать созданный ранее код. К сожалению, подобно всему хорошему, такая возможность не приходит сама собой и не проявляется автоматически. Фактически каждый класс в процессе его разработки должен *проектироваться* для многократного использования, если, разумеется, его код должен повторно использоваться. Перед тем как рассмотреть проектирование для многократного использования, давайте посмотрим на то, что мы подразумеваем под термином «многократно используемый».

Классы являются многократно (повторно) используемыми только в том случае, когда другие программисты скорее станут использовать их, а не писать собственные. Это означает, что повторно используемый класс должен не только отвечать техническим требованиям, но и быть понятным и легким в применении. Фраза «отвечать техническим требованиям» означает, что класс надежно выполняет поставленные задачи, причем эффективно и без ненужных побочных эффектов. Создание класса, легкого в применении и понятного пользователю, означает полную гарантию предсказуемости его поведения.

Важно помнить, что не все классы *должны* повторно использоваться. Возможность многократного использования стоит времени и усилий на разработку, и они должны быть потрачены на создание классов, наиболее подходящих для этого. Однако если принято решение, что класс должен проектироваться для повторного использования, следует подумать и о том, *каким образом* он должен далее использоваться.

Для многократного использования класса существуют два способа. Во-первых, пользователь, использующий класс, может выполнить *композицию* объектов, помещая экземпляр используемого класса внутрь экземпляра собственного класса в виде переменной (как *часть* объекта), или, во-вторых, он может *наследовать* из этого класса. Эти механизмы отражают две доступные в системе Смолток модели многократного использования. Существующую библиотеку классов можно представлять как набор частей, которые могут быть собраны вместе, или как дерево, из которого можно организовать наследование в любой его точке.

В системе Смолток повторное использование класса чаще всего сводится к первому механизму, то есть чаще создаются и используются экземпляры существующих классов. Создавая собственный класс, вы все время будете использовать числа, строки, наборы, окна, виджеты и так далее. Из большинства классов существующей иерархии наследование реализуется

так уж часто будете. Исключение составляет класс `Object` — суперкласс для всех других классов, и класс `ViewManager` — суперкласс для окон приложений, о котором пойдет речь в разделе 16.2.

Все это означает, что, когда вы создаете собственные повторно используемые классы, вы сначала должны задуматься о том, как пользователь будет работать с их экземплярами, а уж потом о том, как он сможет из них наследовать. Проектирование наследования — достаточно сложная задача.

Формирование объектов из частей, разработанных для совместного использования, позволяет обойти ограничение системы, связанное с одиночным наследованием (каждый класс имеет точно один непосредственный суперкласс, а не комбинацию многих родительских классов). Каждая часть составного объекта имеет собственное место в иерархии наследования. Попытка найти в иерархии место для такого класса, экземпляр которого представлял бы объект, эквивалентный составному, всегда будет приводить к компромиссу.

Но на одних пожеланиях повторно используемого класса не создашь. Каким еще правилам стоит следовать? Наиболее очевидно, что всегда надо стараться создавать код общего вида. Возможности, предоставляемые системой Смолток, основаны на том, что язык Смолток — это язык «без типов» (“typeless”). Если в интерфейсе создаваемого класса не нужно ограничивать тип объекта, то и не ограничивайте его. Если ограничения необходимы, старайтесь сделать их минимальными. Например, если метод принимает в качестве параметра набор, пишите код, работающий с максимально большим числом разных наборов, а не только с массивами.

Один из способов сделать класс общим состоит в том, чтобы избегать в теле методов прямого использования констант. Если подобное имеет смысл, встраивайте их в интерфейс в качестве методов. Конечно, это несколько усложнит интерфейс класса, поэтому надо стараться создавать более простые версии методов с меньшим числом параметров. А эти методы, возможно, будут вызывать более сложные методы со значениями по умолчанию для некоторых параметров. Такой прием часто используется в иерархии классов системы, где он иногда проходит много уровней все возрастающей общности (и возрастающего числа параметров), пока не возникнет метод, который фактически вызывается для исполнения и выполняет всю работу. Старайтесь разрабатывать классы так, чтобы пользователи знали только о тех возможностях, которые они реально используют.

В заключение стоит еще сказать, что только опыт эксплуатации созданного класса расставит все по своим местам. Нельзя предсказать будущее. Но если класс удовлетворяет основным требованиям, не стоит «на предвидение» тратить время.

Наверное, вы уже обратили внимание, что от общих правил, относящихся к классу в целом, мы незаметно перешли к частным правилам, относящимся к вопросам создания кода, то есть непосредственно к реализации класса. Чтобы наш разговор об этом был более содержательным и более понятным, давайте его продолжим в конце этой части после того, как построим несколько классов.

11.3. Использование отладчика

Век вывихнут. О злобный жребий мой!
Век вправить должен я своей рукой. . .

Шекспир. «Гамлет»

11.3.1. Определение класса `WordIndex`

Чтобы получить некоторый опыт отладки создаваемого кода, построим новый класс, преднамеренно допустив ошибки в коде методов, и постараемся найти их¹. Таким образом, в этом параграфе мы акцентируем внимание не на особенностях создания класса (этому мы посвятим все остальные примеры), а именно на поиске ошибок с помощью отладчика.

Определим новый класс с именем `WordIndex`, экземпляры которого позволят нам создавать базу данных документов, опираясь на те слова, которые они содержат. Документы (текстовые файлы) рассматриваются как цепочки слов, состоящие из алфавитно-цифровых символов и разделенные не алфавитно-цифровыми символами. Запрос к базе данных состоит из массива строк (экземпляров класса `String`), при этом каждая строка представляет слово. В ответ на запрос экземпляр класса `WordIndex` должен возвращать набор имен файлов, тексты которых содержат все указанные в запросе слова. Например, таким способом можно определить по характеристикам, хранящимся в компьютере отдела кадров, тех служащих, характеристики которых содержат слова «объектно-ориентированное программирование».

Каждый экземпляр класса `WordIndex` будет иметь следующие переменные экземпляра:

`documents` — множество строк, каждая из которых представляет путь к файлу, содержащему тот документ, слова которого будут введены в экземпляр класса.

¹ Этот пример взят из документации, поставляемой разработчиком вместе с системой *Smalltalk Express*.

words — словарь, каждый ключ которого есть слово, а каждое значение является множеством, содержащим пути к тем документам из базы данных, которые содержат слово-ключ.

Итак, откройте окно просмотра иерархии классов и добавьте в систему новый класс, определение которого имеет следующий вид:

```
Object subclass: #WordIndex
  instanceVariableNames: 'documents words '
  classVariableNames: ''
  poolDictionaries: ''
```

Определите в **WordIndex** следующие шесть методов экземпляра:

addDocument: pathName

"Добавить все слова из документа, описанного строкой **pathName**, к словарю слов."

```
| word wordStream |
(documents includes: pathName)
  ifTrue: [self removeDocument: pathName].
wordStream := File pathName: pathName.
documents add: pathName.
[(word := wordStream nextWord) == nil]
  whileFalse: [self addWord: word asLowerCase to: pathName].
wordStream close
```

addWord: wordString for: pathName

"Добавить **wordString** к словарю слов для документа, описанного именем **pathName**."

```
(words at: wordString) add: pathName
```

initialize

"Инициализировать новый экземпляр класса **WordIndex**."

```
documents := Set new.
words := Dictionary new
```

locateDocuments: queryWords

"Возвратить множество, содержащее имена тех документов, которые содержат все слова из массива **queryWords**."

```
| answer bag |
bag := Bag new.
answer := Set new.
queryWords do: [:word |
  bag addAll: (documents at: word ifAbsent: [#( )]).
bag asSet do: [:document |
  queryWords size = (bag occurrencesOf: document)
```

```

    ifTrue: [answer add: document]].
  ^ answer asSortedCollection asArray

```

removeDocument: pathName

```

"Удалить из словаря words строку pathName, описывающую
документ."
words do: [ :docs | docs remove: pathName].
self removeUnusedWords

```

removeUnusedWords

```

"Удалить из словаря words все слова, имеющие пустой набор
документов."
| newWords |
newWords := Dictionary new.
words associationsDo: [ :anAssoc | anAssoc value isEmpty
    ifFalse: [newWords add: anAssoc]].
words := newWords

```

Чтобы добавить определение класса `WordIndex` и его методы в образ системы *Smalltalk Express*, можно воспользоваться готовым файлом, который поставляется вместе с системой, выполняя, например, в рабочем окне выражение:

```
(File pathName: 'tutorial\wrdindx8.st') fileIn; close
```

Чтобы имя нового класса появилось в иерархии, надо из меню `Classes` окна просмотра иерархии классов выбрать пункт `Update (Обновить)`.

11.3.2. Отладка класса `WordIndex`

Давайте рассмотрим класс `WordIndex` в терминах сообщений, которые создают экземпляр класса `WordIndex` и посылают запросы. Напомним, что преднамеренно в приведенных текстах методов сделаны ошибки.

Итак, предположим, что класс работает так, как надо, и посмотрим, что он делает. Чтобы воспользоваться экземпляром класса, его сначала надо создать; для этого следует выполнить выражение:

```
MyIndex := WordIndex new initialize
```

Объект `MyIndex` создается как новая глобальная переменная, которая ссылается на новый экземпляр класса `WordIndex`. Метод `initialize` инициализирует все переменные экземпляра `WordIndex`: теперь переменная `documents` содержит пустое множество, а переменная `words` содержит пустой словарь. При выполнении этого выражения ошибка не возникла.

Далее добавим файлы `chapter.5` и `chapter.6`, поставляемые с системой, как документы экземпляра `MyIndex`. Это делает метод `addDocument:`, ко-

торый создает файловый поток, чтобы просмотреть документ, многократно посылая файловому потоку сообщение `nextWord`, которое получает следующее слово, а затем использует метод `addWord:for:` для ввода каждой пары «слово/документ» в словарь `words`. Значит, чтобы добавить слова из файлов `chapter.5`, `chapter.6` в экземпляр, надо выполнить следующие два выражения:

```
MyIndex addDocument: 'tutorial\chapter.5'.
```

```
MyIndex addDocument: 'tutorial\chapter.6'.
```

Возникает окно `Walkback`, значит есть какая-то ошибка. В данном случае метка окна `Walkback` информирует нас о том, что сообщение `addWord:to:` не было понято (`addWord:to: not understood`), в то время как верхняя строка в текстовой панели показывает `WordIndex` как класс, объект которого не понял сообщение.

Как уже отмечалось выше, можно предпринять одно из трех действий. В этом случае, чтобы установить причину ошибки, достаточно информации из окна `Walkback`. Посмотрите на код в классе `WordIndex`, используя окно просмотра иерархии классов. Там определен метод с именем `addWord:for:`, а в методе `addDocument:` послано сообщение `addWord:to:`, которое и не было понято. Чтобы исправить ошибку, закройте окно `Walkback` и, пользуясь окном просмотра иерархии классов, исправьте метод `addDocument:`, вставляя `addWord:for:` вместо `addWord:to:`.

Попробуем снова добавить файлы обучающей программы, используя те же выражения. На этот раз возникает новое окно `Walkback`, заголовок которого сообщает: `Key is missing` (Ключ отсутствует). Так как причина ошибки не столь очевидна, получим более подробную информацию, открывая окно `Debugger`, для чего в окне `Walkback` нажмем на кнопку `Debug`.

Левая верхняя панель (списковая панель) окна `Debugger` повторяет информацию из окна `Walkback`, и ее можно использовать для того, чтобы выбирать в ней строки посланных сообщений (`Walkback`-строки), получая в других панелях окна связанную с выбранной строкой информацию.

Чтобы понять причину ошибки, сначала выберем в левой верхней панели строку `WordIndex>>addDocument:`. В нижней текстовой панели появится текст метода `addDocument:` из класса `WordIndex` с выделенным текстом `self addWord: word asLowerCase for: pathName.`

Это означает, что данное сообщение было послано, но его выполнение не было завершено. Просматривая значения всех переменных в средней верхней панели, убеждаемся в том, что здесь все в порядке. Теперь выберем в левой верхней панели строку `WordIndex>>addWord:for:`, в нижней текстовой панели появится текст метода `addWord:for:` с выделенным текстом `words at: wordString.` Со значениями переменных здесь тоже все в порядке.

Поднимемся в левой верхней панели еще на строчку и выберем `Dictionary>>at:`. Текстовая панель внизу окна отобразит исходный текст метода `at:` класса `Dictionary`:

at: aKey

```
"Возвратить из получателя значение пары key/value, ключ
которой равен aKey. Если ключа нет, сообщить об ошибке."
| answer |
^ (answer := self lookUpKey: aKey) == nil
  ifTrue: [self errorAbsentKey]
  ifFalse: [answer value]
```

с выделенным текстом `self errorAbsentKey`, который является текущим выражением, выполнение которого в этом методе не было завершено к моменту останова. Метод `at:` возвращает значение той пары `key/value` из словаря, ключ которой равняется аргументу `aKey`. Если ключ отсутствует, вызывается метод `errorAbsentKey`, приводящий к инициализации окна `Walkback`.

В верхней средней панели найдем `self`, аргумент `aKey` и временную переменную `answer`. Выбрав `self`, видим в панели справа его значение — пустой словарь. Теперь выберем аргумент `aKey`, его значение — строка `'tutorial'`, первое слово в файле. Таким образом, мы пытались производить поиск в пустом словаре.

Возвратимся к строке `WordIndex>>addWord:for:`, выберем параметр `wordString`. Как и раньше, это строка `'tutorial'`. Таким образом, мы обращались к словарю `words`, но не проверили, присутствует ли в словаре заданный ключ. Исправим метод `addWord:for:` в нижней текстовой панели окна `Debugger` так, чтобы он выглядел следующим образом:

addWord: wordString for: pathName

```
"Добавить wordString к словарю слов для документа,
описанного именем пути pathName."
(words includesKey: wordString)
  ifFalse: [words at: wordString put: Set new].
(words at: wordString) add: pathname.
```

Сохраним новую редакцию метода и посмотрим, что произошло в окне `Debugger`. Строки списковой панели, расположенные выше строки `WordIndex>>addWord:for:`, исчезли, так как был изменен метод, с которым они были связаны, а сама строка `WordIndex>>addWord:for:` стала выделенной. Выберем пункт `Restart` из меню `Debugger`, выполнение возобновится, снова посылая выделенное сообщение. Поскольку ошибка исправлена, окно `Debugger` исчезнет, а метод наконец-то выполнится.

Чтобы послать запрос объекту `MyIndex`, надо воспользоваться сообщением `locateDocuments:`. Каждый запрос должен возвращать массив строк,

содержащий пути к тем документам, в которых есть все слова, перечисленные в аргументе этого сообщения.

Метод `locateDocuments`: сложнее остальных методов в классе. Он использует экземпляр класса `Bag`, чтобы в нем накапливать пути для всех файлов, содержащих каждое слово из запроса. Напомним, что экземпляры класса `Bag`, в отличие от множеств, могут содержать дубликаты объектов. Затем экземпляр класса `Bag` исследуется с целью отыскать в нем все те документы, которые повторяются столько раз, сколько слов в запросе; это именно те документы, которые содержат все слова.

Итак, пробуем послать запрос, выполняя выражение:

```
MyIndex locateDocuments: #'show' 'class'.
```

Снова возникает окно `Walkback`, информируя о том, что сообщение `at:ifAbsent:`, посланное экземпляру класса `Set`, не было им понято. Откроем окно `Debugger`, выберем в левой верхней панели строку

```
Set(Object) >> doesNotUnderstand:
```

Затем выберем `self` из списка временных переменных. Его значение равно `Set('tutorial\chapter.6' 'tutorial\chapter.5')`.

Теперь выберем строку `[] WordIndex>>locateDocuments: (3-ю сверху)`. Посмотрим на исходный текст метода, в котором внутри блока выделено незавершенное сообщение `at:ifAbsent:`, и исследуем значения временных переменных. В сообщении в качестве получателя используется переменная экземпляра `documents` — экземпляр класса `Set`. Значение, распечатанное выше, подтверждает это, поскольку содержит пути к документам. Следовательно, или мы послали неправильное сообщение объекту `documents`, или `documents` — неправильный получатель сообщения. Но код

```
bag addAll: (documents at: word ifAbsent: [#()])
```

пытается добавить в переменную `bag` все документы, которые включают строку, содержащуюся в переменной `word`. Такая информация содержится в переменной `words`. Значит, неправильным является получатель сообщения. Следует использовать словарь `words`:

```
bag addAll: (words at: word ifAbsent: [#()]).
```

Используя текстовую панель окна `Debugger`, изменим метод `locateDocuments:`, сохраним измененный метод и повторим запрос. Приложение должно работать корректно. Отладка завершена.

11.3.3. О кнопках **Hop**, **Skip** и **Jump**

Теперь, когда мы отладили класс `WordIndex`, давайте попробуем использовать отладчик `Debugger` для того, чтобы увидеть, как функционирует приложение, непосредственно наблюдая за посылкой сообщений. Откроем

окно **Walkback**, а из него окно **Debugger**, выполняя вместе следующие два выражения:

```
self halt. "Контрольная точка — вызов окна Walkback."  
MyIndex locateDocuments: #'each' 'talk').
```

Нажатие кнопок **Hop**, **Skip** и **Jump**, как уже отмечалось, вызывает выполнение порции кода. Дважды нажмите кнопку **Hop** и понаблюдайте за изменениями окна **Debugger**. Первое нажатие выделит выражение `MyIndex locateDocuments: #'each' 'talk')`. После второго нажатия кнопки выражение начнет выполняться, и выделенным окажется первое выражение метода `locateDocuments:`, то есть `Bag new`. Нажмите **Hop** снова. Выполнение сделает следующий шаг и перейдет в метод `new` класса `Bag`, и так далее. Обращаясь к панелям переменных, можно исследовать состояние объектов после каждого шага **Hop**.

Теперь нажмите несколько раз кнопку **Skip**. Обратите внимание, что выделяемый код является частью того же самого метода до тех пор, пока выполнение метода не закончится. Это позволяет сконцентрировать внимание на пошаговом выполнении данного метода и проигнорировать вычисление всех сообщений, посылаемых этим методом.

Jump выполняет наиболее «длинные» шаги из всех трех пошаговых кнопок. Если не устанавливались контрольные точки, **Jump** выполняет весь код до конца отлаживаемого выражения. Следовательно, нажимая во время отладки один раз **Jump**, мы завершим выполнение выражения `MyIndex locateDocuments: #'each' 'talk')`.

ГЛАВА 12

ДОМАШНЯЯ БУХГАЛТЕРИЯ

... И был глубокий эконом ...
А. Пушкин. «Евгений Онегин»

Приведем простой пример класса, в котором для хранения информации используются словари. Назовем наш класс `FinancialBook` (Финансовая-Книга). Основные идеи примера взяты из книги [5] (в ней он назывался `FinancialHistory`), но мы его немного изменили для того, чтобы в одном примере охватить больше деталей и создать модель для построения приложения с графическим интерфейсом пользователя.

12.1. Цель и определение класса

Нам хотелось бы в своих семьях вести строгий учет и контроль над поступлениями и тратами денежных средств. Деньги зарабатываются всеми членами семьи, причем каждый может работать в нескольких местах, то есть может иметь несколько *источников*. И мы хотим знать, какая сумма денег приходит в семью из каждого такого источника. Чтобы не усложнять данную учебную модель, мы будем учитывать не начисленные, а реальные суммы, которые попадают в семью за вычетом всех налогов. Все суммы будем задавать в рублях и копейках, принудительно округляя числа до двух знаков после запятой. В расходовании денежных средств (*expenditures*) также принимают участие *все* члены семьи, причем *причины расходования средств* (на еду, на одежду, на книги и тому подобное) не зависят от того, кто конкретно их тратит. Нас интересует не то, *кто* сколько тратит денег, а *на что* тратятся деньги.

Данный класс мы строим для того, чтобы в любой момент иметь возможность, во-первых, внести в нашу семейную книгу учета новую информацию о доходах и расходах и, во-вторых, получить информацию о том, сколько денег сейчас в наличии, какова сумма доходов и сумма расходов за весь прошедший период ведения книги учета, какова сумма, полученная из каждого источника, какова сумма, потраченная по каждой причине. Таким образом, проектируемый класс должен отвечать на следующие сообщения: `receive: amount from: source` запоминает, что получили сумму `amount` из источника `source`; в качестве источника договоримся использовать его текстовое описание (строку); возвращает финансовую книгу;

`totalReceivedFrom: source` возвращает сумму денег, полученную из источника `source`;

`sources` возвращает набор всех источников поступления денег;

`spend: amount for: reason` запоминает, что сумму `amount` израсходовали по причине `reason`; причины тоже будем представлять описывающими их строками; возвращает финансовую книгу;

`totalSpentFor: reason` возвращает сумму денег, потраченную по причине `reason`;

`reasons` возвращает набор всех причин расходования средств.

`totalIncome` возвращает сумму доходов;

`totalExpenditure` возвращает сумму расходов;

`cashOnHand` возвращает наличную сумму денег.

Хранение числовых значений со строками в качестве ключей реализуем в виде словарей. Хотя обычно в бухгалтерии источники доходов и причины расходов формализуют как счета учета и хранят вместе, мы в этом примере для простоты заведем два различных словаря `incomes` и `expenditures` для доходов и расходов, соответственно.

Общие суммы доходов и расходов можно всегда вычислить. Но поскольку записей в соответствующих словарях может быть много, такое вычисление может стать неэффективным. В то же время, если общие суммы доходов и расходов хранить в переменных объекта и обязанность изменять общие суммы возложить на методы добавления новых доходов и расходов, накладные затраты будут ничтожны. Поэтому заведем еще две числовые переменные `totalIncome` и `totalExpenditure`. При такой структуре сумму наличных денег вычислить всегда просто: от суммы доходов надо отнять сумму расходов. Из всего сказанного следует такое определение класса:

```
Object subclass: #FinancialBook
  instanceVariableNames:
    'incomes expenditures totalIncome totalExpenditure '
  classVariableNames: ' '
  poolDictionaries: ' '
```

12.2. Создание нового экземпляра

Как мы уже знаем, основные методы класса — это методы, создающие новый экземпляр с разумной инициализацией переменных. В этом классе создадим два метода: один с нулевой стартовой суммой денег (`new`), а другой с задаваемой аргументом стартовой суммой денег (`newWith:`).

new

"Создает новый экземпляр с нулевой стартовой суммой."

^ super new setInitialBalans: 0

newWith: amount

"Создает новый экземпляр со стартовой суммой amount."

^ super new setInitialBalans: amount

Обратим внимание на то, что методы создания экземпляров (**newWith:** и **new**) не имеют прямого доступа к переменным создаваемого экземпляра, поскольку выполняются не классом, которому принадлежат новые экземпляры, а метаклассом класса. Поэтому методы создания экземпляра сначала создают экземпляр с неинициализированными переменными, а потом посылают ему сообщение инициализации **setInitialBalance:**. Метод для этого сообщения находится в классе **FinancialBook** среди методов экземпляра. Он уже «умеет» присваивать переменным экземпляра те значения, которые обеспечивают правильное поведение созданного объекта.

Сообщения инициализации, как правило, не рассматриваются как часть доступного интерфейса экземпляра. Обычно такие сообщения посылаются только методами класса и рассматриваются как *частные*. Если реализация поддерживает категории методов, то такой метод должен находиться в категории частных методов экземпляра (**private**). Если такой поддержки нет, как в *Smalltalk Express*, то предназначение метода рекомендуется подчеркивать словом «Частный», расположенным в самом начале комментария к методу.

setInitialBalans: amount

"Частный. Инициализирует все переменные нового экземпляра."

(amount isNumber and: [amount > 0])

ifTrue: [totalIncome := amount roundTo: 0.01]

ifFalse: [totalIncome := 0].

totalExpenditure := 0.

incomes := Dictionary new.

expenditures := Dictionary new

12.3. Методы доступа к информации

Чтобы вносить в книгу учета новую информацию и получать из этой книги нужные нам сведения, необходимы методы доступа к переменным экземпляра, которые позволяют получать (**get-методы**) и определять (**put-методы**) значения переменных. Не всегда и не ко всем переменным пользователь может обращаться и/или изменять их значения. В данном классе все переменные экземпляра будут получать свои значения в процессе до-

бавления в экземпляр новой информации о доходах и расходах. Поэтому put-методы пока определять не будем. Что же касается get-методов, то, по нашему мнению, методы, возвращающие значения переменных `totalIncome` и `totalExpenditure`, должны входить в интерфейс экземпляра класса, а методы, возвращающие словари `incomes` и `expenditures` — нет, и мы их тоже не станем пока определять. Не имея графического интерфейса, представить полное содержание словарей весьма затруднительно (они могут оказаться достаточно большими). Всегда лучше думать не о переменных, а об атрибутах объекта, которым могут соответствовать (а могут и не соответствовать) переменные. Итак, из всех методов, возможных в этом классе, сначала определим только три следующих:

totalExpenditure

"Возвращает общую сумму всех расходов."

^ totalExpenditure

totalIncome

"Возвращает общую сумму доходов."

^ totalIncome

cashOnHand

"Возвращает сумму наличных денег."

^ totalIncome - totalExpenditure

Работа со словарями должна быть согласована с изменением итоговых значений, поэтому мы не будем предоставлять доступа к самим словарям, а лишь обеспечим достаточную информацию об их содержимом.

sources

"Возвращает множество всех источников поступления денег."

^ incomes keys

totalReceivedFrom: source

"Возвращает общую сумму, полученную из источника source. Если такого источника нет, возвращает 0."

(incomes includesKey: source)

if True: [^ incomes at: source]

if False: [^ 0]

reasons

"Возвращает множество причин расходов."

^ expenditures keys

totalSpentFor: reason

"Возвращает общую сумму, потраченную по причине reason.
 Если по указанной причине деньги не тратились, возвращает 0."
 (expenditures includesKey: reason)
 ifTrue: [^ expenditures at: reason]
 ifFalse: [^ 0]

12.4. Методы ввода информации

Для ввода новой информации нужны всего два метода: один для изменения доходной части, а другой — расходной. Если ключи уже есть в словаре, новую сумму надо добавить к старой, а если нет, создать по данным новую пару и поместить ее в словарь. Кроме того, необходимо еще изменить итоговое значение. Оба метода должны возвращать получателя сообщения, то есть экземпляр класса `FinancialBook`.

В следующей части мы напишем интерфейс пользователя для нашей книги учета доходов и расходов и на него возложим проверку правильности ввода информации пользователем. Здесь же будем считать, что все данные введены правильно.

receive: amount from: source

"Запоминает, что из источника source получена сумма amount."
 incomes at: source
 put: (self totalReceivedFrom: source) + amount roundTo: 0.01.
 totalIncome := totalIncome + amount roundTo: 0.01

spend: amount for: reason

"Запоминает, что по причине reason потрачена сумма amount."
 expenditures at: reason
 put: (self totalSpentFor: reason) + amount roundTo: 0.01.
 totalExpenditure := totalExpenditure + amount roundTo: 0.01

Создание класса для решения поставленных выше задач завершено. Можно создать книгу учета доходов и расходов семьи, выполняя, например, следующие выражения:

```
Smalltalk at: #FamilyFinancialBook1999 put: nil.
FamilyFinancialBook1999 := FinancialBook new.
Внесем в эту книгу некоторую информацию:
FamilyFinancialBook1999 receive: 500 from: 'Зарплата Жены';
    receive: 400 from: 'Зарплата Мужа';
    receive: 300 from: 'Пенсия Тещи';
    receive: 100 from: 'Стипендия Сына'.
```

```
FamilyFinancialBook1999 spend: 150.50 for: 'На Питание';  
    spend: 49.50 for: 'На Книги';  
    spend: 300 for: 'На Одежду'.
```

Теперь на основании введенных данных из книги учета можно получить полезную информацию:

```
FamilyFinancialBook1999 cashOnHand      → 800.00
```

Чтобы сохранить этот объект для дальнейшей работы, следует сохранить образ системы.

12.5. Задания для самостоятельной работы

1. В классе `FinancialBook` определите все оставшиеся методы доступа к переменным экземпляра, указывая в комментарии на то, что это частные методы. Они нам потребуются в главе 18.
2. Перепишите класс `FinancialBook` так, чтобы он хранил не только итоговые суммы доходов и расходов, итоговые суммы для каждой статьи доходов и расходов, но и каждую конкретную сумму и дату выполнения операции по каждой статье доходов и расходов.
3. Перепишите класс `FinancialBook` так, чтобы он мог безопасно работать в многопоточной среде.
4. Перепишите класс `FinancialBook` так, чтобы он мог хранить информацию и о том, *кто* тратит деньги.

ГЛАВА 13

КЛАССЫ TRIANGLE И NUMBERARRAY

13.1. Постановка задачи

Теперь все внимание мы направим на технологию создания нового класса. Следующая наша задача будет связана со школьной математикой. Всем хорошо известно, что такое треугольник. Все, надеемся, хорошо помнят, сколько разнообразных задач, связанных с «решением треугольников», нам пришлось перерешать на протяжении наших школьных лет. Вот мы и создадим класс с именем `Triangle`, экземплярами которого будут треугольники, а сообщения к ним позволят вычислять по имеющимся данным его новые характеристики.

Вначале договоримся об обозначениях, которые будем применять в последующих рассуждениях и формулах. Углы будем всегда измерять в радианах и обозначать буквами A, B, C . Стороны треугольника будем называть по противоположащим им углам и их длины обозначать буквами a, b, c , соответственно. Высоты и медианы, если они нам понадобятся, будем обозначать по углу. Например, β_A — биссектриса угла A , h_A — высота треугольника, опущенная из угла A на сторону a , m_A — медиана, проведенная из угла A в середину стороны a . Площадь треугольника будем обозначать через S , а полупериметр треугольника — через p , то есть $2p = a + b + c$.

Пользуясь этими обозначениями, напомним три основные теоремы для треугольников общего вида и формулы вычисления основных его характеристик (см., например, [27, с. 277, 364–367]):

Теорема косинусов: $a^2 = b^2 + c^2 - 2bc \cos A$.

Теорема синусов: $\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$.

Теорема (формула) Герона: $S = \sqrt{p(p-a)(p-b)(p-c)}$.

Радиус описанного круга: $R = \frac{abc}{4S}$.

Радиус вписанного круга: $r = S/p$.

Длина биссектрисы: $\beta_A = \frac{2}{b+c} \sqrt{bc p(p-a)}$.

Длина медианы: $m_A = \frac{1}{2} \sqrt{2b^2 + 2c^2 - a^2}$.

Длина высоты: $h_A = 2S/a = b \sin C = c \sin B$.

Основная задача состоит в том, чтобы по известным данным суметь вычислить неизвестные, а анализ приведенных формул показывает, что наиболее универсальным будет представление треугольника длинами трех его сторон. Обозначая длины сторон треугольника через *sideA*, *sideB*, *sideC*, дадим следующее определение класса *Triangle*:

```
Object subclass: #Triangle
  instanceVariableNames: 'sideA sideB sideC'
  classVariableNames: ''
  poolDictionaries: ''
```

При анализе приведенных выше формул нетрудно заметить, что нам следует уметь вычислять сумму и произведение длин всех сторон треугольника, полезно знать наименьшую и наибольшую длину его сторон, то есть работать с набором длин его сторон. Этот набор представим в виде числового массива, с которым и будем производить требуемые вычисления. Но в *Smalltalk Express* нет числовых массивов. Поэтому определим еще один класс, необходимый нам для решения поставленной задачи. Назовем его *NumberArray*. Нетрудно заметить, что разумнее всего такой класс сделать подклассом класса *Array*. Отличительной особенностью класса *NumberArray* от созданных нами ранее классов будет то, что определяется он с помощью сообщения, первое ключевое слово которого *variableSubclass:*, а не *subclass:*. Это означает, что экземпляры данного класса будут иметь индексированные переменные. Определение этого класса будет таким:

```
Array variableSubclass: #NumberArray
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

Других новых классов для решения задачи нам определять не нужно. Сначала займемся вспомогательным классом *NumberArray*.

13.2. Поведение объектов *NumberArray*

Так же, как и во всех языках программирования, перед тем как использовать объект, необходимо убедиться в том, что все его переменные инициализированы должным образом. Напомним, что система автоматически инициализирует все переменные значением *nil*. Но когда они должны иметь другие, более осмысленные значения, об этом следует позаботиться явно. В нашем случае экземпляры класса *NumberArray* должны вести себя так же, как и все массивы, но состоять только из чисел. Следовательно, придется переопределить методы, создающие новый экземпляр класса. Кроме

этого, экземпляры будут обладать новым поведением. Например, они должны уметь вычислять наибольший и наименьший элемент массива, вычислять сумму и произведение всех своих элементов. Это тот минимум новых свойств, которые нам потребуются при решении основной задачи.

Начнем с метода класса, определяющего новый экземпляр. Как известно, при создании нового массива с помощью сообщения `new: aSize`, создается экземпляр класса `Array`, все элементы которого равны `nil`. Просматривая иерархию класса `NumberArray`, нетрудно убедиться, что все остальные наследуемые методы создания экземпляра (например, `with:`) используют это сообщение. У нас же при инициализации элементов массива должны использоваться только числа. Поэтому надо изменить реализацию только этого наследуемого сообщения.

new: aSize with: aNumber

```
"Создает числовой набор, инициализированный числом aNumber."  
^(self basicNew: aSize) atAllPut: aNumber.
```

Частным случаем этого метода будет метод, по умолчанию определяющий все элементы равными нулю:

new: aSize

```
"Создает числовой набор, инициализированный числом 0."  
^(self basicNew: aSize) atAllPut: 0.
```

Но почему все элементы набора должны быть равными? Можно придумать более сложные способы задания элементов. Напишем такой метод создания нового экземпляра, который вычисляет элементы с помощью блока с одним аргументом, когда аргумент по порядку совпадает с индексом определяемого элемента.

new: aSize accordingTo: aBlock

```
"Создает числовой набор, инициализированный числами,  
которые получаются при выполнении одноаргументного  
блока aBlock, когда его аргумент — индекс элемента."  
| array |  
array := self basicNew: aSize.  
1 to: aSize do: [:i | array at: i put: (aBlock value: i)].  
^ array.
```

Можно определять элемент массива и с помощью блока с двумя аргументами, первый из которых — произвольно задаваемое число, а второй — индекс элемента. Этот метод нам пригодится при построении матриц.

new: aSize accordingTo: aBlock with: aNumber

```
"Создает числовой набор, инициализированный числами, которые  
получаются при выполнении двухаргументного блока, его первый  
аргумент — aNumber, а второй — индекс элемента."
```

```
| array |
array := self basicNew: aSize.
1 to: aSize do: [:k | array at: k
                put:(aBlock value: aNumber value: k)].
^ array.
```

Обратимся к методам экземпляра. Сначала по той же причине, что и выше, переопределим сообщение `at:put:`, которое размещает в числовом массиве новый элемент:

at: index put: aNumber

```
"Проверяет, что aNumber число, и, если это так,
размещает его в указанной позиции index."
(aNumber isNumber)
    ifTrue: [^ self basicAt: index put: aNumber]
    ifFalse: [^ 0].
```

Расширим функциональные возможности числовых массивов и определим новые методы экземпляра, которые позднее применим при работе с треугольниками.

max

```
"Возвращает наибольший элемент числового массива."
| max |
max := self at: 1.
self do: [:a | max < a ifTrue: [max := a]].
^ max.
```

min

```
"Возвращает наименьший элемент числового массива."
| min |
min:= self at: 1.
self do: [:a | min > a ifTrue: [min := a]].
^ min.
```

mult

```
"Возвращает произведение всех элементов числового массива."
| mult |
mult := 1.
self do: [:a | mult := mult * a].
^ mult.
```

sum

```
"Возвращает сумму всех элементов числового массива."
| sum |
```

```
sum := 0.
self do: [:a | sum := sum + a].
^sum.
```

maybeSidesOfTriangle

"Возвращает true, если элементы числового массива могут служить длинами сторон треугольника. Иначе возвращает false."

```
(self size = 3) not ifTrue: [^false].
(self min <= 0) ifTrue: [^false].
(self sum <= self max * 2) ifTrue: [^false].
^true.
```

Чтобы было удобно работать с числовыми массивами, надо создать методы, которые позволяли бы производить с ними простейшие арифметические операции, вести в них поиск нужной информации.

+ aNumberOrANumberArray

"Прибавляет к каждому элементу приемника или число, или соответствующий элемент числового массива."

```
| temp |
(aNumberOrANumberArray isNumber)
  ifTrue: [temp := NumberArray new: self size
        with: aNumberOrANumberArray]
  ifFalse: [temp := aNumberOrANumberArray].
1 to: (self size min: temp size) do: [:i |
  self at: i put: (self at: i) + (temp at: i)].
```

multiplyFrom: index on: aNumber

"Умножает, начиная с index, все числа приемника на aNumber, если aNumber число. Иначе ничего не делает."

```
(aNumber isNumber)
  ifTrue: [index to: self size do: [:i |
  self at: i put: (self at: i) * aNumber]]
  ifFalse: [^self].
```

*** aNumber**

"Умножает каждый элемент приемника на aNumber, если aNumber число. Иначе ничего не делает."

```
self multiplyFrom: 1 on: aNumber.
```

- aNumberOrANumberArray

"Отнимает от каждого элемента приемника число или соответствующий элемент числового массива."

```
self + (aNumberOrANumberArray * -1).
```

indexAbsMaxElementFrom: index

"Вычисляет и возвращает индекс наибольшего по абсолютной величине элемента приемника, начиная поиск с индекса `index`."

```
| max ind el |
```

```
ind := index.
```

```
max := (self at: index) abs.
```

```
index to: self size do: [:i | el := (self at: i) abs.
```

```
(max < el) ifTrue: [max := el. ind := i]].
```

```
^ind.
```

Таких методов можно придумать много. Мы ограничились теми, которые будут полезны при реализации в следующей главе числовых матриц. В случае необходимости протокол класса `NumberArray` можно расширить, например, определяя операции округления элементов числового массива, поиска индекса наименьшего элемента массива и так далее.

13.3. Поведение объектов *Triangle*

Начнем с протокола методов класса. Этот протокол должен содержать методы создания нового экземпляра. Мы уже решили, что треугольник представляется длинами трех своих сторон. Но ведь полностью определить треугольник можно, определяя две стороны и величину угла между ними или задавая одну сторону и величину двух прилежащих к ней углов. Есть и другие, более «экзотические» способы задания треугольника.

Ограничимся только определением треугольника по трем сторонам, которое будет использовать частный метод экземпляра `sideA:sideB:sideC:`, инициализирующий переменные (он описан далее). Обратите внимание, что имена метода класса и метода экземпляра совпадают, но это не страшно: они посылаются разным объектам системы.

sideA: aNumber1 sideB: aNumber2 sideC: aNumber3

"Проверяет допустимость данных. Если они допустимы, тогда создает треугольник со сторонами, имеющими длины, равные аргументам. Если они не допустимы, сообщает об ошибке."

```
| sides |
```

```
sides := NumberArray with: aNumber1
```

```
with: aNumber2 with: aNumber3.
```

```
sides maybeSidesOfTriangle
```

```
ifTrue: [^super new sideA: aNumber1
```

```
sideB: aNumber2 sideC: aNumber3]
```

```
ifFalse: [self error: 'It is inadmissible data for triangle.'].
```

Алгоритм метода очень прост. Сначала проверяется, можно ли на основе введенных данных построить треугольник. Если нельзя — сообщается об ошибке. Если можно, то выражение `^ super new` создает новый треугольник с неинициализированными переменными (с переменными, ссылающимися на nil), а затем сообщение `sideA:sideB:sideC:` определяет все три его переменные экземпляра.

Перейдем к протоколу экземпляра и посмотрим, как он реализуется.

13.3.1. Методы для переменных

Начнем с частного метода, определяющего стороны треугольника.

sideA: aNumber1 sideB: aNumber2 sideC: aNumber3

"Частный. Определяет стороны получателя по данным аргументам."

sideA := aNumber1.

sideB := aNumber2.

sideC := aNumber3.

Сообщения `sideA`, `sideB`, `sideC` должны возвращать длину соответствующей стороны треугольника. Поскольку их реализации аналогичны, приведем только одну:

sideA

"Возвращает длину стороны a."

^ sideA.

Помимо метода, полностью определяющего экземпляр, обычно предусматриваются методы, изменяющие значение каждой из переменных экземпляра (`get`-методы). Нам кажется, что в данном классе такие методы определять не нужно, поскольку они потенциально опасны: простое изменение длины одной из сторон может привести к разрушению треугольника (сторона может оказаться или очень длинной, или очень короткой). Дело в том, что переменные экземпляра не являются независимыми друг от друга: треугольник — это не множество из трех произвольных положительных чисел. Поэтому мы считаем, что созданный треугольник изменять нельзя.

13.3.2. Методы «решения треугольника»

В приложениях, использующих треугольники, как правило, требуется вычислить неизвестные характеристики треугольника, например, найти его углы, вычислить площадь треугольника, периметр, радиус вписанной и описанной окружности, найти длину одной из его высот, медиан или биссектрис. Приведем решения некоторых подобных задач. Основой всех вычислений сделаем периметр и площадь треугольника.

perimeter

"Возвращает периметр треугольника."

^ self sides sum.

halfPerimeter

"Возвращает полупериметр треугольника."

^ 0.5 * self perimeter.

area

"Вычисляет площадь треугольника."

| p |

p := self halfPerimeter.

^ (p * (p - self sideA) * (p - self sideB) * (p - self sideC)) sqrt.

Теперь не представляет труда вычислить радиусы вписанной в треугольник и описанной около треугольника окружностей.

incircleRadius

"Вычисляет радиус вписанной окружности."

^ self area / self halfPerimeter.

circumcircleRadius

"Вычисляет радиус описанной окружности."

^ (self sides mult) / (4 * self area).

Далее, как и в теории, углы будем называть по противоположным сторонам: *angleA* — величина в радианах угла, лежащего против стороны *a*. Высоты, медианы и биссектрисы будем связывать с вершиной, из которой они выходят: *heightA* — высота, опущенная из вершины *A* на сторону *a*, *medianA* — медиана из угла *A* в середину стороны *a*, *bisectrixA* — биссектриса угла *A*.

Чтобы найти углы треугольника, можно использовать теорему косинусов. Но после уже написанных методов проще воспользоваться следствием теоремы: $\operatorname{tg} A / 2 = r / (p - a)$. Аналогичные формулы имеют место и для других углов. Эту формулу «запрограммируем» в следующем вспомогательном методе:

angle: side

"Частный. Вычисляет в радианах угол, лежащий против стороны *side*."

| t |

side = 'a' ifTrue: [t := self sideA].

side = 'b' ifTrue: [t := self sideB].

side = 'c' ifTrue: [t := self sideC].

^ 2*((self incircleRadius) / (self halfPerimeter - t)) arcTan.

Теперь легко написать методы для вычисления каждого угла треугольника. Поскольку они аналогичны, приведем только один из них.

angleA

"Вычисляет угол, лежащий против стороны sideA."

`^ self angle: 'a'.`

Если величина угла треугольника нужна не в радианах, а в градусах, нужно воспользоваться сообщением `radiansToDegrees`. Например:

```
| tr3 |
tr3 := Triangle sideA: 3 angleB: 1.2 angleC: 0.73.
^ tr3 angleA radiansToDegrees           →    69.4191455
```

Совершенно аналогично можно было бы написать методы для вычисления высот: сначала написать частный метод, реализующий вычисление по формуле, а затем на его основе написать три метода, вычисляющих длину каждой высоты треугольника. Но можно и не пользоваться вспомогательным методом, а в каждом из них непосредственно реализовать достаточно простую формулу вычисления высоты треугольника. Приведем только один из трех методов.

heightB

"Вычисляет высоту, опущенную на сторону sideB."

`^(2 * self area) / (self sideB).`

Поскольку в вычислениях используются тригонометрические и обратные тригонометрические функции, а также функция квадратного корня, все результаты, независимо от представления исходных данных, будут экземплярами класса `Float` и будут представлять приближенное значение. Если нужен результат с определенной степенью точности (обычно достаточно 4–6 знаков после десятичной точки), надо воспользоваться сообщением `roundTo:`. Например,

```
| tr1 |
tr1 := Triangle sideA: 3 sideB: 4 sideC: 5.
^(tr1 angleC) roundTo: 0.0001           →    1,5708
```

В заключение напишем два тестирующих метода, проверяющих, является ли треугольник равнобедренным или равносторонним, и переопределим одно наследуемое из класса `Object` сообщение. Так как в тестирующих методах будут сравниваться числа с плавающей точкой, ответ на эти вопросы может быть дан только с определенной степенью точности. Чтобы пояснить сказанное, обратим внимание на результаты выполнения двух выражений:

```
((2 sqrt) * (2 sqrt)) = 2.0           → false
(((2 sqrt) * (2 sqrt)) roundTo: 0.001) = 2.0.   → true
```

isEquilateralUpTo: aNumber

"Вернуть true, если треугольник равносторонний, то есть все стороны данного треугольника равны с точностью до aNumber, иначе вернуть false."

```
^ ((self sideA - self sideB) abs < aNumber) and:
  [(self sideA - self sideC) abs < aNumber].
```

isIsoscelesUpTo: aNumber

"Вернуть true, если треугольник равнобедренный, то есть две стороны данного треугольника равны с точностью до aNumber, иначе вернуть false."

```
^ ((self sideA - self sideB) abs < aNumber) |
  ((self sideA - self sideC) abs < aNumber) |
  ((self sideB - self sideC) abs < aNumber).
```

Для того чтобы после применения команды Show It печаталось понятное пользователю представление треугольника, переопределим сообщение printOn:, которое используется сообщением printString. Оно записывает в поток имя класса (это делает выражение super printOn: aStream), а затем в круглых скобках выводит длины всех сторон.

printOn: aStream

"Записать в поток aStream ASCII-представление треугольника."

```
super printOn: aStream.
```

```
aStream nextPutAll: '(';
```

```
  nextPutAll: self sideA asString; space;
```

```
  nextPutAll: self sideB asString; space;
```

```
  nextPutAll: self sideC asString;
```

```
  nextPutAll: ')'.
```

13.4. Задания для самостоятельной работы

1. Реализуйте в классе `NumberArray` возможные унарные и бинарные сообщения из класса `Number` типа округления, умножения на число и так далее.
2. Реализуйте в классе `Triangle` методы класса, которые создают новый треугольник по двум сторонам и углу (в радианах) между ними, а также по стороне и двум прилежащим к ней углам (в радианах), если указанные в сообщении данные допустимы, а иначе сообщают об ошибке.
3. Реализуйте в классе `Triangle` метод для сообщения `storeOn:`.
4. Реализуйте в классе `Triangle` методы для вычисления длин каждой из медиан и биссектрис треугольника.

5. Реализуйте в классе `Triangle` метод сравнения на равенство двух треугольников.

6. Предположим, что класс `Triangle` определен следующим образом:

```
Object subclass: #Triangle
```

```
  instanceVariableNames: 'sides'
```

```
  classVariableNames: ''
```

```
  poolDictionaries: ''
```

Здесь `sides` — числовой массив, содержащий длины трех сторон треугольника. Перепишите реализацию класса, исходя из нового определения. Мы уже отмечали, что стороны треугольника не являются полностью независимыми друг от друга. В связи с этим можно изменить определение класса `Traingle`, задавая в нем только одну переменную экземпляра, хотя, на первый взгляд, такой подход и противоречит одному из перечисленных ранее правил: не создавать сложных переменных.

7. Предположим, что класс `Triangle` определен следующим образом:

```
Object subclass: #Triangle
```

```
  instanceVariableNames: 'pointA pointB pointC'
```

```
  classVariableNames: ''
```

```
  poolDictionaries: ''
```

Здесь `pointA pointB pointC` — точки, определяющие вершины треугольника в декартовой системе координат на плоскости. Перепишите реализацию класса, исходя из нового определения. Напишите методы, которые вычисляли бы координаты точек пересечения высот, медиан, биссектрис треугольника между собой и с соответствующими сторонами треугольника.

ГЛАВА 14

МАТРИЦЫ

Следующий пример связан с матрицами, которые обычно изучаются в курсе высшей математики. Многие стандартные задачи по программированию также связаны с матрицами. Те, кто забыл, что это такое, могут обратиться к любому учебнику по алгебре или высшей математике. Читатели, никогда не изучавшие «таких наук», могут или пропустить полностью этот пример, или ограничиться протоколами строящихся классов, совершенно не обращая внимания на реализацию (что мы и советуем сделать). Все определения и результаты, необходимые для понимания интерфейса создаваемых классов, приведены в первом разделе главы.

14.1. Основные математические определения

Матрицей размера $m \times n$ (или $m \times n$ -матрицей) называется прямоугольная таблица чисел

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix},$$

состоящая из m строк и n столбцов. Если $m = n$, то матрица называется *квадратной*, а число n — ее порядком, или размерностью. Составляющие матрицу числа a_{lk} , $l = 1, \dots, m$, $k = 1, \dots, n$, называются ее *элементами*. При таком двухиндексном обозначении элементов матрицы первый индекс всегда указывает номер строки, а второй — номер столбца, на пересечении которых стоит данный элемент. Часто матрицу записывают более коротко:

$$(a_{lk})_{l=1, k=1}^{m, n}.$$

Единичной матрицей порядка n называется квадратная матрица, на главной диагонали которой стоят единицы ($a_{ii} = 1$), а остальные элементы являются нулями.

Суммой $m \times n$ -матриц $A = (a_{lk})_{l=1, k=1}^{m, n}$ и $B = (b_{lk})_{l=1, k=1}^{m, n}$ называется $m \times n$ -матрица $C = (c_{lk})_{l=1, k=1}^{m, n}$, у которой $c_{lk} = a_{lk} + b_{lk}$, $l = 1, \dots, m$, $k = 1, \dots, n$.

Произведением $m \times n$ -матрицы $A = (a_{lk})_{l=1, k=1}^{m, n}$ на число β называется $m \times n$ -матрица $\beta \cdot A = (\beta \cdot a_{lk})_{l=1, k=1}^{m, n}$.

Произведением $m \times n$ -матрицы $A = (a_{lk})_{l=1, k=1}^{m, n}$ на $n \times s$ -матрицу $B = (b_{lk})_{l=1, k=1}^{n, s}$ называется $m \times s$ -матрица $C = (c_{lk})_{l=1, k=1}^{m, s}$, у которой элемент

c_{lk} получается как сумма произведений i -х элементов l -й строки матрицы A и k -го столбца матрицы B , то есть $c_{lk} = \sum_{i=1}^n a_{li}b_{ik}$.

Для $m \times n$ -матрицы $A = (a_{lk})_{l=1, k=1}^{m, n}$ транспонированной к ней матрицей называется $n \times m$ -матрица $A^T = (a_{kl})_{k=1, l=1}^{n, m}$, в которой строки исходной матрицы становятся столбцами, а столбцы — строками.

Новые задачи, связанные с квадратной матрицей A порядка n , основываются на понятии *определителя* матрицы. Чтобы дать определение этого достаточно сложного понятия, воспользуемся рекуррентным соотношением. Для этого нам потребуется понятие подматрицы (минора) A_{ij} , получающейся из матрицы A удалением i -й строки и j -го столбца, то есть

$$A_{ij} = \begin{pmatrix} a_{11} & \cdots & a_{1(j-1)} & a_{1(j+1)} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ a_{(i-1)1} & \cdots & a_{(i-1)(j-1)} & a_{(i-1)(j+1)} & \cdots & a_{(i-1)n} \\ a_{(i+1)1} & \cdots & a_{(i+1)(j-1)} & a_{(i+1)(j+1)} & \cdots & a_{(i+1)n} \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{m(j-1)} & a_{m(j+1)} & \cdots & a_{mn} \end{pmatrix}$$

Определителем квадратной матрицы 1-го порядка A называется единственный элемент матрицы — a_{11} , который обозначается через $\det A$ или $|A|$. Если известна формула вычисления определителя квадратной матрицы $(n-1)$ -го порядка, то определителем квадратной матрицы n -го порядка A называется число

$$\det A = |A| = \sum_{j=1}^n (-1)^{i+j} a_{ij} \det A_{ij},$$

которое (как доказывается) не зависит от выбора индекса i . Поэтому обычно выбирают $i = 1$. Приведенная выше формула вычисления определителя называется разложением определителя по i -й строке. Из этого определения для квадратных матриц 2-го и 3-го порядка следует, что

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21},$$

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - \\ - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31}.$$

Отметим, что для любой квадратной матрицы $\det A = \det A^T$.

Квадратная матрица называется *вырожденной (особой)*, если ее определитель равен нулю, и *невырожденной* — в противном случае. Если A — невырожденная матрица порядка n , то для нее существует, и притом един-

ственная, матрица A^{-1} порядка n , такая, что $AA^{-1} = A^{-1}A = E$, где E — единичная матрица порядка n . Матрица A^{-1} называется *обратной* к A .

Системой n линейных уравнений с n неизвестными x_1, \dots, x_n и правыми частями b_1, b_2, \dots, b_n называется система уравнений вида

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases}$$

Набор чисел x_1, x_2, \dots, x_n , подстановка которых в каждое уравнение системы превращает его в тождество, называется решением системы уравнений. Можно доказать, что система имеет единственное решение тогда и только тогда, когда матрица этой системы $(a_{lk})_{l=1, k=1}^{n, n}$ невырождена.

Это почти вся теория, которая нам нужна для дальнейшей работы.

14.2. Постановка задачи

Чтобы решать задачи, связанные с матрицами, построим класс `NumericalMatrix` (ЧисловаяМатрица), предоставляющий следующие возможности:

- Создание матрицы любого размера.
- Задание значения любого элемента матрицы.
- Выполнение стандартных матричных операций, как-то: (1) сложение двух матриц, (2) умножение матрицы на число, (3) умножение матрицы на матрицу, (4) умножение строки матрицы на число, (5) сложение (вычитание) двух строк матрицы, (6) возвращение любого элемента матрицы, (7) построение транспонированной матрицы.
- Проверка, является ли матрица квадратной, и нахождение для такой матрицы: (а) определителя, (б) обратной матрицы, если она есть, (с) решения системы линейных уравнений с заданной правой частью.

Для решения таких задач нужна только таблица чисел, представляющая конкретную матрицу, по которой можно было бы достаточно просто определить ее размеры. При этом мы предполагаем, что уже определенная матрица не может менять своих размеров. Если требуется изменить размер матрицы — удалить или добавить несколько строк и/или столбцов, то надо определять по исходной матрице новую. Поэтому определим матрицу как массив элементов, каждый из которых является числовым массивом — экземпляром созданного ранее класса `NumberArray`. При таком определении число строк матрицы — размер массива, а число столбцов — размер любого элемента массива. Возможны и другие варианты определения структуры экземпляра класса `NumericalMatrix`.

```
Array variableSubclass: #NumericalMatrix
```

```
instanceVariableNames: ''
```

```
classVariableNames: ''
```

```
poolDictionaries: ''
```

Чтобы выделить операции, характерные только для квадратных матриц, можно было бы создать в классе `NumericalMatrix` подкласс `SquareMatrix`. Но поскольку квадратные матрицы могут возникать как результат операций с «обычными» матрицами, нам не кажется целесообразным делать это. Иначе пришлось бы в каждой такой операции отслеживать появление квадратной матрицы и определять ее как экземпляр класса `SquareMatrix`, а не как экземпляр класса `NumericalMatrix`. Проще перед выполнением операции, специфической для квадратных матриц, проверить, является ли матрица квадратной.

14.3. Протоколы класса `NumericalMatrix`

14.3.1. Создание экземпляра

Начнем с методов класса, создающих новые матрицы. В классе `NumericalMatrix` создадим два метода: первый из них будет создавать матрицу со всеми элементами, равными нулю, а второй — с элементами, значения которых задаются результатом вычисления блока с двумя переменными, где первая переменная — номер строки, а вторая — номер столбца элемента. Поскольку мы собираемся переопределять сообщения `at:` и `at:put:`, в этих методах мы воспользуемся сообщениями `basicAt:` и `basicAt:put:`.

rows: aNumber1 cols: aNumber2

```
"Создает матрицу размера aNumber1 × aNumber2
```

```
со всеми элементами, равными нулю."
```

```
| matrix |
```

```
matrix := super new: aNumber1.
```

```
1 to: aNumber1 do: [:i |
```

```
matrix basicAt: i
```

```
put: (NumberArray new: aNumber2)].
```

```
^ matrix.
```

rows: aNumber1 cols: aNumber2 accordingTo: aBlock

```
"Создает матрицу размера aNumber1 × aNumber2 с элементами,
```

```
равными значениям двухаргументного блока aBlock."
```

```
| matrix |
```

```
matrix := super new: aNumber1.
```

```
1 to: aNumber1 do: [:j | matrix basicAt: j
    put: (NumberArray new: aNumber2 accordingTo: aBlock with: j)].
^ matrix.
```

Опираясь на последний метод, напишем метод создания единичной квадратной матрицы.

newUnitSquare: anOrder

```
"Создает единичную квадратную матрицу порядка anOrder."
^ self rows: anOrder cols: anOrder
    accordingTo: [:i :j | (i = j) ifTrue: [1] ifFalse: [0]].
```

14.3.2. Методы определения размеров матрицы

Все создаваемые далее методы входят в протокол экземпляра класса *NumericalMatrix*. Начнем с методов, вычисляющих общие характеристики матрицы. Сначала переопределим метод `size`, который будет возвращать точку (экземпляр класса *Point*). Поэтому в теле метода `rows` используем псевдопеременную `super` (а не `self`), чтобы воспользоваться методом суперкласса.

size

```
"Возвращает точку, соответствующую размеру матрицы."
^ self rows @ self cols.
```

rows

```
"Возвращает число строк матрицы."
^ super size.
```

cols

```
"Возвращает число столбцов матрицы."
^ (self basicAt: 1) size.
```

isSquare

```
"Возвращает true, если получатель — квадратная матрица."
^ (self rows = self cols).
```

14.3.3. Методы доступа

Теперь обратимся к методам доступа, позволяющим посмотреть и изменить элементы уже созданной матрицы. Обратите внимание: если операция не может быть выполнена из-за несоответствия заданных индексов размерам матрицы, о такой ошибке будет сообщать метод `errorInBounds:`, наследуемый из класса *IndexedCollection*. Так что об ошибочных индексах в создаваемых методах заботиться не нужно.

row: aRow col: aColumn

"Возвращает элемент матрицы, который расположен в строке aRow и в столбце aColumn."
`^(self at: aRow) at: aColumn.`

row: aRow col: aColumn put: aNumber

"Делает число aNumber элементом матрицы, стоящим в строке aRow и в столбце aColumn; возвращает aNumber."
`^(self at: aRow) at: aColumn put: aNumber.`

row: i

"Возвращает i-ю строку матрицы."
`^super basicAt: i.`

row: i put: aNumberArray

"Заменяет строку i матрицы массивом aNumberArray, если это возможно. Если нет — ничего не делает. Возвращает aNumberArray."
`((aNumberArray isArray))
 and: [aNumberArray size = self cols]
 ifTrue: [^super basicAt: i put: aNumberArray]
 ifFalse: [^aNumberArray].`

col: j

"Возвращает столбец j матрицы."
`^NumberArray new: self rows
 accordingTo: [:i | self row: i col: j].`

at: aPoint

"Возвращает элемент матрицы, расположенный в позиции aPoint."
`^self row: aPoint x col: aPoint y.`

at: aPoint put: aNumber

"Делает число aNumber элементом матрицы, стоящим в позиции aPoint, возвращает aNumber."
`^self row: aPoint x col: aPoint y put: aNumber.`

Пользуясь созданными методами доступа, можно написать метод создания новой матрицы посредством копирования уже существующей.

copy

"Возвращает новую матрицу — копию получателя."
`^self class
 rows: (self rows) cols: (self cols)
 accordingTo: [:i :j | self row: i col: j].`

14.3.4. Операции с матрицами

Приведем без реализации те сообщения из протокола экземпляра класса *NumericalMatrix*, которые имеют дело с операциями над матрицами. Реализацию мы рассмотрим в следующем разделе, чтобы не знакомые с матрицами читатели могли, если захотят, «со спокойной совестью» его пропустить.

Класс *NumericalMatrix*

Протокол экземпляра

- + **aMatrix** Если получатель и матрица **aMatrix** имеют одинаковые размеры, возвращает матрицу, являющуюся суммой получателя и аргумента. Иначе сообщает об ошибке.
- * **aMatrixOrNumber** Возвращает матрицу, являющуюся результатом умножения получателя на аргумент **aMatrixOrNumber**, который может быть соответствующего размера матрицей или числом. Иначе сообщает об ошибке.
- roundTo: aNumber** Возвращает матрицу, каждый элемент которой с точностью **aNumber** является приближенным значением соответствующего элемента получателя.
- transpose** Возвращает новую матрицу, транспонированную по отношению к получателю.
- copyDeleteRow: aRow col: aColumn** Возвращает новую матрицу, получающуюся из получателя удалением строки **aRow** и столбца **aColumn**, если аргументы являются допустимыми. Иначе сообщает об ошибке.
- determinant** Для невырожденной квадратной матрицы вычисляет и возвращает определитель. Иначе сообщает об ошибке.
- determinantRec** Для невырожденной квадратной матрицы рекурсивно вычисляет и возвращает определитель. Иначе сообщает об ошибке.
- solveEquationWith: aRightSide** Если аргумент **aRightSide** — $1 \times n$ -матрица свободных членов, а получатель сообщения — невырожденная квадратная матрица порядка n , возвращает решение системы n линейных уравнений с n неизвестными в виде $n \times 1$ -матрицы. Если матрица вырожденная, возвращает `nil`. Если матрица не является квадратной, сообщает об ошибке.
- inverse** Для невырожденной квадратной матрицы вычисляет и возвращает обратную матрицу. Если обратная матрица не существует, возвращает `nil`. Если матрица не является квадратной, сообщает об ошибке.

Приведем несколько примеров, использующих сообщения из данного протокола. Все эти примеры можно записать как методы класса с именами, например, `example1`, `example2` и так далее, а затем для их вызова посылать соответствующее сообщение классу. Текст метода будет отличаться от приводимого ниже кода тем, что, во-первых, в нем вместо явного имени класса может стоять `self`, и, во-вторых, вместо глобальных переменных для обозначения матриц можно использовать локальные переменные метода.

Но прежде, чтобы иметь возможность представлять матрицы в виде экземпляра класса `String`, переопределим сообщение `printOn:`.

printOn: aStream

"Записывает в поток `aStream` ASCII-представление матрицы."

```
aStream cr; nextPutAll: 'a NumericalMatrix'; nextPut: $(
```

```
1 to: (self rows) do: [:row | aStream cr.
```

```
    1 to: (self cols) do: [:col |
```

```
        aStream nextPutAll: (self row: row col: col) asString; space]].
```

```
aStream nextPut: $).
```

Чтобы привести примеры, определим две матрицы:

```
M1 := NumericalMatrix rows: 4 cols: 3 accordingTo:[:i :j| 1/(j+i)] →
```

```
a NumericalMatrix(
```

```
    1/2  1/3  1/4
```

```
    1/3  1/4  1/5
```

```
    1/4  1/5  1/6
```

```
    1/5  1/6  1/7)
```

```
M2 := NumericalMatrix rows: 4 cols: 3 accordingTo:[:i :j| j*i] →
```

```
a NumericalMatrix(
```

```
    1  2  3
```

```
    2  4  6
```

```
    3  6  9
```

```
    4  8  12)
```

Тогда можно выполнить следующие выражения:

```
(M1 + M2) →
```

```
a NumericalMatrix(
```

```
    3/2  7/3  13/4
```

```
    7/3  17/4  31/5
```

```
    13/4  31/5  55/6
```

```
    21/5  49/6  85/7)
```

```
M1 * M2 transpose →
```

```
a NumericalMatrix(
```

```
    23/12  23/6  23/4  23/3
```

```
    43/30  43/15  43/10  86/15
```

```
    23/20  23/10  69/20  23/5
```

```
    101/105  202/105  101/35  404/105)
```

```
(M2 deleteRow: 2 col: 2) →
```

```
a NumericalMatrix(
```

```
    1  3
```

```
    3  9
```

```
    4  12)
```

Для примеров с квадратными матрицами определим новую матрицу:

```
M3 := NumericalMatrix rows: 3 cols: 3
      accordingTo: [:i :j | (1/(i+j)) asFloat] →
a NumericalMatrix(
  0.5      3.33333333e-1      0.25
  3.33333333e-1      0.25      0.2
  0.25      0.2      1.66666667e-1)
```

Тогда можно выполнить следующие выражения:

```
M3 determinant → 2.31481481e-5
```

```
M3 inverse →
```

```
a NumericalMatrix(
  72.0  -240.0  180.0
 -240.0  900.0  -720.0
  180.0  -720.0  600.0)
```

```
(M3 solveEquationWith: (NumericalMatrix rows: 3 cols: 1
      accordingTo: [:i :j | i])) →
```

```
a NumericalMatrix(
  132
 -600
  540)
```

14.3.5. Реализация простых операций с матрицами

Реализации методов для сложения, умножения и транспонирования матриц следуют их математическим определениям. Правда, метод для умножения матриц не прямой, а использует два частных метода для умножения матрицы на матрицу и на число. Если операция не может быть выполнена из-за несоответствия размеров матриц, о такой ошибке сообщает приведенный первым метод `errorOfCorrespondence:`. Ради краткости, комментарии в методах, вошедших в приведенный выше протокол, не приводятся.

errorOfCorrespondence: aSizeMatrix

"Частный. Сообщает об ошибке несоответствия размеров матриц."

```
^self error: 'the matrix size is incompatible with receiver.'
```

+ aMatrix

```
(self size = aMatrix size)
```

```
ifTrue: [^ NumericalMatrix rows: (self rows) cols: (self cols)
      accordingTo: [:i :j]
```

```
      (self row: i col: j) + (aMatrix row: i col: j)]]
```

```
ifFalse: [self errorOfCorrespondence: aMatrix size].
```

multiplyOnNumber: aNumber

"Частный. Возвращает результат умножения получателя на число."

```
^ NumericalMatrix rows: (self rows) cols: (self cols)
  accordingTo: [:i :j | (self row: i col: j) * aNumber].
```

multiplyOnMatrix: aMatrix

"Частный. Возвращает результат умножения получателя на матрицу aMatrix, если это возможно, иначе сообщает об ошибке."

```
| sum numcols |
numcols := self cols.
(numcols = aMatrix rows)
  ifTrue: [^ NumericalMatrix rows:(self rows) cols:(aMatrix cols)
    accordingTo: [:i :j | sum := 0.
      1 to: numcols do: [:p | sum := sum +
        ((self row: i col: p) * (aMatrix row: p col: j))].
      sum ]]
  ifFalse: [self errorOfCorrespondence: aMatrix size].
```

*** aMatrixOrNumber**

```
(aMatrixOrNumber isNumber)
  ifTrue: [^self multiplyOnNumber: aMatrixOrNumber]
  ifFalse: [^self multiplyOnMatrix: aMatrixOrNumber].
```

roundTo: aNumber

```
^ NumericalMatrix rows: (self rows) cols: (self cols)
  accordingTo: [:i :j | (self row: i col: j) roundTo: aNumber].
```

transpose

```
^ NumericalMatrix rows: (self cols) cols: (self rows)
  accordingTo: [:i :j | (self row: j col: i)].
```

copyDeleteRow: aRow col: aColumn

```
^ NumericalMatrix rows:(self rows - 1) cols:(self cols - 1)
  accordingTo: [:i :j |
    self row: ((i < aRow) ifTrue: [i] ifFalse: [i+1])
    col: ((j < aColumn) ifTrue: [j] ifFalse: [j+1])].
```

Для упрощения реализации более сложных алгоритмов линейной алгебры сначала напишем несколько вспомогательных методов, реализующих наиболее часто используемые операции с матрицами.

fromRow: ind1 subtractRow: ind2 multipliedBy: aNumber

"Преобразует матрицу, вычитая из строки ind1 строку ind2, умноженную на число aNumber."

```
self row: ind1
  put: (self row: ind1) - ((self row: ind2) copy*aNumber).
```

multiplyRow: i fromCol: j on: aNumber

"Преобразует матрицу, умножая строку *i* на число *aNumber*, начиная со столбца *j* и до конца строки."
 | aRow |
 aRow := self row: i.
 aRow multiplyFrom: j on: aNumber.
 self row: i put: aRow.

multiplyRow: i on: aNumber

"Преобразует матрицу, умножая строку *i* на число *aNumber*."
 self row: i put: (self row: i) * aNumber.

toRow: ind1 multipliedBy: aNumber addRow: ind2

"Преобразует матрицу, прибавляя к строке *ind1*, умноженной на число *aNumber*, строку *ind2*."
 self row: ind1
 put: (self row: ind1) * aNumber + (self row: ind2).

permuteRows: index1 and: index2

"Преобразует матрицу, меняя местами указанные аргументами строки."
 |temp|
 temp := self row: index1.
 self row: index1 put: (self row: index2).
 self row: index2 put: temp.

normalize: i

"Частный (для метода Гаусса). Преобразует элементы квадратной подматрицы получателя с левым верхним элементом *i@i* так, чтобы элемент *i@i* стал равным 1, а все элементы столбца *i* ниже строки *i* стали равными 0. Это достигается операциями умножения строк на числа и складыванием (вычитанием) строк."
 | aRow el |
 self multiplyRow: i fromCol: i on: (1 / (self row: i col: i)).
 aRow := self row: i.
 (i+1) to: self rows do: [:k |
 el := self row: k col: i.
 (el = 0)
 ifFalse: [self fromRow: k subtractRow: i multipliedBy: el]].

Реализацию основных алгоритмов линейной алгебры, упомянутых ранее, оставляем читателю в качестве упражнения.

14.4. Задания для самостоятельной работы

1. Напишите метод класса, создающий матрицу указанного размера, все элементы которой равны заданному числу.
2. Напишите методы класса, создающие квадратную матрицу указанного порядка, все элементы которой: (1) равны нулю, (2) равны заданному числу, (3) вычисляются с помощью блока с двумя аргументами.
3. Напишите методы, которые возвращали бы новую матрицу, представляющую собой: (1) указанную строку получателя, (2) указанный столбец получателя.
4. Напишите методы, которые заменяли бы указанный столбец или строку получателя на заданный подходящий аргумент.
5. Напишите методы, которые создавали бы новую матрицу, удаляя из существующей матрицы: (1) указанную строку, (2) указанный столбец, (3) указанные строку и столбец.
6. Напишите итеративные методы, которые вычисляли бы по квадратной матрице ее: (а) определитель $\det A$, (б) след $\text{tr}(A) = \sum_{i=1}^n a_{ii}$.
7. Напишите метод, возвращающий матрицу, обратную к квадратной матрице. Для вычисления обратной матрицы можно использовать метод присоединенной матрицы. Если $A^{-1} = (a_{lk}^{(-1)})_{l=1, k=1}^{n, n}$ — матрица, обратная к невырожденной матрице $A = (a_{lk})_{l=1, k=1}^{n, n}$, то $a_{lk}^{(-1)} = (-1)^{l+k} \frac{\det A_{kl}}{\det A}$.
8. Напишите метод, который реализует известный алгоритм Гаусса и возвращает решение системы линейных уравнений, если оно существует.
9. Квадратная матрица называется *симметричной*, если $a_{ij} = a_{ji}$ при всех допустимых значениях индексов. Из определения следует, что вся информация о симметричной матрице может храниться, например, только в элементах, расположенных на и выше ее главной диагонали. Создайте подкласс класса `NumericalMatrix` с именем `SymmetricMatrix`, экземпляр которого будет занимать значительно меньше места, и переопределите необходимые методы класса и экземпляра.
10. Пусть в $(m \times n)$ -матрице A выбраны произвольно k строк и k столбцов ($0 \leq k \leq \min(m, n)$). Элементы, стоящие на пересечении выбранных строк и столбцов, образуют квадратную матрицу порядка k , определитель которой называется минором k -го порядка матрицы A . Максимальный порядок r отличных от нуля миноров произвольной матрицы A называется ее *рангом*. Напишите метод `rank`, который вычисляет ранг матрицы¹.

¹ При реализации метода можно воспользоваться алгоритмом Гаусса.

11. Перепишите класс `NumericalMatrix`, делая его подклассом класса `NumberArray` и определяя в нем две именованные переменные экземпляра:

```
NumberArray variableSubclass: #NumericalMatrix
    instanceVariableNames: ' row column '
    classVariableNames: ' '
    poolDictionaries: ' '
```

В этом случае отличительной чертой экземпляра класса `NumericalMatrix` является то, что все его элементы расположены линейным образом в числовом массиве, строки и столбцы из которого выделяются в соответствии с задаваемыми размерами `row` и `column`. Имеет ли, на ваш взгляд, такая реализация преимущества перед реализацией, приведенной выше?

12. Напишите метод `determinantRec` рекурсивного вычисления определителя квадратной матрицы, который следует данному в начале главы определению определителя. Сравните его с итерационным методом.

ГЛАВА 15

ОСОБЕННОСТИ СОЗДАНИЯ КОДА

Ранее мы уже рассмотрели некоторые правила создания текстов на языке Смолток. Вероятно, читатель (особенно тот, который хорошо знает английский) заметил, что выражения на языке Смолток выглядят почти правильными предложениями английского языка. Это не случайно. Многие правила, которые мы рассмотрим ниже, обсуждая «технику кодирования», направлены на то, чтобы смолтоковский текст был «удобочитаемым».

Итак, «... *поговорим о странностях...*» создания программ на языке Смолток. Если вы примете к исполнению эти правила и соглашения, это поможет писать понятный и эффективно выполняемый код, который легко интегрировать в существующую библиотеку классов и использовать при решении других задач.

15.1. Что такое смолтоковский стиль

Программирование на языке Смолток сводится к расширению существующей библиотеки классов, которая разрабатывалась на протяжении многих лет (начиная с 1972 года) многими программистами. За это время появился некий «смолтоковский стиль» программирования. Большинство принципов, о которых пойдет речь, отражают те приемы, которые использовались опытными программистами и которые можно легко отыскать в библиотеке — «сокровищнице» разнообразных примеров.

Смолтоковский стиль — приемы и правила, позволяющие в процессе создания кода понятнее выразить заключенные в нем основные идеи. Это чрезвычайно полезно, прежде всего потому, что помогает пользователю понимать, что делает каждый конкретный класс или метод, только читая исходный текст.

Правило — не закон, поэтому и в библиотеке классов есть примеры того, как одна и та же идея реализуется совершенно разными способами. Но существенные отклонения от общепринятого стиля затрудняют понимание кода. Надеемся, что нам удастся убедить вас принять выработанный стиль, чтобы создаваемый вами текст был подобен лучшим смолтоковским образцам. Тогда его будет легко читать, он будет более понятен другим программистам, и, в случае необходимости, его легко будет изменить. Даже только это, при прочих равных условиях, может способствовать более частому использованию другими именно вашего кода.

15.2. Соглашение об именах

В языке Смолток существует совсем мало «законов» относительно имен для классов, переменных и методов (а также категорий и протоколов в тех реализациях, в которых они есть¹). Имеющиеся правила мы рассмотрели в первой части. Здесь же остановимся на соглашениях, которых следует придерживаться программистам, подбирая имена для своих классов, переменных, методов и других объектов.

Не забывайте, что имена общих переменных (глобальных переменных, переменных пула, переменных класса) начинаются с заглавной буквы, а имена временных переменных начинаются со строчной буквы. Кроме того, имена не могут содержать пробелов. Поэтому, если имя состоит из нескольких слов, последующее слово пишется без пробела с заглавной буквы.

Старайтесь делать имена переменных настолько описательными, насколько это возможно, тем более что имена могут быть достаточно длинными. Можете включить в имя информацию о назначении данной переменной. Например, `occupiedRectangle`, `lineDelimiter`, `nextFourBytes`. Последний пример использует общее соглашение об указании на набор объектов: используется существительное во множественном числе. Если надо дать имя, которое отражает логическое состояние, следует использовать нечто подобно `isNil`, `isChanged`, `wasConverted`, `hasBeenEdited`. Используя такие имена, вы сможете писать классические смолтоковские выражения:

```
occupiedRectangle hasBeenEdited ifTrue: [ "Do something" ].
```

Общее соглашение об именах параметров при определении метода сводится к использованию типа или имени класса того объекта, который должен передаваться, с приставкой 'a' или 'an' (в соответствии с правилами английского языка). Например, `anArray`, `aTriangle`. Если параметр метода может быть экземпляром нескольких разных классов, следует использовать самый «нижний» общий суперкласс, например, `aCollection` или, в самом общем случае, `anObject`. Последний пример не столь бессмысленен, как может показаться, поскольку указывает на то, что в качестве аргумента может передаваться экземпляр *любого* класса.

Так же тщательно, как имя переменной, следует выбирать и имя метода (селектор), чтобы оно описывало цель метода. На длину имени метода нет существенных ограничений. Методы, которые преследуют только цель обращения к переменным, должны называться по имени такой переменной. Например, если есть переменная экземпляра с именем `sideA`, метод, который возвращает ее значение, следует назвать `sideA`, а не `returnSideA`,

¹В некоторых новых реализациях используются пакеты и пространства имен.

`getSideA` или как-нибудь еще. Точно так же метод, который устанавливает значение этой переменной, следует назвать `sideA:`, а не `setSideA:`.

Старайтесь создавать описательные имена методов, а не повелительные. Например, лучше использовать имя `area` (площадь), а не `computeArea` (вычислитьПлощадь). В первом случае имя метода связывается с видом возвращаемого методом значения (объекта). Это очень важно, например, тогда, когда в выражении посылаются сразу несколько сообщений. Сравните следующие два выражения:

```
aTriangle area asSquareMetres.  
aTriangle computeArea convertToSquareMetres.
```

Может быть, различия между ними кажутся достаточно тонкими, но программистам на языке Смолток (и большинству непрограммистов) первое выражение намного понятнее, чем второе. Если вы хотите использовать слова подобные `compute` и `convert`, лучше используйте причастие прошедшего времени: `computedArea` (вычисленнаяПлощадь), `convertedToSquareMetres` (преобразованноеВКвадратныеМетры).

При наименовании метода, который имеет несколько параметров, старайтесь создавать имя, которое передает и цель, и, если возможно, тип каждого параметра. Например:

```
PaintBox drawLineFrom: x  
to: y  
usingPen: aPen  
inColour: #red.
```

Обратите внимание, что в этом случае мы отказались от правила о создании описательного имени метода и использовали `drawLineFrom:to:usingPen:inColour:` вместо имени `lineFrom:to:usingPen:inColour:.` Здесь мы больше заинтересованы в *побочном эффекте*, а не в возвращаемом значении.

И, наконец, при выборе имени метода думайте о той работе, которую выполняет метод, а не о том способе, которым работа выполняется. Другими словами, называйте метод *до*, а не *после* его реализации. Поступая так, вы избежите демонстрации способа реализации, который со временем вполне может измениться, а цели, ради, которых создавался метод, останутся. Но иногда, когда все же требуется подчеркнуть характер реализации метода, можно это сделать с помощью понятного суффикса (или приставки) к основному имени (например, `determinantRec`).

Теперь обратимся к именам классов. Как и прежде, основная цель при выборе имени должна состоять в сообщении цели существования данного класса. Однако можно постараться сообщить и нечто об иерархии классов, если вы считаете это достаточно важным. Например, `OrderedCollection` — подкласс класса `Collection`. Вы могли бы продолжить иерархию, создавая

подкласс с именем, скажем, `OptimizedOrderedCollection`. Обязательное следование такому соглашению о наименовании не всегда необходимо или желательно — все это вопрос вашего стиля и здравого смысла. Например, класс `Set` — тоже подкласс класса `Collection`, но в данном случае слово «Set» связывается с целью создания класса, и вряд ли здесь стоит еще что-то изобретать.

Категории классов и протоколов существуют не во всех реализациях языка. Но там, где они существуют, они вводятся только для того, чтобы помочь разумным образом организовать классы или методы. Следовательно, и использовать надо такие имена, которые облегчают решение этой задачи. В имена категорий можно включать пробелы.

Имена категорий, используемые в иерархии классов в системе VisualWorks, подобные `accessing`, `updating`, `displaying` и `private`, идеально вписываются в предложенный стиль. Поддерживая высказанные соглашения о наименованиях, используйте их. Однако мало только использовать такие имена, надо еще использовать их *правильно*. Не помещайте, например, в протокол `accessing` методов, отличных от методов, обеспечивающих доступ к переменным экземпляра. Если метод частный и предназначен для использования только разработчиками, помещайте его в категорию `private`. Хорошее правило, которому стоит следовать, состоит в том, чтобы использовать в качестве имени протокола причастие настоящего времени — слово, которое заканчивается на «-ing». Например, `calculating` (методы для вычислений) или `printing` (методы для печати).

15.3. Доступ к переменным экземпляра

Давайте посмотрим внимательно на то, как стоит обращаться к переменным экземпляра. Если класс определяет или наследует переменную экземпляра, то в методах, определенных в данном классе, к каждой такой переменной можно обращаться просто по имени. Этот прием применяется и для назначения значения переменной, и при использовании ее значения в различных выражениях. Например, внутри метода того класса, который имеет переменные экземпляра с именами `area`, `width` и `height`, допустимо следующее выражение: `area := width * height`.

Как правило, программист определяет еще и методы доступа, с помощью которых любые другие объекты, в том числе и сам объект, могли бы обращаться к переменным. Многие программисты рекомендуют, чтобы объект даже к собственным переменным обращался не по имени, а сообщением, посланным самому себе. Тогда приведенное выше выражение стало бы таким: `self area: self width * self height`.

Если методы `area`:, `width` и `height` определены корректно, это выражение эквивалентно первому. Ради чего стоит так поступать? Ответ состоит в следующем: избегая прямого доступа к переменным, мы тем самым потенциально увеличиваем гибкость и, как результат, расширяем возможности многократного использования созданного кода. Переменные экземпляра реально представляют собой характеристики (или свойства) объекта. Точнее, это такие свойства, значения которых хранятся в объекте и, как правило, не вычисляются. Однако впоследствии может возникнуть ситуация, в которой значение свойства должно стать вычисляемым. Если предположить, что к этой переменной экземпляра всегда обращались только посредством метода, задача замены метода доступа на новый становится очень простой. Достаточно только оставить то же самое имя метода, но изменить способ вычисления значения переменной. Если же к переменной обращались по имени, ситуация становится более сложной. Необходимо не только создать новый метод, но и отследить каждую прямую ссылку на переменную и заменить ее на сообщение, вызывающее новый метод.

Надеемся, преимущество методов доступа к переменным экземпляра через сообщения не вызывает сомнений. Есть ли у него недостатки? Есть. Наиболее очевидный — эффективность. Доступ к собственным переменным через сообщения обязательно медленнее, чем прямая ссылка на них. Сообщение доступа ищет ту же самую ссылку на переменную, но теперь еще тратится время на передачу сообщения. Доступ к переменным через сообщения может оказаться еще и менее понятен пользователю (хотя это вопрос привычки). И наконец, необходимость определять методы доступа даже для «частных» переменных, то есть для таких переменных, обращение к которым не предполагается извне экземпляров данного класса (независимо от того, были ли они определены или унаследованы), может выглядеть несколько неудобной. Но помните, что в системе Смолток реально нет ничего частного. В конечном счете, любой пользователь всегда сможет определить собственный метод доступа или воспользоваться методом `instVarAt`:, определенным в классе `Object`.

Общей рекомендации о том, как программировать доступ к переменным, дать нельзя. Программист сам должен решить, что для него важнее: эффективность при отсутствии гибкости или, наоборот, максимальная гибкость с некоторой потерей эффективности. Необходим поиск компромисса: на какие переменные стоит ссылаться непосредственно (и скрыть от пользователя), а для каких использовать методы доступа. Не забывайте, что в случае многократного обращения к некоторой переменной нужное значение можно сохранить во временной переменной, а не использовать каждый раз сообщение.

Перечисленные выше правила в равной степени применимы к переменным класса, глобальным переменным и константам. Если есть некоторая фундаментальная константа, важная для создаваемого класса, стоит поместить такую константу в метод и всегда пользоваться только им, а не вставлять такую константу непосредственно в методы. Например, число π (отношение длины окружности к ее диаметру) вычисляется с помощью выражения `Float pi`, возвращающего число 3.14159265.

15.4. О структурировании методов

Одна из основных задач, которую приходится решать при создании методов, состоит в необходимости написания небольших по объему методов. Метод, содержащий более десяти строк кода, уже надо рассматривать как большой. Надо стараться разбить текст длинного метода на два-три небольших. Наряду с использованием временных переменных для хранения результатов промежуточных вычислений такой прием делает код легким для чтения и понимания, изменения и повторного использования. Надо стараться делать вспомогательные методы настолько универсальными, насколько это возможно, а затем «составлять» из них функциональность, требуемую в основном методе.

Старайтесь при записи метода надлежащим образом форматировать код, отражая его логику. Продуманный формат текста метода делает код проще для понимания, отладки и изменения. Не стесняйтесь расставлять круглые скобки, если считаете, что это делает код более понятным.

Как мы знаем, методы автоматически возвращают получателя сообщения (`self`), если в теле метода явно не предусмотрено ничего другого. Однако если возвращение методом объекта `self` — важная черта метода, можно указать это явно, используя в качестве последнего выражения конструкцию `^ self`. Это же полезно сделать и тогда, когда в процессе выполнения метода достигается точка, в которой надо прекратить вычисления, и совершенно не важно возвращаемое значение. Возвращение объектом значения `self` позволяет посылать объектам цепочки сообщений. Не стоит возвращать особый объект, скажем, `nil`, поскольку такой особый объект следующего посланного сообщения почти наверняка не поймет.

Когда вы хотите использовать условное выражение (`ifTrue:`, `ifFalse:`, и т.д.), подумайте о том, нельзя ли получить тот же самый эффект, структурируя классы и методы так, чтобы нужная операция выполнялась как результат поиска метода по иерархии (то есть постарайтесь использовать полиморфизм). Особенно полезно поступать именно так, если перед принятием решения «*что делать?*» приходится проверять, *какому классу при-*

надлежит объект. Это один из признаков того, что создаваемые классы были не совсем точно спроектированы. Если сложилось совсем уж безвыходное положение, можно проверить, поймет ли конкретный объект конкретное сообщение, используя метод `respondsTo:`, определенный в классе `Object`. Но доводить дело до этого не стоит.

15.5. Использование комментариев

Как и другие языки программирования, язык Смолток дает возможность размещать в коде комментарии. Тщательно составленные комментарии значительно расширяют возможности пользователей в понимании кода (как, впрочем, и возможности авторов кода через несколько недель, месяцев или лет). Однако верно и обратное. Наличие непродуманных и неряшливых комментариев хуже, чем отсутствие комментариев вообще.

Все смолтоковские системы позволяют располагать комментарии внутри текста методов (а некоторые и в классах). Комментарии могут встраиваться в код метода, используя пару двойных кавычек ("`...`"). Хорошим стилем считается размещение комментария непосредственно в начале каждого метода. Такой комментарий должен описывать назначение метода, параметры, которые он принимает (хотя это должно быть очевидно, если тщательно подобраны имена метода и его параметров), какое значение возвращает метод и нечто специальное, если это нужно. При желании можно включить сюда информацию об авторе и времени создания метода.

В сложных случаях очень полезно, располагая комментарий в начале метода, поместить в него пример использования данного метода. Тогда любой человек, изучающий код, сможет с помощью мыши выбрать текст примера и выполнить его, чтобы увидеть, как он работает.

Дополнительные комментарии внутри метода должны сообщать о том, *что и почему* происходит, а не *как* это происходит. "Происходит увеличение индекса" — бесполезный комментарий для выражения `index := index + inc`. Фраза "Переход к следующему служащему" намного более информативна, если объект имеет дело со служащими. Приложив минимальные усилия, сам код на языке Смолток можно сделать понятным для читателя, что сделает ненужным слишком объемный и тривиальный комментарий.

Если позволяет реализация, можно поместить комментарий и в каждый класс, объясняя побудительные мотивы его создания и то, что он делает. Когда вы воспользуетесь комментариями классов системной библиотеки, пытаясь выяснить, что же делают классы системы, вы поймете, как будут благодарны вам за подобные комментарии пользователи ваших классов!

15.6. На что еще стоит обратить внимание

Обратимся к правилам кодирования, основанным на здравом смысле. Это означает, что приводимые далее советы, вообще говоря, полезны, но всегда будут возникать конкретные ситуации, в которых придется делать кое-что другое и совсем не так.

1. Если мы имеем дело с логическими истиной и ложью, то надо пользоваться только `false` и `true`, а не `0` и `1`, или `nil` с некоторым объектом, отличным от `nil`.
2. Если нужна переменная, которая принимает фиксированное число значений, постарайтесь закодировать такое значение, используя число или системное имя, а не строку (имена более эффективны, так как они уникальны и могут сравниваться с помощью сообщения `==`). Например, чтобы описать размер некоторого объекта, стоит использовать имена `#large`, `#medium` и `#small`, а не числа, скажем, `3`, `2` и `1`. Посмотрите на следующие три выражения:

```
MyObject size = 3 ifTrue: [ ... ].
```

```
MyObject size == #large ifTrue: [ ... ].
```

```
MyObject isLarge ifTrue: [ ... ].
```

Насколько проще понять то, что делает второе выражение, по сравнению с тем, что делает первое. Третье выражение еще проще, в нем предполагается, что написан метод с именем `isLarge`, который содержит в себе выражение типа `size == #large` и возвращает `true` или `false`.

3. Один из наиболее полезных строительных блоков — словари. Мы уже видели, как они используются в качестве простой структуры данных, хранящей пары, организованные как «имя» и «значение». Словари могут содержать любые объекты, что существенно расширяет возможности их применения. Например, словарь может использоваться как своеобразная управляющая структура, когда в качестве значений используются блоки. Такой словарь позволяет сконструировать, если нужно, своего рода «оператор `case`». Первое из приведенных ниже выражений показывает словарь, инициализированный подобным образом (инициализация должна произойти только один раз), а второе выражение показывает один из способов, которым такой словарь можно было бы использовать:

```
MyDict at: #Small put: [ "do a small thing" ];  
      at: #Medium put: [ "do a medium thing" ];  
      at: #Large put: [ "do a large thing" ].
```

```
(MyDict at: case) value.
```

4. Иногда бывает нужен только один объект с определенной структурой и поведением. Например, нужен объект, работа которого состоит в управлении некоторым уникальным ресурсом — файловой системой или таблицей поиска. При таких обстоятельствах соблазнительно сделать такой объект классом. Другими словами, кажется логичным создать специальный класс и написать методы класса, которые будут выполнять всю работу, особенно если объект нуждается в ярком, уникальном, хорошо понимаемом имени.

Однако такой способ создания подобного объекта — не самый лучший. Намного лучше — создать класс, и создать его единственный экземпляр, который и будет реально работать. Основная причина такого решения состоит в том, что никогда нельзя быть уверенным, что в один прекрасный день не потребуется создать еще один столь же уникальный экземпляр. Кроме того, при создании экземпляра ваш уникальный объект будет наследовать только функциональные возможности экземпляров, а не неподобающие ему функциональные возможности классов. Не забывайте, что наследование *только* нужных функциональных возможностей — один из тех принципов, которому надо неукоснительно следовать при проектировании классов.

В подобной ситуации один из практических способов решения состоит в том, чтобы создать класс и определить в нем переменную класса с именем `Default` (ЗначениеПоУмолчанию). Затем создать метод класса с именем `initialize`, который инициализирует эту переменную так, чтобы она содержала требуемый единственный экземпляр данного класса. В заключение остается создать метод класса с именем `default`, который будет возвращать значение переменной `Default`. После этого, когда в какой-то части кода надо использовать этот объект, достаточно вычислить выражение `MyClassName.default`. Теперь, если вдруг потребуется создать более одного экземпляра такого класса, не возникнет никаких проблем.

5. Стоит обратить внимание на отношение к коду, расположенному в иерархии классов. Предоставление любой системой Смолток почти полного исходного текста своей реализации позволяет изменять классы системы. Это очень мощное средство, и подобно всем таким средствам, к нему следует относиться с осторожностью. Абсолютно не верен постулат, что нельзя изменять классы из поставляемой иерархии. Если надо добавить новые функциональные возможности, это нужно делать. Такой вид изменений делает среду разработки более мощной, чем она есть.

Совсем другое дело, если планируются изменения в наиболее фундаментальных классах, типа классов `Object` или `Behavior`. В этом случае надо быть чрезвычайно осмотрительным. На этом пути есть две опасности. Во-первых, можно разрушить систему, и, во-вторых, можно произвести в ней изменения, несовместимые с изменениями, проведенными кем-то еще.

Прежде чем предпринимать подобное, подумайте, нельзя ли достичь того же эффекта без изменения системных классов; как предполагаемые изменения будут взаимодействовать с уже произведенными; как изменения отразятся на фундаментальном поведении системы. Если, принимая все это во внимание, вы отважитесь на изменения, предпримите все меры предосторожности. Начните работу с создания подкласса, и только после проверки перенесите изменения в системный класс.

6. В классе `Object` есть два мощных метода — `perform:` и `become:`, к которым надо относиться особенно внимательно. Изучая класс `Object`, мы уже рассматривали, как можно использовать метод `perform:`. Но помните: все последствия его применения очень трудно отследить.

Метод `become:` меняет значение объекта, получившего такое сообщение. В основном он используется в операциях изменения размеров наборов, переустанавливая каждую ссылку со старой версии набора на новую. Однако эта операция имеет в разных реализациях разную семантику. Например, в *Smalltalk-80* выражение `x become: y` вызывает все ссылки на `x` и `y` и обменивает их, а в *Smalltalk Express* не изменяет ссылок на объект `y`.

Если надо устранить все ссылки на объект `x`, выражение `x become: nil` прекрасно сработает в *Smalltalk Express*, а в *VisualWorks* еще вызовет каждую ссылку на `nil` и переставит ее на `x`. Это равносильно катастрофе! Если все же надо устранить все ссылки на объект `x`, безопаснее выполнить выражение `x become: String new`. Так что перед использованием таких методов стоит тщательно продумать все возможные последствия.

7. Как и во всех других языках программирования, эффективность написанного на языке Смолток кода варьируется достаточно широко. Говоря об «эффективности», мы предполагаем, что выбран правильный (наилучший) алгоритм², который выполняется быстро и/или использует меньший объем памяти.

Есть несколько правил, которые, если их придерживаться, позволяют сделать смолтоковский код достаточно быстрым. Почти все такие правила универсальны и относятся не только к языку Смолток.

Надо избегать некоторых операций, особенно тех, которые всегда являются неэффективными, независимо от того, какой язык программирования используется. Например, надо избегать предварительного вычисления таких значений, которые могут и не потребоваться. *Ленивые вычисления* (*Lazy evaluation*) — таково имя методики, которая откладывает вычисление

² Все вроде бы просто и понятно. Но задайте себе вопрос: «Стоит ли пожертвовать удобочитаемостью или возможностью многократного использования ради эффективности?»

некоторого значения до тех пор, пока оно фактически не потребуется. Однако если некоторое значение уже вычислили, не стоит выбрасывать его, если оно может снова потребоваться. Например, надо избегать многократного вычисления одного и того же значения внутри цикла, перемещая такие вычисления за цикл.

Есть некоторые операции, которые, как хорошо известно, являются в языке Сmolток неэффективными. Класс `Dictionary` медленнее, чем класс `IdentityDictionary`, поскольку использует сообщение `=` вместо более быстрого сообщения `==`. То же справедливо и для классов `Set` и `IdentitySet`. Механизм зависимости является более медленным способом связи между объектами, чем прямые ссылки. Таким образом, если быстроедействие важнее, то, добываясь его, надо быть готовым к тому, чтобы пожертвовать преимуществами механизма зависимости.

Распределение памяти всегда происходит медленно, так что не следует создавать по несколько раз большие объекты и тут же их отбрасывать. Помните, что метод `perform`: может оказаться относительно медленным, а метод `become`: крайне расточительным в отношении ресурсов.

Если возникают сомнения в эффективности созданного кода, стоит провести его полное исследование с помощью инструментов тестирования, хронометрирования и профилирования (их имеет каждая реализация). Они помогут выявить «узкие места». Если таких инструментов нет, то не забывайте, что время выполнения почти любого метода можно определить с помощью выражения `Time millisecondsToRun: aBlock`.

ЧАСТЬ IV

ПОСТРОЕНИЕ ПРИЛОЖЕНИЙ

Не для того я живу, чтобы есть,
а ем для того, чтобы жить.

Квинтилиан Марк Фабий

ГЛАВА 16

ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ

Среди рассмотренных нами классов системы *Smalltalk Express* еще не было классов, обеспечивающих построение интерфейса пользователя и составляющих одну из самых многочисленных и сложных частей иерархии классов. С сожалением надо отметить, что эти классы, как и классы, связанные с графикой, в каждой реализации весьма своеобразны, и потому приложения с интерфейсом пользователя не переносимы между ними. Такое положение сложилось в ходе развития смолтоковских систем. В 70–80-е годы, еще до широкого распространения доступных графических операционных систем, в первой смолтоковской системе — системе *Smalltalk-80* — в основе классов, обеспечивающих построение интерфейса пользователя, лежала триада «Модель — Вид — Контроллер» («Model — View — Controller», сокращенно MVC) [5], [13]. Позже в системе *Smalltalk/V for DOS* подход к построению интерфейса был несколько изменен, и, во избежание путаницы, соответствующая триада была переименована в «Модель — Панель — Диспетчер» («Model — Pane — Dispatcher» — MPD). И хотя вид и панель, контроллер и диспетчер — почти синонимы, набор классов пользовательского интерфейса в этих средах различается.

Когда на персональных компьютерах получили распространение оконные операционные системы, разработчики смолтоковских систем должны были найти способы приспособиться к ним. Каждая среда построения интерфейса должна определить структуру окна просмотра, соответствующую операционной системе, и добиться его полного взаимодействия с отобра-

жаемыми объектами, которые хранят данные и логику приложения. Совокупность таких объектов называют *моделью предметной области*. Вторая задача более трудная, поскольку сильно зависит от самой модели.

Для решения первой задачи было два пути: (1) вернуть полную функциональность триаде MVC, позволяющую создавать многоплатформенный пользовательский интерфейс, одинаково функционирующий под различными оконными системами; или (2) отбросить общие подходы и приспособить систему классов к особенностям конкретной оконной системы персональных компьютеров.

Фирма *Digitalk* выбрала второй путь, позволивший ей создать компактные и эффективные системы *Smalltalk/V*, тесно привязанные к операционным системам Windows и OS/2. Эти системы опираются на *сообщения* (в смысле Windows–OS/2) и поддерживают среду, которую, из-за отсутствия подходящего названия, часто называют программной средой, управляемой событиями. Чтобы работать в такой среде, необходимо: (а) понимать, что такое событие окна среды *Smalltalk Express*; (б) знать ту часть иерархии классов, которая связана с графическим интерфейсом; (в) иметь общее представление о функционировании MS Windows.

Мы предполагаем, что информация о функционировании Windows знакома читателю, и подробно эти вопросы далее рассматривать не будем.

16.1. Класс Window

Из всех объектов, заполняющих мир Windows, видимыми и, возможно, наиболее важными являются окна (*Window*) — «точки соприкосновения» приложения с внешним миром. Система *Smalltalk Express* задачу управления интерфейсом пользователя решает совместно с операционной системой, взаимодействуя через администратор окон *WindowManager*. Окном и его поведением управляют взаимодействующие между собой приложение и операционная система. Такие возможности заложены в экземпляры классов *ViewManager* (АдминистраторОкна) и *Window*.

Основными стандартными строительными блоками создаваемого интерфейса служат подклассы *Window*, в основном это классы *TopPane*, *SubPane* и их подклассы. Экземпляры класса *TopPane* используются при построении самого окна. Такая панель управляет содержащимися в нем многочисленными экземплярами подклассов класса *SubPane* (Подпанель). Данные подпанели и выполняют, в основном, всю работу по отображению информации, поставляемой объектами предметной области приложения. Строительные блоки из подклассов *SubPane* можно собирать не только в

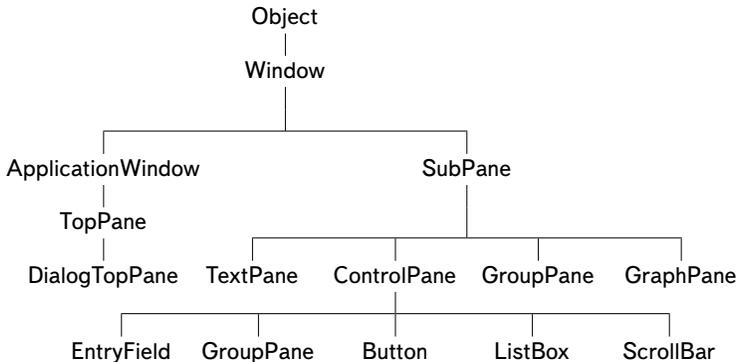


Рис. 16.1. Часть иерархии классов окон системы *Smalltalk Express*

окна, но и в сложные блоки, используя для этого экземпляры класса **GroupPane** (ГрупповаяПанель). Такие объекты хранят в себе экземпляры других подклассов класса **SubPane**, представляя их окну верхнего уровня как единое целое и тем самым облегчая разработку и реализацию сложных окон.

Часть иерархии классов, включающая основные классы панелей, показана на рис. 16.1.

Класс **Window** и его подклассы тесно связаны с операционной системой и ответственны за низкоуровневое взаимодействие с ее окнами. Сам класс **Window** — абстрактный класс, обеспечивающий общие данные и протокол, наследуемый всеми подклассами, представляющими окна MS Windows. В частности, они ответственны за обработку Windows-сообщений, понимаемых системой Смолток (экземпляр **Window** является объектной оболочкой функции окна Windows). Методы, обрабатывающие Windows-сообщения, по соглашению имеют имена, совпадающие с мнемоническим обозначением этого сообщения в файле заголовков `windows.h` или в заголовочном файле какой-либо подсистемы Windows. Например, для обработки сообщения `WM_INITMENU` вызывается метод `wmInitmenu: wordInteger with: longInteger`. Связь методов с сообщениями Windows устанавливается с помощью массива `WinEvents`, если код сообщения меньше 1024, и с помощью словаря `WinEventsExtra` — в противном случае. Эти наборы хранят селекторы методов, обрабатывающих сообщения Windows с кодами, равными индексу массива (ключу словаря). Если для какого-то сообщения Windows нет сопоставленного ему метода, то такое сообщение будет обрабатывать функция окна, предоставляемая системой Windows по умолчанию.

В реальной жизни создание нового подкласса класса **Window** имеет смысл только при создании новых панелей (или, по-другому, *виджетов*). Создание такого подкласса и/или расширение самого класса **Window** тре-

буется и в том случае, когда необходимо научить окна обрабатывать новые сообщения, связанные с реализацией доступа к новым программным интерфейсам (API), не предусмотренным в системе *Smalltalk Express*, например, к берклиевским гнездам (sockets). Отметим, что включению нового обработчика в массив `WinEvents` или в словарь `WinEventsExtra` должно предшествовать добавление обработчика в класс `Window` или в его подкласс.

16.2. Класс `ViewManager`

Как правило, при построении приложений (особенно если они не требуют специальных возможностей Windows) программист, помимо классов, описывающих объекты предметной области, должен создать подкласс класса `ViewManager`, связанный с приложением и визуально представляющий объекты приложения. Этот класс часто называют классом окна приложения¹. Стоит отметить, что классы модели предметной области «знают» только о тех аспектах решаемой проблемы, которые связаны со структурой и поведением сущностей, относящихся к предметной области, а не то, как эти сущности представляются в окне. А класс окна приложения создается главным образом для того, чтобы:

- запускать и завершать работу приложения;
- создавать, открывать и закрывать окно просмотра;
- координировать работу окна и отображаемой модели при интерактивном визуальном представлении объектов.

Обычно все связи между классами модели предметной области и классом окна приложения реализуются в последнем через соответствующие переменные экземпляра.

Сам класс `ViewManager` — абстрактный класс, реализующий общий протокол для той части приложения, которая отвечает за взаимодействие с пользователем (или шире — с внешним миром). Объекты предметной области приложения ответственны за предоставляемую интерфейсу информацию, а окно приложения ответственно за координацию между этой информацией и составляющими окно виджетами. Например, если произведен

¹ Помимо класса `ViewManager` в *Smalltalk Express* предусмотрена еще одна возможность создания приложения — включение нового класса в иерархию класса `ApplicationWindow`. Это подкласс в `Window`, который также предоставляет общий протокол для окон верхнего уровня. Но такое приложение имеет смысл создавать только в тех случаях, когда необходимо добиться глубокой интеграции с Windows и нет необходимости поддерживать сложную модель и многооконный интерфейс.

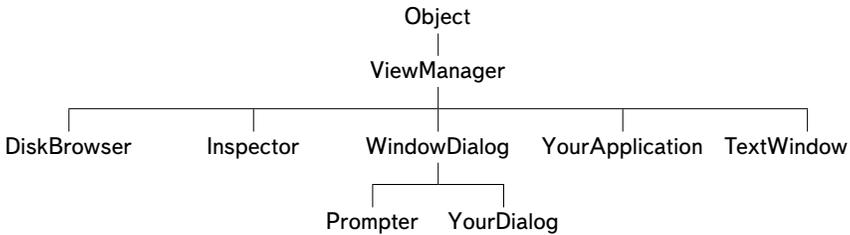


Рис. 16.2. Некоторые администраторы окон в системе *Smalltalk Express*

выбор записи в списковой панели, приложение может отреагировать на это, изменяя содержание текстовой панели, связанной со списком. Координация между внутренними панелями окна достигается через *события* окна среды *Smalltalk Express* — сигнал, который указывает на то, что произошло нечто, требующее внимания других объектов. Вот два простых примера: щелчок левой кнопкой мыши на кнопке (экземпляре класса **Button**) порождает событие **#clicked** (щелчок); выбор элемента в списковой панели генерирует событие **#select** (выбор). Для обработки событий все классы приложения должны действовать совместно, выполняя в ответ на события соответствующие им методы.

Перечислим основные функции, которые выполняют подклассы класса **ViewManager** и их экземпляры.

1. Управление самим окном приложения. Например, создавать и уничтожать окно, менять его свойства.

2. Создание панелей окна и управление ими. Приложение обычно запускается при посылке экземпляру класса окна приложения метода **open** или **openOn:**, который посылает ряд сообщений инициализации. Выбор между **openOn:** и **open** в значительной степени зависит от того, необходим ли методу, открывающему окно, аргумент. Метод, открывающий окно, инициализирует метку окна, создает само окно и все панели окна, определяя по мере необходимости для каждой панели следующие данные:

- владельца панели, обычно это сам экземпляр создаваемого подкласса класса **ViewManager**;
- выражение для вычисления рамки области, занимаемой панелью в окне приложения;
- имя панели окна;
- имя метода, создающего меню панели;
- имя одного или нескольких методов, которые нужно выполнять тогда, когда происходят события панели. Среди них *обязательно* должно быть имя обработчика события **#getContent**s.

3. Обеспечение содержимого панелей. Приложение должно предусмотреть для каждой панели метод, определяющий содержимое панели. Это обработчик события `#getContents`. По этому методу всегда можно определить панель, используя имя обработчика в сообщении `changed`: и позволяя приложению модифицировать панель.

4. Поддержка связей и синхронизации. Выбор или изменение содержащихся в панели данных может воздействовать на другие панели и на все приложение. Если, например, редактируется текст в текстовой панели, изменения текста локальны, так как не затрагивают другие панели и само приложение. Однако когда изменения сохраняются, текст нужно или откомпилировать в выбранном классе, или сохранить в файле, или поместить в файл регистрации изменений. Сохранить текст может только приложение, так как последствия сохранения уже глобальны.

Действия, приводящие к глобальным последствиям, требуют синхронизации. Важнейшую роль в синхронизации играет сообщение `when:perform:`. После `when`: указывается событие панели, а после `perform`: — селектор сообщения, посылаемого приложению как реакция на событие. Сообщение `when:perform`: посылается при создании окна приложения и определяет для каждой панели селектор сообщения, которое должно посылаться приложению тогда, когда в панели происходит указанное событие. Создавая метод-обработчик события, следует помнить, что единственным параметром такого метода должна быть сама панель.

Если требуется изменение других панелей, то для передачи соответствующим панелям сообщений могут применяться два механизма: явная передача, которая использует имя панели или саму панель, и передача через механизм зависимостей (сообщения `changed:`, `changed:with:`, `changed:with:with:`). В *Smalltalk Express* принято, что в классах интерфейса пользователя обычно первый аргумент этого сообщения является селектором обработчика события `#getContents` панели, которая должна обновиться.

5. Определение меню панелей. Когда во время создания панели ей посылается сообщение `when: #getMenu perform: #message`, в приложении должен быть определен метод с селектором `#message`. Этот метод должен устанавливать меню панели.

Часто в дополнение к классам окон приложения проектировщики должны создавать и диалоговые окна, которые позволяли бы обрабатывать в интерактивном режиме запросы пользователя к приложению и запросы приложения к пользователю. Для создания диалогового окна надо добавить новый подкласс в классе `WindowDialog` — подклассе класса `ViewManager`. Экземпляры `WindowDialog`, в отличие от окон приложений, являются модальными окнами, то есть приложение, его вызвавшее, не может продолжить

свою работу до тех пор, пока модальное окно не будет закрыто и не обеспечит приложение нужной информацией. Примером широко используемого подкласса `WindowDialog` является рассмотренный в 3.1.4 класс `Prompter`. Кроме того, диалоговые окна могут создаваться и как подклассы в классе `DialogBox`, который сам является подклассом класса `Window`. Также часто используются экземпляры класса `MessageBox`.

Но вернемся к главной теме — окнам приложений. Окно приложения имеет структуру, необходимую для отображения свойств объектов предметной области. Но окно не должно реализовывать свойства и логику предметной области. Окно должно быть именно интерфейсом между предметной областью и пользователем. При таком подходе модель предметной области отделяют от внешнего интерфейса — от окна приложения, которое обеспечивает взаимодействие с моделью. Такое разделение помогает создавать приложения любой сложности, компоненты которых могут повторно использоваться и в других приложениях.

16.3. Панели и события

Чтобы создаваемый для приложения подкласс класса `ViewManager` мог открыть окно, необходимо включить в него метод экземпляра `open`, который полностью сгенерирует внутреннюю структуру окна приложения и в заключение пошлет себе сообщение `openWindow`, отображая окно на экране. В среде, управляемой событиями, метод `open` будет выглядеть так:

open

```
"Открыть окно просмотра для приложения."
self labelWithoutPrefix: 'Sample Application'.
self addSubpane: (PaneClass new          "Добавить в окно панель."
                  ... specifications ...
                  yourself).
self addSubpane: ...
...
self addSubpane: ...
self openWindow.                        "Отобразить окно."
```

Ниже мы построим простой пример — приложение `PhoneBook` и внимательно рассмотрим все особенности построения метода `open`, но сначала ближе познакомимся с некоторыми «строительными блоками» интерфейса, а именно с панелями и с событиями, происходящими в панелях, с основными сообщениями, входящими в протокол классов панелей.

16.3.1. События панелей

Каждая встроенная в окно панель генерирует свой набор смолтоковских событий (не путайте их с сообщениями, генерируемыми операционной системой, с которыми событие системы Смолток связывается благодаря классу `Window`). Событие — не объект. Каждое событие представляется именем (экземпляром класса `Symbol`), например, `#clicked`. Чтобы получить весь набор событий, о которых сообщает панель, достаточно классу, экземпляром которого является данный объект, послать сообщение `supportedEvents`. Но есть события, которые генерируют почти все такие объекты. Вот список наиболее важных событий вместе с ожидаемыми ответами приложения.

События класса `TopPane`

`#opened` Событие возникает при открытии окна приложения прежде, чем реальное окно появится на экране. В ответ на это событие приложение формирует окно и все его панели, выполняет необходимую инициализацию, а затем отображает окно на экране.

`#close` Событие возникает как результат попытки закрыть окно приложения. В ответ приложение может потребовать подтверждения на закрытие окна, и если оно будет получено, сначала завершает всю свою работу, после чего выполняет выражение `self close`, чтобы действительно закрыть окно.

События класса `SubPane`

`#getContents` Событие возникает каждый раз, когда панели посылается модификационное сообщение `update`. Оно обязательно возникает, когда первоначально открывается окно приложения: когда сообщение `open` посылается приложению, для всех панелей окна генерируется событие `#getContents`. В ответ приложение предпринимает необходимые действия, чтобы заново отобразить содержимое каждой панели. Например:

- установить радиокнопку так, чтобы она имела значение "on" (включено);
- обеспечить списковую панель списком, который нужно отобразить на экране, и элементом, который должен быть выбран;
- обеспечить текстовую панель текстом, который нужно отобразить на экране.

`#getPopupMenu` Событие возникает тогда, когда пользователь пытается обратиться к всплывающему меню панели. В ответ приложение должно установить в панель всплывающее (контекстное) меню.

Теперь перечислим события наиболее часто используемых подклассов класса `SubPane`. Некоторыми из них мы воспользуемся ниже при построении примеров.

События кнопки `Button`

`#clicked` Событие возникает, когда кнопка нажата.

События панели `TextPane`

`#save` Событие возникает при выборе пункта `save` во всплывающем меню панели. В ответ приложение получает текст из текстовой панели и обрабатывает его, после чего сообщает об этом текстовой панели (используя сообщение `modified: false`), чтобы она знала, что содержащийся в ней текст сохранен.

События панели `ListPane`

`#select` Событие возникает, когда пользователь любым образом выбирает элемент из списка.

`#doubleClickSelect` Событие возникает тогда, когда пользователь производит двойной щелчок левой кнопкой мыши на элементе списка.

События панели `GraphPane`

`#button1Down` Нажата левая кнопка мыши.

`#button1DownShift` Нажата левая кнопка мыши и клавиша [**↑ Shift**].

`#button1Up` Отпущена левая кнопка мыши.

`#button1DoubleClick` Двойной щелчок левой кнопкой мыши.

`#button1Move` Перемещение мыши с нажатой левой кнопкой.

`#button2Down` Нажата правая кнопка мыши.

`#button2Up` Отпущена правая кнопка мыши.

`#button2DoubleClick` Двойной щелчок правой кнопкой мыши.

`#button2Move` Перемещение мыши с нажатой правой кнопкой.

`#mouseMove` Перемещение мыши.

Приложение может некоторым образом отреагировать на движение мыши или нажатие ее кнопки. Во время последнего события можно выяснить текущее положение в координатах панели курсора мыши, посылая панели сообщение `mouseLocation`.

16.3.2. Протоколы классов панелей

Знания о происходящих в панелях событиях не достаточно для создания работающего приложения. Желательно еще знать основные сообщения, посылаемые панелям. Приведем здесь наиболее часто используемые.

Класс `TextPane` Протокол экземпляра

`contents` Возвращает строку, отражающую содержимое панели.

`contents: aString` Задает содержимое получателя равным строке `aString`.

`modified: aBoolean` Устанавливает значение переменной экземпляра `modified` равным `aBoolean`. Если в текстовой панели сделаны изменения ее содержимого, значение переменной `modified` автоматически устанавливается равным `true`.

Класс `ListPane` Протокол экземпляра

`contents` Возвращает массив (или упорядоченный набор) строк, представляющих содержимое панели.

`contents: anIndexedCollection` Устанавливает содержимое приемника равным `anIndexedCollection`, элементы которого должны преобразовываться (с помощью `printString` или другого сообщения) в экземпляры классов `String` или `Symbol`.

`selection` Возвращает для выбранного в списковой панели элемента его индекс.

`selectedItem` Возвращает для выбранного в списковой панели элемента определяющую его строку (или символ); если ничего не выбрано, возвращает `nil`.

`selection: aStringOrNil` По передаваемой в качестве аргумента строке определяет элемент, который будет выбранным (выделенным) элементом панели; при аргументе, равном `nil`, ничего не выделяется.

Класс `Button` Протокол экземпляра

`contents` Возвращает строку, которая является меткой (`label`) кнопки.

`contents: aString` Устанавливает метку кнопки.

16.3.3. Механизмы управления событиями

Сами панели, расположенные в окне приложения, не могут разумно отреагировать на возникающие в них события, поскольку не имеют ни малейшего представления о связанной с ними информации из предметной области. Поэтому каждая панель *имеет владельца*, которым, как правило, является содержащее ее окно приложения. Владелец через свои переменные «знает», какие объекты предметной области и каким образом должны

отображаться в окне. Если панель нуждается в некоторой информации, относящейся к приложению, она запрашивает ее косвенно через своего владельца, опираясь на механизм событий. Связывание события с нужным методом происходит при формировании окна и описывается в методе `open` выражением вида

`aPane when: #eventSymbol perform: #eventHandlerSelector.`

Здесь `aPane` — тот виджет (панель) окна, в котором произошло событие. Для обработки события класс окна приложения должен содержать метод экземпляра (*обработчик события*) вида

eventHandlerSelector: aPane

"Обработчик события #eventSymbol."

Событие окна системы *Smalltalk Express* генерируется виджетом и тогда, когда ему требуется некоторая информация от приложения, и тогда, когда требуется сообщить приложению о том, что произошло нечто с самим виджетом. Для того чтобы сообщить владельцу виджета о произошедшем событии, виджет может посылать себе сообщение вида `aPane event: #eventSymbol`, в ответ на которое владелец виджета выполнит конкретный метод, связанный с указанным событием. Если в приложении событие с именем `#eventSymbol` связано с помощью сообщения `when:perform:` с обрабатывающим его методом, имеющим имя `#eventHandlerSelector.`, то в ответ на произошедшее выше событие владелец виджета получит и выполнит сообщение

`perform: #eventHandlerSelector: with: aPane.`

Остается ответить еще на один вопрос: как окно приложения в коде, описывающем его реакцию на событие, может связываться с другим виджетом окна, кроме виджета `aPane`, в котором произошло событие, например, для того, чтобы или получить от него, или передать ему некоторую информацию.

Как мы уже отмечали, есть два механизма для связи между панелями окна: явный, использующий приказ панели модифицировать себя, и неявный — через механизм `change/update`. Явная передача сообщений требует панели или имени панели. Механизм `change/update` сам ответственен за поиск панели по указанному в этом сообщении обработчику события `#getContents` и за запуск на выполнение окном приложения этого обработчика. Сравнение между двумя подходами проведено в табл. 16.1, в которой сообщения слева и справа дают один и тот же результат, а под именем `#handler:` скрывается обработчик события `#getContents` панели `aPane`.

К виджетам можно обращаться и по их именам, если таковые были определены. Чтобы панель окна `aPane` получила имя, ей в методе `open` надо послать сообщение вида `aPane paneName: 'aName'`, где `'aName'` —

Таблица 16.1. Способы отправки сообщений виджетам

Прямое сообщение	Механизм change/update
aPane update	self changed: #handler:
aPane selector	self changed: #handler: with: #selector
aPane selector: parameter	self changed: #handler: with: #selector with: parameter

строка, идентифицирующая панель. Если панель названа, она может быть вызвана окном приложения (self) посредством отправки сообщения

self paneNamed: 'aName'.

В самой системе *Smalltalk Express* все время используется смесь из двух описанных выше механизмов, а обращения к виджетам по именам почти нет: такая возможность — относительно недавнее добавление. Инструменты, такие как браузеры иерархии классов, браузеры дисков и отладчики, были созданы намного раньше появления такой возможности и используют механизм change/update.

16.4. Пример: телефонная книга

16.4.1. Постановка задачи

Чтобы сказанное стало понятнее, объясним все еще раз при построении приложения PhoneBook (ТелефоннаяКнига)². Постановка задачи следующая: есть много людей, которым я регулярно звоню по телефону, и я хочу создать приложение, которое

- будет отображать список имен людей;
- список имен будет сортироваться в алфавитном порядке;
- при выборе из списка имени должен отображаться номер телефона;
- имена и номера телефонов могут добавляться и удаляться из списка.

Размышляя над проблемой, можно возвращаться к постановке задачи, изменять ее, добиваясь наилучшего решения. Цель должна состоять в краткой и понятной формулировке задачи, что потом позволит правильно опре-

² Такой пример приводится в файле chapter.11, поставляемом вместе с системой *Smalltalk Express*. В нашем учебнике он изменен.

делить тот способ ее решения, который в наибольшей степени соответствует предъявляемым к приложению требованиям.

Опираясь на формулировку задачи, представим, как должно выглядеть окно приложения. Нам кажется, что окно телефонной книги должно состоять, во-первых, из списковой панели, отображающей список имен, в которой можно выбрать одно из них, можно добавить новое имя и можно удалить имеющиеся. Во-вторых, нужна текстовая панель, в которой в ответ на сделанный в первой панели выбор будет отображаться соответствующий телефонный номер (если он есть), который можно здесь же отредактировать или ввести, если его еще нет. По этому «словесному эскизу» полезно создать рисунок интерфейса пользователя, который послужит отправной точкой для определения остальной части приложения. С каждой из панелей можно связать определенные действия, которые полезно представить как пункты меню, выполняющие роль посредников, используемых пользователем для того, чтобы посылать приложению необходимые сообщения.

Итак, мы преобразовали задачу в «словесный эскиз» того конкретного окна, которое надо будет сформировать, и по этому «эскизу» создали рисунок интерфейса пользователя.

Зная все, что нужно, о решаемой задаче и имея в наличии эскиз интерфейса пользователя, можно определить классы объектов, которые реализуют приложение. Как мы уже говорили, для окна приложения нужен новый подкласс класса `ViewMapper`, назовем его `PhoneBook` (ТелефоннаяКнига), экземпляры которого и будут являться полнофункциональными «телефонными книгами». Поскольку в вашем окне приложения появятся меню и два типа панелей, нам понадобятся экземпляры классов `Menu`, `ListPane` и `TextPane`. После некоторых раздумий, можно прийти к выводу, что словарь (экземпляр класса `Dictionary`) более всего подходит в качестве модели предметной области. С его помощью мы установим связи между именами людей и номерами их телефонов. Сами имена и номера телефонов будем представлять как строки (экземпляры класса `String`). Задача очень простая и для ее решения другие классы не нужны.

Выбранные нами классы отражают естественное представление о решении задачи. Но как быть, если выбор не столь очевиден? Важно сделать некоторый выбор, даже если он и не самый лучший. Можно ошибиться, но в среде, которая легко позволяет проводить изменения, лучшее решение можно очень быстро создать позже. Любой прогресс в разработке приложения расширит понимание того, какие классы и как следует использовать.

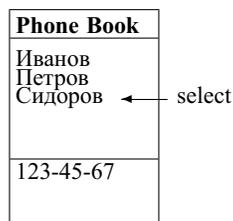


Рис. 16.3. Внешний вид окна `PhoneBook`.

Когда задача сформулирована, есть эскиз окна и выделены используемые классы, нужно определить необходимые переменные в новых классах. В данном примере единственный новый класс — класс `PhoneBook` — является визуализирующим классом для предметной области, представленной словарем. Его экземпляр нуждается в переменных, содержащих конкретные данные. Определим следующие переменные:

- `phones` — словарь, содержащий имена и их телефонные номера;
- `selectedName` — выбранное из списка имя.

Следовательно, определение нового класса может быть таким:

```
ViewManager subclass: #PhoneBook
instanceVariableNames: 'phones selectedName'
classVariableNames: ''
poolDictionaries: ''
```

Есть ли общие правила, позволяющие решить, какие переменные экземпляра следует использовать? Самый лучший способ — обращение к интерфейсу объекта. Какие сообщения будут посылаться объекту? Каким будет поведение объекта? Ответы на эти вопросы подскажут, какие переменные потребуются. Всегда полезно перечислить все те сообщения, которые должны реализовывать новые классы. Хорошая отправная точка для этого — расширение «эскиза окна» посредством внесения в него тех сообщений, которые будут использоваться каждой панелью. В нашей ситуации будут нужны сообщения для: инициализации переменных; обращения к меню, определяемому приложением; доступа к содержимому панелей в целях их отображения; сохранения содержимого измененных панелей; выборки элементов из конкретного меню; открытия окна. Таким образом, надо будет реализовать следующие сообщения:

`setDictionary` — инициализировать переменные экземпляра;

`open` — открыть окно телефонной книги;

`add` — добавить новое имя в телефонную книгу;

`remove` — удалить выбранное имя из телефонной книги;

`list: aListPane` — заполнить панель `aListPane` именами людей из словаря;

`nameSelected: aListPane` — отобразить номер телефона для имени, выбранного в `aListPane`;

`listMenu: aListPane` — определить меню для `aListPane`;

`text: aTextPane` — установить содержимое панели `aTextPane` в виде номера телефона для выбранного имени;

`textFrom: aTextPane` — получить содержимое текстовой панели в виде номера телефона для выбранного имени и занести его в словарь.

Теперь наш проект полностью описан и единственное, что остается — реализовать методы для перечисленных сообщений. Важнейший из них — метод, открывающий окно (в нашем случае метод `open`). С него и начнем, по ходу дела изучая возможности среды, управляемой событиями. Но прежде напишем единственный в нашем случае метод класса `new`, создающий его новый экземпляр, и единственный частный метод среди методов экземпляра (`setDictionary`), инициализирующий переменные экземпляра.

! PhoneBook class methods !

new

"Создает новый экземпляр класса."

^ super new setDictionary.

! PhoneBook methods !

setDictionary

"Инициализирует словарь."

phones := Dictionary new.

16.4.2. Метод `open`

Сначала полностью приведем текст метода экземпляра `open` нашего приложения, а затем подробно опишем, что и почему метод делает:

open

"Создает окно просмотра телефонной книги, определяя размеры панелей и их поведение; затем открывает окно."

self label: 'Phone Book'.

self addSubpane: (ListPane new owner: self;
 when: #getContents perform: #list; ;
 when: #select perform: #nameSelected;;
 when: #getMenu perform: #listMenu;;
 framingRatio: (0@0 rightBottom: 1 @ 0.8)).

self addSubpane: (TextPane new owner: self;
 when: #getContents perform: #text;;
 when: #save perform: #textFrom;;
 framingRatio: (0 @ 0.8 rightBottom: 1 @ 0.2)).

self openWindow.

Псевдопеременная `self`, как всегда, ссылается на получателя сообщения `open`, то есть на экземпляр класса `PhoneBook`. Первое, что происходит в методе, — объекту сообщают, что надо установить метку окна равной строке `'Phone Book'`. Затем в окно добавляются (`addSubpane:`) две панели: экземпляры классов `ListPane` и `TextPane`. Последняя строка метода посылает

объекту сообщение `openWindow`, реализованное в классе `ViewManager`, что приводит к созданию окна и его появлению на экране.

Посмотрим на определение панелей окна, например, на выражение `ListPane new framingRatio: (0@0 rightBottom: 1 @ 0.8)`.

Именно здесь определяется, какую часть окна будет занимать создаваемая панель. В главе, посвященной графике, мы уже создавали окна, и там исходный размер окна определялся посредством посылки сообщений типа `frame: (100 @ 100 extent: 400 @ 200)` (что, впрочем, не совсем корректно, поскольку при этом никак не учитываются реальные размеры экрана). Но когда создается панель внутри окна, размеры которого могут изменяться пользователем, занимаемая панелью часть окна должна определяться относительно размеров самого окна, то есть относительно исходного прямоугольника. Так, выражение с сообщением `framingRatio`: указывает, что создаваемая панель должна занимать все окно в горизонтальном направлении и 0,8 окна в вертикальном. В качестве альтернативы размещение панели может определяться с помощью блока, в котором абсолютные размеры панели вычисляются через абсолютные размеры прямоугольника всего окна. Например, выражение

```
ListPane new framingBlock: [:box |
    Rectangle leftTop: box leftTop extent: box extent * (1 @ 0.8)])
```

определяет то же расположение панели в окне, что и предыдущее (здесь `box` — прямоугольник окна).

Сообщение `owner: self` информирует панель о ее подконтрольности содержащему ее окну, или, другими словами, информирует панель о том, что окно является ее владельцем. Это означает, что панель может посылать сообщения «в» и получать сообщения «из» окна приложения.

Оставшиеся сообщения имеют одно и то же имя, но разные аргументы. Его шаблон нам знаком — `when: #eventSymbol perform: #eventHandlerSelector:`. Как мы знаем, такое сообщение информирует окно о том, что когда в данной панели происходит событие, описываемое символом `#eventSymbol`, владельцу панели, в данном случае экземпляру класса `PhoneBook`, нужно послать сообщение `eventHandlerSelector:` (обработчик события). Такое сообщение должно иметь ровно один аргумент — *панель, в которой произошло событие*.

Рассмотрим каждое из таких сообщений и реализуем указываемые в них методы. Начнем с сообщения `when: #getContents perform: #list:`, посылаемого списковой панели. Оно означает, что когда произойдет событие `#getContents`, необходимо получить новое содержимое панели и отобразить его. Для этого списковая панель должна послать своему владельцу сообщение `list: aPane`, в ответ на которое и будет выполнена указанная

работа. Поскольку моделью приложения у нас является словарь `phones`, а отображать в списковой панели надо ключи словаря, обработчик данного события должен выглядеть так:

list: aListPane

"Отображает в списковой панели `aListPane` имена людей из телефонной книги."

`aListPane contents: phones keys asSortedCollection.`

Аналогичное сообщение `when: #getContents perform: #text:` посылается текстовой панели. Метод `text:` должен отображать в текстовой панели номер телефона лица, выбранного в списковой панели, и ничего не отображать, если выбор не был сделан.

text: aTextPane

"Определяет содержимое текстовой панели как номер телефона человека, выбранного в списковой панели."

`aTextPane contents: (phones at: selectedName ifAbsent: [String new]).`

Реализация обработчика события `#save` текстовой панели должна сохранить новое содержимое текстовой панели в словаре `phones` в качестве значения для выбранного в списковой панели ключа (лица) и сообщить текстовой панели, что ее содержимое сохранено (для этой цели текстовые панели имеют переменную `modified`).

textFrom: aTextPane

"Сохраняет в словаре `phones` содержимое `aTextPane` как номер телефона выбранного в списковой панели имени."

`selectedName isNil ifTrue: [^true].`

`phones at: selectedName put: aTextPane contents.`

`aTextPane modified: false.`

Обработчик события `#select` списковой панели должен отображать в текстовой панели номер телефона выбранного лица. Следовательно, между этими двумя панелями должна быть установлена связь. Для этого используется механизм поиска панели по обработчику события `#getContents`, то есть используется сообщение `changed: #text:`, которое отыщет в окне просмотра панель с методом `#text:` в качестве обработчика события `#getContents` и заставит владельца панели выполнить этот метод.

nameSelected: aListPane

"Отображает в текстовой панели номер телефона в соответствии с выбором в списковой панели."

`selectedName := aListPane selectedItem.`

`self changed: #text:.`

Теперь напишем метод, создающий и включающий в окно специальное меню для списковой панели, и методы обработки для каждого из пунктов этого меню. Начнем с метода для построения меню.

listMenu: aListPane

"Определяет всплывающее меню для панели aListPane."

```
aListPane setMenu: ((Menu
    labels: '~Add\ ~Remove' withCrs
    lines: Array new
    selectors: #(add remove))
    title: '~Phones').
```

Текст метода `listMenu:` требует пояснений, так как построение меню еще не рассматривалось. Прежде всего заметим, что меню в этом методе создается посылкой классу `Menu` сообщения `labels:lines:selectors:.` Элементы меню определяются строкой-аргументом ключевого слова `labels:` и отделяются друг от друга символом `\`, который заменяется символом возврата каретки (благодаря сообщению `withCrs`, посылаемому строке). Символ `~` объявляет следующий после него символ подчеркнутым и одновременно определяет его как горячую клавишу. Массив-аргумент ключевого слова `lines:` — либо пустой массив, либо содержит номера строк меню, после которых надо провести в меню разделительную горизонтальную линию. Селекторы, перечисленные в массиве после ключевого слова `selectors:`, — имена методов, которые будут выполняться при выборе соответствующего пункта меню. Сообщение `labels:lines:selectors:` всегда посылается владельцу меню — экземпляру класса `PhoneBook`.

Сообщение `title:` со строкой в качестве аргумента посылается уже созданному меню и определяет имя данного меню, под которым оно устанавливается в строку меню владельца панели. Тем самым доступ к меню становится возможным и через строку меню окна приложения, и через событие `#getMenu`, возникающее при нажатии в списковой панели правой кнопки мыши. Если в подобном методе заголовок меню не определяется, *Smalltalk Express* создаст его сам в виде `'Untitled'` и добавит меню с таким именем к строке меню, напоминая программисту, что необходимо обеспечить более описательный заголовок.

Наконец, чтобы сделать такое меню работающим, реализуем в классе `PhoneBook` следующие два метода:

add

"Добавляет новое имя в телефонную книгу."

```
| key |
```

```
self textModified ifTrue: [^ self].
```

```
key := Prompter prompt: 'Enter new name:' default: String new.
```

```
(key isNil or: [phones includesKey: key]) or: [key = ""] ifTrue: [^self].
selectedName := key.
phones at: key put: nil.
self changed: #list;
    changed: #list: with: #selection: with: key;
    changed: #text:.
```

remove

"Удаляет выделенное имя из телефонной книги."

```
phones removeKey: selectedName ifAbsent: [ ].
selectedName := nil.
self changed: #list; changed: #text:.
```

Здесь есть новые моменты. Прежде всего, обратим внимание на выражение `self textModified ifTrue: [^self]` из метода `add`, которое проверяет, остались ли в окне текстовые панели, содержимое которых не было сохранено. Самое интересное для нас в методе `add` — второе из сообщений с ключевым словом `changed:`. Благодаря ему по `#list:` будет найдена панель, которая понимает этот обработчик, и ей, а не владельцу панели, будет послано сообщение `selection: key`, выделяющее в списке введенное имя.

В методе `remove` нет ничего для нас нового. Обратим внимание только на то, что после удаления выделенного имени из списка новое выделенное имя не задается (`selectedName := nil`).

Построение завершено. Отметим, что все использованные нами в окне приложения панели (виджеты) являлись экземплярами подклассов класса `SubPane`. И хотя каждый виджет имеет уникальные характеристики, обрабатывает события и выполняет сообщения присущим только ему способом, все они обладают следующими общими свойствами:

- Имеют владельца, обычно подкласс класса `ViewManager`.
- Сообщения `framingBlock:` (или `framingRatio:`) определяют область, занимаемую панелью в окне.
- Имеют набор событий, о которых сообщают владельцу; связь между конкретным событием и сообщением устанавливается посылкой себе сообщения `when:perform:`.
- Устанавливают свое содержимое, сообщая владельцу о событии `#getContents`; имя обработчика события `#getContents` определяет панель при посылке владельцу `changed:-`сообщений.
- Могут устанавливать меню, сообщая владельцу о событии `#getMenu`.

Теперь, чтобы начать работу с этим приложением, можно, например, определить глобальную переменную `MyPhoneBook`, вычисляя выражение `Smalltalk at: #MyPhoneBook put: PhoneBook new`, а затем открыть эту кни-

гу для работы, вычисляя выражение `MyPhoneBook open`. Чтобы сохранить всю введенную в телефонную книгу информацию, необходимо сохранить образ системы при выходе из нее.

16.5. Пример: переводчик

16.5.1. Постановка задачи

Чтобы рассмотреть новые моменты в построении приложения, приведем пример, в котором реализуем простое окно для «перевода» слов с английского языка на русский³. Постановка задачи почти очевидна: построить приложение, которое позволяет выбирать в любой из двух панелей — панели английских слов и панели русских слов — существующее слово и получать в другой панели его перевод, если он был введен в приложение.

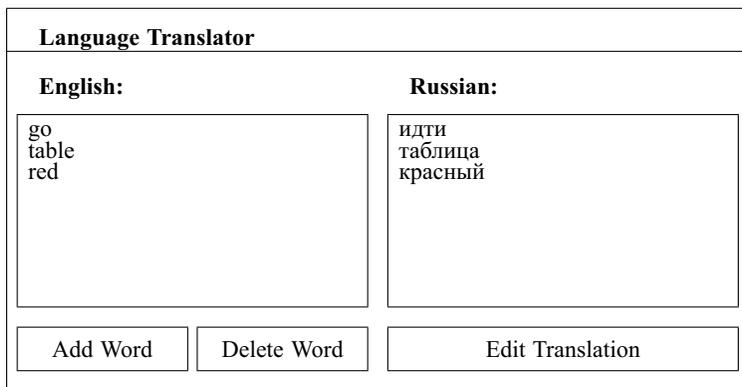


Рис. 16.4. Внешний вид окна переводчика

При реализации этого приложения, в отличие от предыдущего, мы будем использовать кнопки, а не меню, для чего определим две списковые панели и три кнопки. Выше каждой панели расположим метку. Левая списковая панель с меткой **English:** — панель английских слов (ключей словаря), в то время как правая панель с меткой **Russian:** — панель русских слов (значений словаря). Слова будем представлять экземплярами класса `String`. А моделью предметной области вновь определим экземпляр класса `Dictionary`. Внешний вид окна переводчика, соответствующий этому описанию, приведен на рис. 16.4.

³ Пример представляет собой переработку примера из [12, гл. 5]

Полное состояние приложения поддерживается переменными экземпляра. Нам потребуются всего две переменные:

`translation` — словарь с английскими словами в качестве ключей и их русским переводом в качестве значений;

`englishSelection` — выбранное в панели английское слово, или `nil`, если выбор не был сделан.

Опишем функциональные возможности приложения. В окне, если выделяется слово в панели английских слов, соответствующий ему перевод должен выделяться в панели русских слов, и наоборот. Новые слова могут добавляться и удаляться из английской панели с помощью кнопок `'Add Word'` и `'Delete Word'` соответственно. Перевод английского слова может быть отредактирован при нажатии на кнопку `'Edit Translation'`. Если в приложении происходят изменения — произведен выбор в одной из панелей, введено или удалено английское слово, отредактирован перевод — надо вызывать соответствующий обработчик, который должен гарантировать, что окно приложения в соответствии с происшедшими изменениями будет обновлено. В табл. 16.2 перечислим все события окна и его виджетов вместе с именами обработчиков событий.

Таблица 16.2. События окна `LanguageTranslator`

Элемент интерфейса	Событие	Имя обработчика
Окно <code>LanguageTranslator</code>	<code>#opened</code>	<code>open</code>
Панель <code>englishPane</code>	<code>#getContents</code> <code>#select</code>	<code>updateEnglishList:</code> <code>selectEnglishItem:</code>
Панель <code>russianPane</code>	<code>#getContents</code> <code>#select</code>	<code>updateRussianList:</code> <code>selectRussianItem:</code>
Кнопка <code>addWordButton</code>	<code>#clicked</code>	<code>clickedAddWord:</code>
Кнопка <code>delWordButton</code>	<code>#clicked</code>	<code>clickedDeleteWord:</code>
Кнопка <code>editTranslationButton</code>	<code>#clicked</code>	<code>clickedEditTranslation:</code>

По созданному эскизу очевидно, как определить новый класс. Назовем его `LanguageTranslator` и приведем определение класса вместе с методом класса `new` и частным методом экземпляра `setDictionary`, которые необходимы для создания нового экземпляра.

```

ViewManager subclass: #LanguageTranslator
  instanceVariableNames: 'translation englishSelection'
  classVariableNames: ''
  poolDictionaries: ''

```

! LanguageTranslator class methods !

new

"Создает новый экземпляр."

^ super new setDictionary.

! LanguageTranslator methods !

setDictionary

"Частный. Инициализирует переменную экземпляра."

translation := Dictionary new.

16.5.2. Метод open

Теперь реализуем все указанные методы, начиная с метода **open**, который похож на метод **open** предыдущего приложения. Различие состоит в том, что здесь он значительно больше, поскольку в окно встраивается больше виджетов (вместо меню используются кнопки), а перед каждой списковой панелью располагается метка, которая является экземпляром класса **StaticText**. Кроме того, некоторые из панелей окна при определении получают имена.

Надо сказать, что в системе *Smalltalk Express* и во многих других системах «рисунок» интерфейса пользователя и тесно связанный с ним метод **open** не создаются вручную. Для их построения используется специальный инструмент. В системе *Smalltalk Express* этот инструмент называется **WindowBuilder Pro**. О том, как им пользоваться, мы поговорим в следующей главе. А пока, не очень заботясь о «красоте» создаваемого окна, сосредоточим все свое внимание на реализации функциональности приложения.

open

"Создает окно переводчика, определяет размеры панелей окна и их поведение. Открывает окно."

```
self labelWithoutPrefix: 'Language Translator';
    noSmalltalkMenuBar. "без стандартной строки меню"
self addSubpane: (StaticText new owner: self;
    framingRatio:((1/20)@(1/20) corner: (9/20)@(3/20));
    contents: 'English:';
    leftJustified); "выровнять влево"
addSubpane: (StaticText new owner: self;
    framingRatio:((11/20)@(1/20) corner: (19/20)@(3/20));
    contents: 'Russian';
    leftJustified);
```

```
addSubpane: (ListPane new owner: self;
  paneName: 'englishPane'; "внутреннее имя панели"
  framingRatio:((1/20)@(4/20) corner: (9/20)@(15/20));
  when: #getContents perform: #updateEnglishList;;
  when: #select perform: #selectEnglishItem;);
addSubpane: (ListPane new owner: self;
  framingRatio:((11/20)@(4/20) corner: (19/20)@(15/20));
  paneName: 'russianPane';
  when: #getContents perform: #updateRussianList;;
  when: #select perform: #selectRussianItem;);
addSubpane: (Button new owner: self;
  framingRatio:((1/20)@(16/20) corner: (4/20)@(19/20));
  paneName: 'addWordButton';
  when: #clicked perform: #clickedAddWord;;
  contents: 'Add Word');
addSubpane: (Button new owner: self;
  framingRatio:((6/20)@(16/20) corner: (9/20)@(19/20));
  paneName: 'deleteWordButton';
  when: #clicked perform: #clickedDeleteWord;;
  contents: 'Delete Word');
addSubpane: (Button new owner: self;
  framingRatio:((11/20)@(16/20) corner: (19/20)@(19/20));
  paneName: 'editTranslationButton';
  when: #clicked perform: #clickedEditTranslation;;
  contents: 'Edit Translation').
self openWindow.
```

16.5.3. Методы обработки событий

В этом приложении обработчики событий будут активно использовать внутренние имена панелей, определенные в методе `open` строкой-аргументом ключевого слова `paneName:`, а также модифицирующее сообщение `update`. Напомним, что посылка сообщения `update` (оно реализовано в классе `SubPane`) любой панели вызовет в ней событие `#getContents`, а это повлечет за собой выполнение владельцем панели обработчика данного события. Но, описывая функциональность окна, мы говорили об операциях, которые затрагивают сразу обе списковые панели. Поэтому, чтобы не повторять несколько раз посылку одной и той же последовательности сообщений, реализуем в самом классе `LanguageTranslator` метод экземпляра `update`, который будет производить модификацию сразу двух списковых панелей.

update

"Модифицирует приложение, для чего модифицирует панели русских и английских слов, посылая им сообщение `update`. Посылка панелям модифицирующего сообщения порождает в них событие `getContents`, которое вызывает обработчик, определенный для него в панели."

```
(self paneNamed: 'englishPane') update.  
(self paneNamed: 'russianPane') update.
```

Реализуем сами обработчики событий, упомянутые в методе `open`, снабжая каждый из них развернутым комментарием.

*Обработчики событий панели 'englishPane'***updateEnglishList: aPane**

"Обработчик события `#getContents`. Отображает в панели текущий список английских слов, выбирает слово из списка, соответствующее значению переменной `englishSelection`."

```
aPane contents: translation keys asSortedCollection;  
selection: englishSelection.
```

selectEnglishItem: aPane

"Обработчик события `#select`. Выбранное английское слово запоминается в переменной `englishSelection`. Русской панели посылается модифицирующее сообщение, чтобы определить перевод и выбирать его."

```
englishSelection := aPane selectedItem.  
(self paneNamed: 'russianPane') update.
```

*Обработчики событий панели 'russianPane'***updateRussianList: aPane**

"Обработчик события `#getContents`. Отображает в панели список русских слов, выбирает слово, соответствующее значению переменной `englishSelection`."

```
aPane contents: translation values asSortedCollection;  
selection: (translation at: englishSelection ifAbsent: [nil]).
```

selectRussianItem: aPane

"Обработчик события `#select`. Когда выбирается русское слово, соответствующее английское слово запоминается в переменной `englishSelection` и панель английских слов модифицируется."

```
englishSelection := translation  
keyAtValue: aPane selectedItem ifAbsent: [nil].  
(self paneNamed: 'englishPane') update.
```

Обработчик события #clicked кнопки AddWord**clickedAddWord: aPane**

"Вызывает диалоговое окно для ввода нового английского слова.

Если новое слово является допустимым, добавляет его в словарь translation и выбирает его, модифицирует обе панели."

| response |

response := Prompter prompt: 'New Word:' default: "".

response isNil | (response = "") "nil — отказ от ввода"

ifTrue: [^ self].

translation at: response put: (translation at: response

ifAbsent: [response,'translation']).

englishSelection := response.

self update.

Обработчик события #clicked кнопки DeleteWord**clickedDeleteWord: aPane**

"После подтверждения удаляет выбранный элемент и его

перевод из словаря и модифицирует списковые панели."

englishSelection isNil

ifTrue: [MessageBox message:

'You must select the English word to be deleted.']

ifFalse: [(MessageBox confirm:

'Are you sure you want to delete', englishSelection,?')

ifTrue: [translation removeKey: englishSelection.

englishSelection := nil. self update]].

Обработчик события #clicked кнопки EditTranslator**clickedEditTranslation: aPane**

"Если выбор сделан, открывает диалоговое окно

для редактирования существующего перевода."

| newTranslation |

englishSelection isNil ifTrue: [^ MessageBox message:

'You must select the translation to be edited.'].

(newTranslation := Prompter prompt: 'Edit the translation:'

default:(translation at: englishSelection)) isNil

ifTrue: [^ self].

translation at: englishSelection put: newTranslation.

(self paneNamed:'russianPane') update.

Чтобы иметь возможность открыть приложение на словаре, который был создан ранее, добавим следующий метод экземпляра:

openOn: aDictionary

"Открывает окно на словаре aDictionary."

```
translation := aDictionary.
```

```
self open.
```

Заключительный штрих: чтобы иметь возможность достаточно просто проверить работу созданного приложения, напишем *метод класса*, который позволит открыть окно приложения и поэкспериментировать с ним.

example1

"Метод тестирования окна LanguageTranslator."

```
self new openOn: (Dictionary new
                 at: 'red' put: 'красный';
                 at: 'green' put: 'зеленый';
                 at: 'blue' put: 'синий';
                 yourself).
```

Теперь для запуска приложения достаточно выполнить выражение `LanguageTranslator example1`.

16.6. О цикле разработки приложений

Если проанализировать все, что было сделано при построении классов `PhoneBook` и `LanguageTranslator`, можно выделить несколько характерных этапов их разработки.

- 1) Формулировка задачи.
- 2) Создание эскиза приложения и его окон.
- 3) Описание используемых классов.
- 4) Описание состояний используемых объектов.
- 5) Описание интерфейса используемых объектов.
- 6) Реализация необходимых методов.

Все перечисленные этапы тесно связаны друг с другом и чаще всего реализуются параллельно или последовательно (особенно этапы 3–5), повторяясь несколько раз. Разрабатывая новый проект, сначала мы получаем только «первый срез» в решении задачи. Полученный опыт разработки и эксплуатации приложения приведет к более глубокому пониманию проблемы, позволит увидеть новые подходы к ее решению. И, проходя уже более быстро те же шесть этапов, можно получить лучшее решение.

Смолтоковская система поддерживает (и даже провоцирует) эволюционный подход к разработке программного продукта, поскольку предоставляет разработчику множество возможностей. На практике разработка классов и

приложений может оказаться очень увлекательным занятием. Всегда можно придумать и добавить несколько строк кода, которые сделают интерфейс приложения более изящным или повысят эффективность метода. Поэтому важно помнить о первоначально поставленных задачах. Если созданный прототип удовлетворяет заданным критериям, стоит остановиться, приобрести опыт работы с программным продуктом и только затем переходить к его усовершенствованию.

Возвращаясь к классу `PhoneBook`, заметим, что в нашем проекте есть небольшой дефект. Включая в словарь новое имя, можно ошибиться. Если ошибку заметили во время набора, до включения в словарь, ее легко исправить. Если же ошибочный ключ уже внесен в словарь, то сейчас единственный способ исправить его — удалить старую запись и ввести новую, но при этом потеряется правильный номер телефона.

Те же рассуждения справедливы и в отношении панели английских слов приложения `LanguageTranslator`. Но здесь есть и более значительный дефект. Дело в том, что значения в словаре, в отличие от ключей, не являются уникальными (по крайней мере, словари при их заполнении не проверяют уникальность значений). Для нашего приложения это может привести к его некорректной работе, если у двух разных английских слов окажутся одинаковые переводы.

Устранение отмеченных недостатков мы оставляем в качестве упражнения для читателей. Как варианты решений можем предложить следующее. В классе `PhoneBook` изменить метод `listMenu`, добавив в меню пункт `Correct`, и написать связанный с ним метод `correct`, позволяющий исправить уже внесенное в список имя. Для решения проблемы с переводами можно, например, создать новый подкласс в иерархии класса `Dictionary`, который будет обеспечивать уникальность не только ключей, но и значений, а в переменной `translation` хранить экземпляр созданного класса. Но это не лучшее решение. Лучше поискать для значения переменной `translation` другой набор, лучше отвечающий новому требованию к приложению. Внести изменения будет совсем не сложно.

Теперь мы знаем основные принципы построения интерфейса пользователя в системе *Smalltalk Express*. Но «вручную» приятно строить только достаточно простые приложения, в то время как у реальных приложений интерфейс может быть очень сложным. Поэтому нам нужны инструменты, которые помогут автоматизировать и значительно ускорить процесс построения интерфейса пользователя.

ГЛАВА 17

WINDOWBUILDER PRO

Смолтоковские системы давно рассматриваются как одни из лучших для построения приложений со сложным интерфейсом пользователя, и каждая реализация содержит необходимые для этого инструменты. Все они служат одной цели: освободить программиста от утомительной работы по проектированию графических интерфейсов. Чтобы объяснить и проиллюстрировать технологию визуального построения интерфейса, подробно рассмотрим инструмент *WindowBuilder Pro*¹ (WBP). Сначала в процессе разработки простого приложения, мы познакомимся с инструментом WBP, его возможностями и особенностями, а затем с его помощью построим более сложное приложение.

17.1. Построение простого счетчика

Построим приложение, представляющее окно с именем CounterDemo, с двумя большими кнопками “Increment” и “Decrement” в левой половине окна и с текстовым полем в правой половине окна, которое после открытия окна содержит значение “0”. Работа счетчика очень проста: при нажатии кнопки “Increment” значение счетчика, отображаемое в текстовом поле, увеличивается на 1, а при нажатии кнопки “Decrement” — уменьшается на 1. Операции, связанные с кнопками, продублируем в виде пунктов меню Actions, а само меню разместим в панели меню под строкой заголовка окна. После завершения построения окно счетчика должно иметь вид, представленный на рис. 17.1.

Создание приложения с помощью WBP состоит из трех процессов, которые мы подробно рассмотрим, объясняя каждый шаг:

- 1) создание желаемого интерфейса пользователя,
- 2) создание необходимых меню окна и панелей,
- 3) написание методов, обеспечивающих функциональность приложения.

¹ Описание инструмента WBP, приведенное в этой главе, опирается на [17]. Перевод этого учебника на русский язык, как часть дипломной работы, был выполнен студенткой механико-математического факультета Ростовского государственного университета А.С. Потапенко.

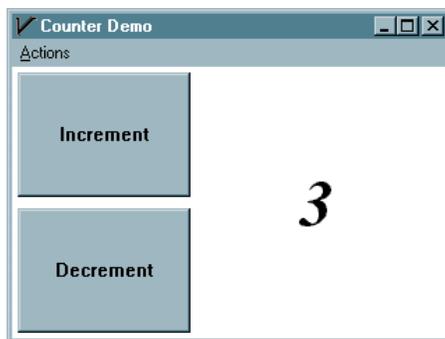


Рис. 17.1. Внешний вид окна Counter Demo

17.1.1. Создание интерфейса

Итак, надо создать приложение, все объекты которого уже созданы и хранятся так, что их легко найти. В этом случае все наше внимание будет сконцентрировано на главном: на проблемах проектирования интерфейса пользователя, что при наличии «умного» инструмента не столь сложно, как кажется на первый взгляд.

Процесс создания интерфейса заключается в «наполнении» окна приложения посредством выбора из палитры нужных виджетов, их размещения в окне и последующей настройки в соответствии с предъявляемыми к интерфейсу требованиями. Но прежде всего следует запустить сам инструмент WBP (рис. 17.2). Для этого можно воспользоваться специальным меню WindowBuilderPro, расположенным в меню окна Transcript. Для построения нового окна из предлагаемых этим меню пунктов надо выбрать или New Window, или New Dialog.

Если необходимо продолжить работу с уже имеющимся окном приложения, надо из меню WindowBuilderPro выбрать пункт Edit Window. . . Появится диалоговое окно, из которого можно выбрать нужный объект, после чего окно редактора WBP откроется для дальнейшего редактирования этого объекта. По умолчанию диалоговое окно содержит список только тех подклассов ViewManager, которые «создал» инструмент WBP. Если требуется отредактировать экземпляр какого-либо другого подкласса, надо нажать на кнопку Non WB View. . . , после чего появится новое диалоговое окно, содержащее все подклассы ViewManager, интерфейс экземпляров которых содержит метод createView или open. Для классов, метод open которых не создавался с помощью WBP, будет отображаться некоторая дополнительная информация, касающаяся этих окон.

Выбор и размещение компонентов

Итак, выберем пункт **New Window** из меню **WindowBuilderPro** окна **Transcript**. На экране появится окно инструмента **WBP** (рис. 17.2), в рабочей области которого будет располагаться окно с заголовком **New Window** — это тот фундамент, с которого начинается построение интерфейса пользователя. С момента своего создания это окно уже функционально: оно имеет заголовок, рамку, можно изменять его размеры; в него уже включены кнопки для управления окнами, обычные в данной операционной системе (минимизации/максимизации размеров, закрытия, системного меню).

В нижней части основного окна **WBP** расположена панель, которая используется для изменения *свойств* выбранного в рабочей области виджета. Слева на ней расположены три поля ввода с метками **Text:**, **Style:**, **Name:**. Справа имеются два поля ввода с метками **When:** и **Perform:**. Они предназначены для изменения *поведения*. Первоначально в поле **Text** выделен текст, представляющий имя окна. По умолчанию новому окну дается имя **'New Window'**. Введите в поле ввода **Text** строку **'Counter Demo'** (без кавычек). Обратите внимание, что при этом автоматически изменяется имя в заголовке окна.

Чтобы изменить размеры окна, выберите окно, щелкнув левой кнопкой мыши где-либо на свободном пространстве внутри рамки окна. Захватите курсором маркер (черный квадратик, находящийся в нижнем правом углу окна) и, удерживая нажатой левую кнопку мыши, перемещайте курсор, определяя желаемый размер. Следите за изменениями чисел в правом из двух прямоугольников, расположенных между рабочей областью инструмента и панелью изменения поведения объекта. Остановитесь, когда значения чисел будут примерно равны 300×240 , и отпустите кнопку мыши.

Размеры окна можно изменить, задавая их явно. Для этого выберите окно, найдите в нижней части окна **WBP** панель инструментов и в ней самую правую панель, в которой изображен маленький прямоугольник со стрелками вдоль сторон и числовыми размерами справа от прямоугольника. Нажмите на эту панель (или выберите команду меню **Size > Set Window Size...**). Появится диалоговое окно, в котором надо ввести точку, определяющую новые размеры окна (ширину и высоту). Нажатие на кнопку **OK** приведет к закрытию диалогового окна и изменению размера редактируемого окна. Для того чтобы закрыть диалоговое окно, не изменяя размеры редактируемого окна, необходимо нажать на кнопку **Cancel**.

В этом же меню **Size** есть пункт **Set Window Position...**, который позволяет установить начальную позицию окна при его тестировании в **WBP**. Выбор этого пункта вызовет послушный движению мыши прозрачный прямоугольник, размер которого совпадает с редактируемым окном. Когда пря-

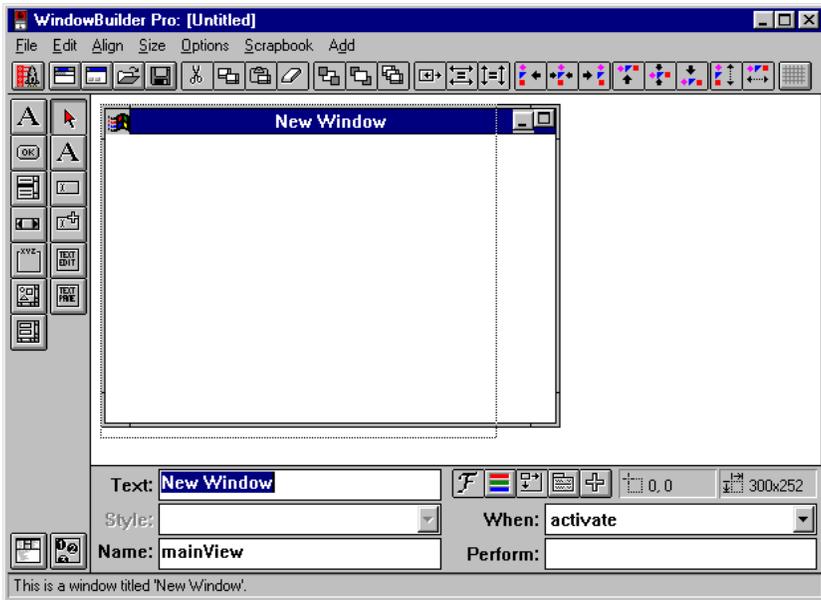


Рис. 17.2. Инструмент WBP

моугольник будет находиться в нужной позиции, нажмите левую кнопку мыши. После этого именно здесь при тестировании будет располагаться создаваемое окно. Можно установить разные позиции для разных окон, что может оказаться удобным при тестировании нескольких приложений сразу. То же самое можно сделать, щелкая в нижней панели инструментов окна WBP левой кнопкой мыши на панели, в которой изображен маленький прямоугольник с перекрестием в верхнем углу и числовыми координатами точки справа от прямоугольника. Если положение окна не устанавливается разработчиком, то при тестировании приложения его окно располагается в центре экрана.

Итак, у нас есть окно, но внутри окна еще ничего нет. Для построения интерфейса — заполнения окна необходимыми виджетами (элементами интерфейса) — предназначена панель с двумя вертикальными палитрами, расположенная в левой части окна WBP. Палитры позволяют выбирать виджеты, которые надо разместить в окне. Левая палитра отображает *категории* виджетов. В соответствии с выбранной категорией меняется правая палитра, отображающая конкретные *виды* виджетов, которые можно добавлять в окно. Палитры виджетов продублированы содержимым меню Add.

Нажмите на вторую сверху кнопку левой палитры (она имеет иконку, похожую на миниатюрную кнопку ОК). Из этой информации и из текста, появившегося в информационной строке, расположенной вдоль нижней границы окна WBP, можно сделать вывод о том, что выбор данной категории открывает палитру кнопок. Их иконки наглядным образом описывают их назначение. Когда выбирается одна из них, в информационной строке отображается поясняющий текст, а курсор в рабочей области изменяет свою форму, показывая, что он *загружен* выбранным виджетом.

Обратите внимание на стрелку в верхней части правой палитры. Когда стрелка выбрана, курсор не загружен виджетом и можно работать с любым элементом интерфейса, уже расположенным в окне.

Выберите из правой палитры кнопку, расположенную сразу под стрелкой (загрузите курсор кнопкой). Когда курсор мыши переместится в область окна, он изменит свой вид на перекрестие. Для того чтобы создать кнопку в окне, можно поступить по-разному. Можно просто установить перекрестие в ту точку окна, в которой предполагается разместить левый верхний угол виджета, и щелкнуть левой кнопкой мыши: в окне появится кнопка, размер которой определяется по умолчанию. А можно не щелкать левой кнопкой мыши, а нажать ее и, не отпуская, переместить мышь вниз и вправо, сразу определяя нужный размер кнопки: размер установится в тот момент, когда кнопка будет отпущена. Первую кнопку поместите так, чтобы она занимала почти всю левую верхнюю четверть окна. Если сразу не удалось поместить кнопку в нужное место или задать ей желаемый размер, не тратьте на это время. Ниже мы рассмотрим, как скорректировать многие характеристики разных элементов интерфейса, в том числе их положение в окне и размеры. А пока разместите в окне еще одну кнопку, которая должна занимать левую нижнюю четверть окна.

Чтобы поместить в окно текстовую панель, выберите в левой палитре пиктограмму с большой буквой “А”, переместитесь на правую палитру и выберите объект **StaticText** (с такой же пиктограммой). Такие виджеты не генерируют никаких событий, но могут принимать сообщения. Переместите курсор в редактируемое окно и расположите его так, чтобы он оказался в центре правой половины окна приложения.

Отметим еще несколько моментов. Ненужный виджет можно удалить из окна либо выбирая его и нажимая в панели инструментов на кнопку с изображением резинки, либо нажимая на клавиатуре **[Ctrl]+[X]**, либо выбирая из меню **Edit > Cut**.

Если при построении приложения нужно разместить несколько однотипных панелей, можно обычным образом выбрать из палитры панель, но, размещая его в окне, нажимать *правую* кнопку мыши. Только последнюю

панель необходимо разместить с помощью левой кнопки мыши. Но завершить размещение однотипных панелей можно и по-другому: выбрать в правой палитре кнопку со стрелкой.

Правильное размещение элементов

Вам часто придется размещать в окне виджеты, поэтому давайте посмотрим, как это делается. В процессе размещения виджетов внутри окна необходимо обращать внимание на следующие основные моменты:

- местоположение и размеры элементов;
- расположение элементов относительно друг друга;
- расположение элементов относительно рамки окна;
- выравнивание выбранных объектов по отношению друг к другу.

Чтобы переместить виджет в окне как единое целое без изменения его размеров, следует сначала выделить нужный виджет, то есть просто щелкнуть левой кнопкой мыши по нему, затем нажать левую кнопку мыши где-либо внутри виджета и, удерживая кнопку, перемещать мышь. Вместе с ней будет перемещаться и виджет. Он займет новое положение в тот момент, когда кнопка мыши будет отпущена. Обратите внимание на поле, помеченное маленьким прямоугольником с перекрестием, в левой части панели свойств виджета: при перемещении мыши с нажатой левой кнопкой числа в нем будут изменяться. Они показывают координаты той точки внутри окна, в которой располагается левый верхний угол виджета.

Если надо переместить выделенный виджет всего на несколько пикселей, можно использовать сочетание клавиши **[Ctrl]** и клавиш со стрелками. То же самое можно сделать, используя пункт меню **Align > Move By Pixel**.

Размеры любого виджета можно изменить точно так же, как мы изменяли размеры основного окна, то есть путем перемещения одного из маркеров, возникающих после выделения виджета. При этом в поле панели свойств виджета, помеченном прямоугольником с размерными линиями, во время изменения размеров виджета числа будут изменяться, показывая его ширину и высоту. Если при настройке размеров объекта требуется произвести изменения лишь в одном направлении (например, сделать кнопку только более широкой, не меняя при этом ее высоту), то нажмите клавишу **[↑ Shift]** прежде чем перемещать маркер выбранного объекта. В результате курсор будет отражать направление, в котором вы начинаете перемещать маркер, и изменение размеров будет происходить только в выбранном вами направлении.

Если необходимо произвести изменение размеров всего на несколько пикселей, можно воспользоваться сочетанием клавиш **[Ctrl]+[↑ Shift]** и

клавиш со стрелками. Тот же эффект достигается при использовании меню **Align** ▸ **Size By Pixel**.

Применяя любой из этих способов, добейтесь того, чтобы верхняя кнопка своим верхним левым углом располагалась в точке **8@4** и имела размеры **108@100** (точность здесь не важна, ваши значения могут не совпадать с указанными на несколько единиц).

Описанные операции будет проще выполнять, если на область создаваемого окна наложить сетку, для чего достаточно нажать на кнопку с изображением такой сетки, расположенную справа в панели инструментов. Размер ячеек сетки (шаг сетки) можно менять, пользуясь пунктом меню **Options** ▸ **Grid Size...**

Можно изменить размеры виджета и его положение, открывая диалоговые окна и явно задавая нужные размеры. Прделаем подобное для нижней кнопки. Итак, выберите нижнюю кнопку и в панели свойств виджета нажмите на пиктограмму в поле, отображающем координаты левого верхнего угла виджета (левое поле с координатами). Появится диалоговое окно для установки положения объекта, в котором показано текущее положение кнопки. Введите новое значение: **8@108**. Нажмите кнопку **ОК**, диалоговое окно исчезнет, а кнопка немедленно займет новое положение.

Нажмите на такую же иконку, расположенную правее и отображающую размеры виджета. Появится похожее диалоговое окно. Как и в первом случае введите значение **108@100** и нажмите кнопку **ОК**. Обе кнопки имеют одинаковые размеры и расположены симметрично вдоль рамки окна.

Теперь выберите объект **StaticText** и установите его размеры и положение в правой половине окна любым способом. Например, пусть точка, определяющая положение, будет равна **156@60**, а размеры — **100@100**.

Рассчитывая положение виджета и его размеры, не забывайте, что все размеры задаются в пикселах. Размеры самого окна определяются по внешней стороне рамки, строка заголовка окна в высоту составляет 25 пикселей, а ширина самой рамки составляет несколько пикселей и зависит от настроек **Windows**. Точка с координатами **0@0** находится в верхнем левом углу внутренней (клиентской) области окна.

Мы завершили формирование внешнего вида окна счетчика. Сделанную работу пора сохранить. Для этого следует выбрать пункт меню **Save** ▸ **File**, либо нажать на кнопку с изображением дискеты, расположенную в панели инструментов, либо ввести клавиатурную комбинацию **[Ctrl]+[S]**. **WBP** откроет диалоговое окно, где надо указать имя для вновь созданного подкласса класса **ViewManager**, экземпляром которого будет наш счетчик. В этом же диалоговом окне определяется и суперкласс для нового класса. Если новое окно создается с самого начала, то его класс, вероятнее всего,

будет непосредственным подклассом класса `ViewManager` (так по умолчанию и устанавливается инструментом). Если же необходимо, чтобы новый интерфейсный класс стал подклассом уже существующего подкласса из `ViewManager`, нужно выбрать соответствующий класс из нижнего комбинированного списка диалогового окна. Остается задать новому подклассу подходящее имя (назовем его `CounterDemo`), набирая его в поле ввода верхнего комбинированного списка и нажимая кнопку `ОК`.

Изменение характеристик компонентов

Каждый тип компонентов интерфейса имеет собственный набор характеристик (свойств). Наиболее важные и часто изменяемые из них отображаются в панели, расположенной в нижней части окна `WBP`. Изменить любое свойство объекта довольно просто. Для этого надо выбрать нужный объект, а затем отредактировать те характеристики, которые появляются в этих двух панелях.

В самом начале построения счетчика мы уже изменили имя окна приложения. Теперь давайте изменим надписи на обеих кнопках окна `Counter Demo`, а потом аналогичным образом изменим свойства объекта `StaticText`.

Итак, открываем меню `Options` и убеждаемся, что пункт `Auto Size` отключен (если это не так, то отключаем его). Зачем мы это делаем? При проектировании интерфейсов автоматическое определение размеров во многих ситуациях бывает очень удобно. Например, для того, чтобы кнопка стала достаточно большой для размещения текста метки и некоторого дополнительного пространства вокруг него. Если возникнет необходимость изменить текст, то при выделенной опции `Auto Size` размеры кнопки изменятся автоматически. Однако, если необходимо сохранять для кнопки или другой панели размеры, заданные первоначально, этот пункт должен быть отключен, что мы и сделали. На объекты, размеры которых фиксированы и не зависят от данных, команда `Auto Size` не оказывает влияния.

В окне `Counter Demo` выбираем верхнюю кнопку, при этом в панели свойств поля меняют свое содержание, и в поле ввода с меткой `Text:` оказывается выделенным слово `Button`. Изменяя имя кнопки, вводим новое имя: `Increment`. В процессе ввода новое имя посимвольно заменяет старое как в поле `Text`, так и на выбранной кнопке. Выберем нижнюю кнопку окна и таким же образом изменим ее имя на `Decrement`.

Теперь щелкнем мышью на объекте `StaticText`, выбирая его. Обратите внимание, что `StaticText` имеет те же свойства `Text:`, `Style:`, `Name:`, `When:` и `Perform:`, что и кнопка. Зададим объекту `StaticText` значение по умолчанию, то есть значение, которое будет отображаться при запуске приложения. Но прежде установим шрифт большего размера. Для этого, предварительно

проверив, что опция **Auto Size** отключена, щелкнем на панели **Font**, на которой изображена буква *ƒ* и которая расположена слева в панели свойств виджета. Появится стандартное окно для выбора и установки шрифта. Выберите понравившийся вам шрифт, установите его размер равным 36 пунктам и закройте диалоговое окно.

При возвращении в **WBP** мы увидим, что метка **Static Text** стала слишком большой, чтобы отобразиться в выделенной для нее области. Теперь в поле ввода **Text** вместо выделенной фразы **Static Text** напечатаем 0 (ноль). Число, отображаемое в текстовой области окна, будет в данном случае лучше смотреться в центре. Чтобы добиться этого, надо обратиться к полю с меткой **Style:**, которое представляет собой комбинированный список. Нажмем на кнопку со стрелкой в правой части этого поля и из появившегося списка выберем элемент **centered** (центрировать). Обратите внимание, что ноль по горизонтали переместился в центр текстового поля **StaticText**. Если созданное окно вас чем-то не удовлетворяет, измените его и добейтесь желаемого (например, изменяя размеры и положение текстового поля, можно добиться того, чтобы отображаемое число располагалось точно в центре не занятой кнопками части окна).

На данном этапе полезно еще раз сохранить сделанные настройки.

Разработка счетчика еще не завершена, он не решает поставленной задачи, но мы пока не написали на языке Смолток ни одной строчки. Однако приложение стоит протестировать. Это можно сделать из окна **WBP** одним из следующих способов:

- выбрать пункт **Test Window** из меню **Edit**;
- использовать клавишный эквивалент **[Ctrl]+[T]**;
- нажать левую кнопку (с ракетой) в панели инструментов.

В любом случае появится созданное нами окно счетчика. Его можно перемещать, изменять размеры, нажимать на его кнопки (пока безрезультатно). Закроем его и вернемся в редактор **WBP**. Что же создал для нас инструмент, пока мы производили в нем все эти манипуляции? Чтобы ответить на этот вопрос, отыщем в нижнем левом углу редактора **WBP** две маленькие кнопки. Нам нужна левая кнопка. Она открывает окно просмотра иерархии классов на классе того приложения, которое в данный момент редактируется в **WBP**. Нажмем на нее. Появится окно просмотра, где будет выделен класс **CounterDemo**. В нем уже есть метод класса **wbCreated**, который всегда возвращает **true**, указывая, что данный класс создан с помощью **WBP**, и метод экземпляра **createViews**, который имеет хорошо нам известную из предыдущего структуру метода **open**. Именно эти два метода и создал **WBP**. Основная структура метода **createVeivs** выглядит так:

createViews

"WARNING! This method was automatically generated by WBP. Code you add here which does not conform to the WBP API will probably be lost the next time you save your layout definition."

```
| v |
self addView: (
v := self topPaneClass new
    "Присваивает главному окну имя v,
    которое можно использовать позже."
    ...;
    addSubPane: (. . .);
    addSubPane: (. . .);
    ...).
```

Обычно сообщение `addView:` в методе `createViews` посылается всего один раз. Затем для создания каждого виджета, появляющегося в окне, посылается сообщение `addSubPane:`. Обратите внимание на комментарий в начале метода. Он предупреждает, что неразумно самостоятельно редактировать этот метод, поскольку WBP его создает автоматически. Если вы добавите сюда код, который не согласуется со способом действия инструмента, в последующем он может быть потерян. Поэтому не изменяйте метод `createViews` вручную.

17.1.2. Создание меню

Мы подошли к заключительному моменту в построении интерфейса: осталось добавить в создаваемое окно меню. Для этого воспользуемся еще одним инструментом, предоставляемым системой: редактором `Menubar`. Прежде чем его запустить, убедитесь в том, что выбрано само окно приложения. После этого или вызовите всплывающее меню рабочей области окна WBP (используя правую кнопку мыши) и выберите из него пункт `Menubar`, или нажмите на кнопку с изображением раскрытого меню, расположенную на панели свойств виджетов. Появится диалоговое окно редактора меню (см. рис. 17.3).

В самом верхнем слева поле ввода напечатайте `Actions` и нажмите клавишу **[Enter]**. Слово `Actions` становится первым в расположенной ниже списковой панели. Напечатайте по порядку нужные элементы меню — `&Increment` и `&Decrement`, нажимая после каждого ввода клавишу **[Enter]** или кнопку `Insert`. Символ `&` (амперсанд) сообщает Windows о подчеркивании следующего за ним символа. После этого пользователь может активировать данный пункт меню с помощью указанной клавиши.

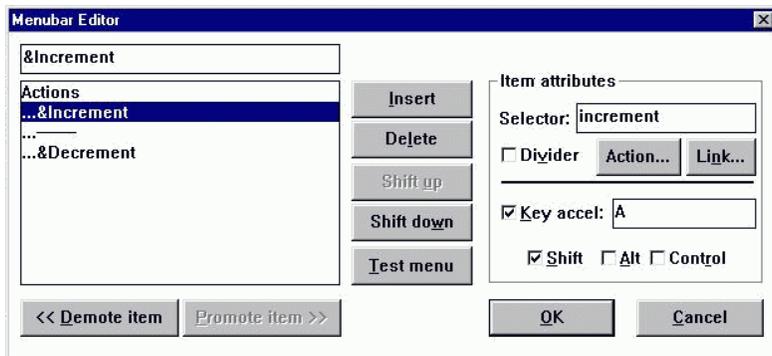


Рис. 17.3. Внешний вид окна редактора меню

Далее необходимо упорядочить пункты меню в иерархию. Начнем с верхнего пункта меню **&Increment**, выбирая его и нажимая ставшую активной кнопку с меткой *PromoteItem*. Теперь элемент списка **&Increment** переместился вправо и ему предшествует многоточие. Таким образом, определено место этого элемента в структуре создаваемого меню. Выберем следующий элемент списка **&Decrement**. Нажмем на кнопку *PromoteItem*, располагая выделенный элемент под элементом **&Increment**. В результате **Actions** — имя меню из строки меню окна, пунктами которого будут **&Increment** и **&Decrement**.

Обратите внимание, что выделенный в списковой панели элемент можно удалить, нажимая кнопку *Delete*, или сдвинуть влево, нажимая кнопку *DemoteItem*. Между пунктами меню можно ввести разделитель — горизонтальную линию. Для этого в левом верхнем поле ввода надо ввести символ «-» (минус). Введенный и выделенный элемент меню можно поднимать вверх или опускать вниз, нажимая на кнопки *Shift Up* и *Shift Down* соответственно. Созданное меню можно сразу же протестировать, нажимая на кнопку *Test menu*.

Каждому пункту меню можно сопоставить клавиатурный акселератор (горячую клавишу). Для этого используется нижняя часть правой панели редактора *Menubar* с именем *Item attributes* (Атрибуты элемента). В *Small-talk Express* определяемое сочетание клавиш должно обязательно начинаться с одной из клавиш [**Shift**], [**Alt**], [**Ctrl**]. Поэтому для определения клавиатурного эквивалента выбору пункта **&Increment**, необходимо выполнить следующие действия:

- выбрать пункт меню **&Increment**,
- выбрать флажок **Shift**,

- выбрать флажок перед меткой **Key accel:**,
- в поле ввода после **Key accel:** ввести символ **A**.

Аналогично для выбора пункта **&Decrement** определим [**⇧ Shift**]+**[S]** в качестве горячей клавиши. Мы специально выбрали клавиши, не совпадающие с подчеркнутыми символами в меню, чтобы подчеркнуть их независимость. В реальном, а не учебном приложении так поступать не стоит.

Если сейчас нажать в редакторе **Menubar** кнопку **OK** и возвратиться в среду **WBP**, то обнаружим, что в редактируемом окне появилась строка меню с именем **Actions**. При этом проведенная нами работа по размещению интерфейса может быть немного подпорчена, но, чтобы все восстановить, достаточно увеличить высоту окна на высоту вставленной строки меню. Перед тем, как переходить к следующему этапу, сохраним сделанное.

В среде *Smalltalk Express* всплывающее меню могут иметь и некоторые типы панелей. Если в окне выбрать такую панель, то кнопка **Menu** станет доступной и, нажав на нее, можно создать или отредактировать всплывающее меню выделенной панели. Отличие при построении строки меню и всплывающего меню состоит в том, что в первом случае пункты меню верхнего уровня являются именами для меню и располагаются на панели меню окна. Для таких пунктов обязательно требуется наличие подменю. Еще одно отличие при построении этих меню состоит в разной реакции редактора меню на нажатие кнопки **Test menu**.

Осталось определить действия приложения, производимые при выборе пунктов меню. Но то же самое надо сделать и для других активных компонентов созданного интерфейса. Пришла пора заставить счетчик выполнять действия, ради которых он создается. Другими словами, пришла пора воспользоваться в редакторе **Menubar** верхней частью панели **Item attributes**, а в самом редакторе **WBP** — полями **When:** и **Perform:**.

17.1.3. Определение функциональности окна приложения

Итак, мы должны соединить активные компоненты интерфейса (кнопки, пункты меню) с методами таким образом, чтобы приложение могло выполнять необходимые операции. Начнем с более простого — с меню.

Снова откроем редактор **Menubar**. Верхняя часть панели **Item attributes** редактора **Menubar** позволяет устанавливать связи между пунктами меню и методами. Для этого надо выбрать элемент меню, а затем напечатать имя метода в поле ввода с меткой **Selector**. Свяжем пункт меню **&Increment** с методом **increment**, а пункт меню **&Decrement** с методом **decrement**. Когда все определения сделаны, нажмем на кнопку **OK** и возвратимся в среду визуального редактора **WBP**.

Сохраним сделанную работу и протестируем окно. Воспользуемся меню и убедимся, что пункты меню на месте и расположены надлежащим образом. Можете выбрать один из них, хотя на данный момент это еще не принесет никаких результатов. Надо написать сами методы `increment` и `decrement`. Но что они будут изменять? Нам понадобится переменная экземпляра, назовем ее `value`. Именно значение этой переменной будет отображаться в окне. Откроем окно просмотра иерархии классов на классе `CounterDemo` и в его определение добавим переменную экземпляра `value`. Сохраним это определение и перейдем в панель методов экземпляра. Прежде чем использовать переменную, ей следует дать начальное значение; в Сmolтоке для этого используется метод с именем `initialize`:

initialize

```
"Определяет начальное значение счетчика равным нулю."  
super initialize.  
value := 0.
```

Допишем пустые методы `increment` и `decrement`, которые для нас создал `WBP`. Мы бы хотели, выбирая пункт `Increment`, увеличивать значение счетчика на 1, а выбирая пункт `Decrement`, уменьшать его значение на 1. Изменив значение переменной `value`, важно не забыть сообщить об этом панели `StaticText`.

increment

```
"Увеличивает счетчик на 1, сообщает об этом StaticText."  
value := value + 1.  
self changed: #value:.
```

decrement

```
"Уменьшает счетчик на 1, сообщает об этом StaticText."  
value := value - 1.  
self changed: #value:.
```

Обратите внимание, что в каждом из этих методов используется обычный способ изменения панелей с помощью метода `changed:.` Метод, который передается в качестве аргумента, называется `value:.` Создадим его.

value: aStaticText

```
"Отображает значение счетчика в панели StaticText."  
aStaticText contents: value printString.
```

С операциями, связанными с меню, мы разобрались; перейдем теперь к кнопкам. Чтобы заставить кнопки работать должным образом, надо связать конкретное событие, происходящее с виджетом, с методом, который должен выполняться в ответ на возникновение такого события. Для этого используется панель свойств виджета с полями `When:` и `Perform:.` Когда

нужный активный виджет окна приложения выбран, из комбинированного списка возможных событий **When**: надо выбрать событие, а в поле ввода **Perform**: ввести имя сообщения, которое будет послано окну приложения, если указанное событие произойдет.

Как и в случае с пунктами меню, мы бы хотели, чтобы пользователь, нажимая на кнопку с именем **Increment**, увеличивал на 1 значение счетчика, отображаемое в поле **Static Text**, а нажимая на кнопку с именем **Decrement**, уменьшал его значение на 1. Метод, обрабатывающий события любого виджета, должен быть *ключевым методом*, аргументом которого является сам виджет. Поэтому для подключения кнопки **Increment** выполним перечисленные ниже действия.

- 1) Выберем кнопку **Increment** в создаваемом окне приложения.
- 2) Убедимся, что в поле **When**: отображается имя события **clicked**. При желании, нажимая на кнопку с маленькой стрелкой справа от поля **When**:, можно увидеть список всех событий, на которые может отвечать кнопка, созданная в **WBP**.
- 3) В поле **Perform**: введем слово **increment**.

Повторим эти действия для кнопки **Decrement**, вводя **decrement** в качестве имени метода. Противоречия с тем, что мы сказали о невозможности использовать ранее созданные методы **increment** и **decrement**, здесь нет. Дело в том, что **WBP** автоматически создаст метод с именем, составленным из строки, введенной в поле **Perform**: и добавленного в ее конец двоеточия (:).

При создании окон с большим числом виджетов полезно увидеть все используемые события и связанные с ними сообщения окна. Для этого из меню **Edit** окна **WBP** достаточно выбрать команду **Event Summary**.

Если мы снова откроем окно просмотра иерархии классов, то увидим, что класс **CounterDemo** уже содержит новые методы экземпляра: **increment: aPane** и **decrement: aPane**. Остается изменить их так, чтобы они имели следующий вид:

increment: aPane

```
"Увеличивает счетчик на 1, сообщает об этом StaticText."  
self increment.
```

decrement: aPane

```
"Уменьшает счетчик на 1, сообщает об этом StaticText."  
self decrement.
```

Закроем браузер иерархии классов, возвратимся в **WBP** и протестируем приложение. Оно все равно не работает. Недоработка почти очевидна: мы не соединили панель **StaticText** с событием, и потому она не обновляется. Вернемся в редактор **WBP** и, чтобы поправить дело, выберем объект

`StaticText`. Убедимся в том, что в поле свойства `When`: отображается событие `getContents`. В поле `Perform`: введем имя уже созданного метода `value`. Сохраним сделанное и снова протестируем окно.

Если все набрано правильно, окно поведет себя как планировалось: по команде будет увеличивать или уменьшать значение счетчика на 1. Когда команда выполняется, новое значение счетчика отображается в панели `StaticText`. Чтобы вне инструмента `WBP` открыть созданное окно счетчика, достаточно вычислить выражение `CounterDemo.open`.

17.2. Инструменты WBP

Рассмотрим, как можно решать с помощью `WBP` некоторые задачи, часто встречающиеся при построении интерфейса. В данном разделе мы рассмотрим следующие операции:

- выравнивание и установка размеров виджетов;
- поведение компонентов окна, меняющего свои размеры;
- изменение внешнего вида виджетов;
- управление порядком табуляции виджетов в окне;
- создание и использование сложных панелей;
- сохранение и поиск компонентов интерфейса в альбоме;
- быстрое макетирование, использующее `LinkButton`, `LinkMenu`, `ActionButton` и `ActionMenu`;
- быстрая модификация средств управления.

17.2.1. Выравнивание и установка размеров виджетов

С перемещением и установкой размеров каждого виджета в `WBP` мы уже знакомы. Если в окне есть несколько объектов, размеры и размещение которых зависят друг от друга, то размещая их по одному, придется долго заниматься «выравниванием по пикселям» каждого объекта в отдельности. К счастью, в `WBP` можно очень быстро выполнить подобные операции, работая сразу с группой виджетов:

- сначала обычным щелчком мыши выбрать базовый виджет, положение и/или размер которого являются «правильными» и который используется `WBP` как шаблон для всех других объектов, выбранных наряду с ним;
- затем, удерживая клавишу [**↑ Shift**], щелкнуть левой кнопкой мыши на нескольких «неправильных» объектах, то есть на тех, которым нужно придать правильное расположение и/или размер (`WBP` отмечает каждый такой объект, помещая его в рамку со светлыми маркерами);

- теперь над выбранными объектами следует выполнить необходимые операции по их выравниванию, либо выбирая команды из меню Align (Выровнять) или Size (Размер), либо используя панель инструментов или клавиатурные команды.

Есть и второй способ выбора множества элементов. Он состоит в том, чтобы в области окна нажать на левую кнопку мыши, а затем, не отпуская ее, переместить мышь таким образом, чтобы необходимые объекты попали в прямоугольник, появившийся в результате движения мыши. Теперь, если отпустить мышь, все компоненты, которые хотя бы частично попали в прямоугольник, будут выделены. Но при таком выборе элементов нет «правильного» объекта. Операции с этой группой будут проводиться по правилам, определяемым инструментом. Например, выравнивание по левой границе элементов будет происходить по самой левой из левых границ выделенных объектов. Чтобы из такого набора снять выделение одного из объектов, надо, удерживая нажатой клавишу [**↑ Shift**], снова его выбрать.

Когда приходится распределять виджеты горизонтально или вертикально, порядок их выбора не имеет значения, но виджеты равномерно располагаются в области, ограниченной крайними сторонами выбранных виджетов, а не по всему окну. Последний способ установки размеров — использование авторазмера. Когда нажимается самая левая из перечисленных кнопок, объекты автоматически изменяют свои размеры. В этом случае определение соответствующего размера зависит от двух факторов: от типа включаемого в операцию виджета и от размера его метки или содержания. Вместо кнопки можно использовать пункт Auto Size Selection из меню Size.

17.2.2. Согласованное изменение размеров

Возможно, для некоторых приложений потребуется использовать окна, размер которых может изменяться пользователем. Естественно, что при этом должны изменяться его компоненты. В системе *Smalltalk Express* для этих целей служит инструмент Framing, который позволяет определить поведение каждого виджета при изменении размеров содержащего его окна.

Связанное с этим инструментом диалоговое окно Set Framing Parameters (Установка Параметров Прямоугольника) открывается для выделенного виджета при нажатии на среднюю кнопку в панели свойств виджета. Инструмент не доступен для модальных диалоговых окон, поскольку они не могут изменять свои размеры. Окно содержит справа две диаграммы, которые дают представление об использованных для виджета настройках его границ. Верхняя диаграмма представляет положение панели до изменения размеров окна, а нижняя — после изменения его размеров.

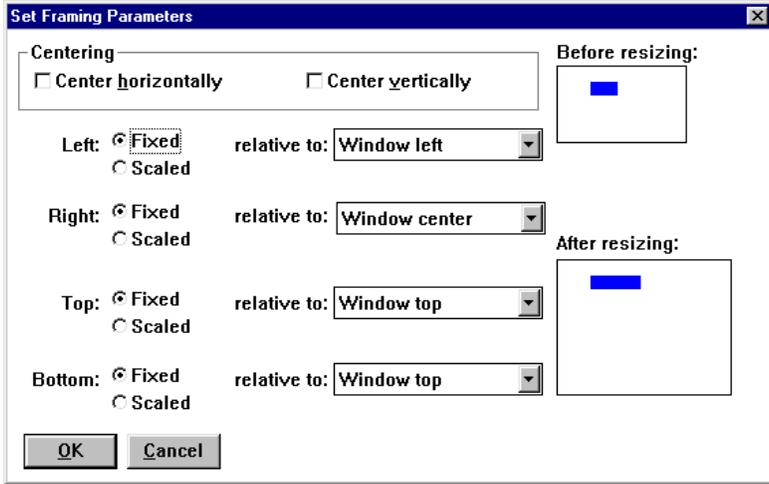


Рис. 17.4. Настройка параметров размера виджета

Верхняя панель слева (с меткой *Centering*) предоставляет возможности простого центрирования компонента — горизонтального или вертикального. Остальные элементы окна позволяют устанавливать характер поведения каждой из его границ при изменении размеров окна, при этом каждая граница ведет себя совершенно независимо от других. Но если компонент центрирован горизонтально или вертикально, то уже нельзя определять поведение соответствующих границ.

Механизм определения характера поведения границ работает одинаково для всех четырех сторон. Например, для левой границы компонента можно установить, будет ли фиксированным (радиокнопка *Fixed*) расстояние от данной стороны компонента до соответствующей стороны окна, или до противоположной (правой) стороны окна, или до вертикального центра, или же относительно противоположной (правой) границы самого объекта. В последнем случае характер поведения по крайней мере одной из сторон должен быть определен относительно самого окна. Например, если характер поведения левой стороны объекта будет установлен относительно своей же правой стороны, то характер поведения правой стороны обязательно должен устанавливаться относительно окна.

Альтернативой к установке фиксированного расстояния границы компонента является установка его пропорционального изменения (радиокнопка *Scaled*). В этом случае данная сторона компонента будет изменяться пропорционально стороне окна. Поскольку поведение каждой стороны можно

определять независимо от поведения других сторон, то в окне реального приложения можно создать много различных вариантов поведения границ его компонентов.

Настройка поведения границ может выполняться сразу для нескольких панелей окна. Для этого, прежде, чем нажимать на кнопку Framing, надо любым образом выделить нужные виджеты. Не бойтесь экспериментировать, поскольку установка характера поведения границ компонентов дает мгновенный эффект и позволяет быстро опробовать несколько вариантов.

17.2.3. Изменение внешнего вида виджета

Многие виджеты тем или иным образом связаны с текстом. Текст изображается с помощью шрифтов, поддерживаемых системой. Мы уже знаем, как изменить используемый виджетом шрифт: надо выделить этот объект, а затем нажать кнопку Font и воспользоваться для выбора нового шрифта специализированным диалоговым окном, после закрытия которого шрифт текста виджета изменится. Как и большинство других команд, эта команда может выполняться для нескольких выбранных объектов. Если шрифт не является изменяемым атрибутом объекта, кнопка Font будет недоступна.

Некоторые из отображаемых объектов имеют несколько стилей своего изображения. Например, статический текст может быть выровнен вправо, влево, или расположен по центру. Левая нижняя панель окна WBP, служащая для определения свойств объектов, включает в себя комбинированный список с меткой Style:, из которого можно выбрать один из доступных стилей изображения выделенного в окне виджета.

При желании всегда можно изменить цвета, используемые в виджете для фона (Backcolor) и символов (Forecolor). Для этого следует нажать на кнопку Color — вторую слева в панели свойств виджета. Цвета, доступные для выделенного виджета, отображаются в двух одноименных панелях возникающего диалогового окна. Если после ряда экспериментов возникает желание вернуться к значениям по умолчанию, достаточно в этих панелях выбрать цвет Default. Эта команда может выполняться одновременно для нескольких выбранных объектов.

Кроме свойств, общих для многих виджетов, есть свойства, присущие только определенным виджетам. Для того чтобы изменить свойства такого виджета, достаточно его выделить в окне и затем нажать кнопку Attribute — кнопку с «медицинским» крестом в панели свойств виджета. Появится специализированный для виджета редактор атрибутов, в котором можно установить желаемые значения. Если объект не имеет специфических редактируемых свойств, кнопка Attribute недоступна.

17.2.4. Установка порядка табуляции

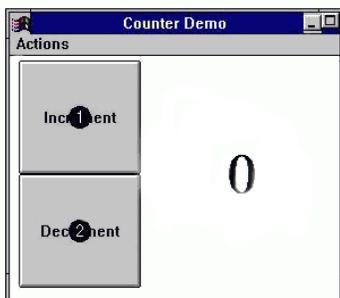


Рис. 17.5. Определение порядка табуляции виджетов.

Порядок, в котором пользователь, работая с клавиатурой, выбирает виджеты в окне, важен для интерфейса. WBP обеспечивает быстрый и простой способ управления порядком перемещения в окне от виджета к виджету с помощью клавиатуры. Если специальный порядок прохождения элементов не определяется, то при нажатии клавиши **[Tab↔]** или сочетания клавиш **[↑Shift]+[Tab↔]** ничего не происходит. Чтобы в создаваемом окне в любой виджет можно было попасть, используя только клавиатуру, и чтобы определить порядок прохождения виджетов, надо воспользоваться редактором порядка табуляции. Его вызов производится нажатием на правую кнопку в нижнем левом углу окна WBP (рядом с кнопкой вызова окна просмотра иерархии классов).

Итак, нажмите эту кнопку. WBP откроет окно редактора порядка табуляции с изображенным в нем редактируемым окном. Если предварительно устанавливался порядок табуляции виджетов окна, то его элементы, доступные с помощью табуляции, будут пронумерованы, начиная с 1 (рис. 17.5). В противном случае элементы пронумерованы не будут.

Чтобы установить или изменить порядок табуляции объектов окна, надо нажать на кнопку Set Tab Order и просто пройти в нужном порядке по элементам окна, нажимая на них левой кнопкой мыши. Все выбранные виджеты будут последовательно пронумерованы. Если такое упорядочивание соответствует вашему желанию, то нажмите кнопку ОК. В противном случае нажмите кнопку Done и повторно выполните процедуру настройки порядка табуляции.

17.2.5. Построение сложных панелей

Одной из мощных возможностей WBP является возможность конструирования составной панели — панели, состоящей из двух или более компонентов, работающих как единое целое. Это позволяет чрезвычайно быстро создавать сложные интерфейсы. Коллекция составных панелей включена в состав WBP. Добраться до них можно через группу составных панелей в палитре виджетов или через пункт Composite из меню Add.

Давайте посмотрим на некоторые панели из этой коллекции, для чего создадим новое окно **WBP**, выбирая пункт **New Window** из меню **File** окна **Transcript**. Изменим размеры окна так, чтобы оно заполнило большую часть области редактирования **WBP**. Теперь из подменю **Add** \triangleright **Composite** выберем объект **NamePane** и разместим его в окне (размещение составной панели ничем не отличается от размещения любого другого виджета, например, кнопки).

Составной объект **NamePane** содержит три экземпляра класса **StaticText**, которые используются в качестве меток, и три поля ввода. Каждую составную панель можно исследовать, дважды щелкая по ней левой кнопкой мыши. Если это проделать с **NamePane**, **WBP** откроет новое окно редактора, в котором можно, например, изменить порядок прохождения его элементов по нажатию клавиши **[Tab \leftrightarrow]**. Но с точки зрения **WBP** это будет уже другая панель. Если после изменений выбрать пункт **Save**, то сделанные изменения составной панели сохранятся для всех экземпляров этого класса. Если же вы хотите, чтобы изменения имели место только для текущего окна, создайте соответствующий подкласс класса **CompositePane** и используйте его экземпляр.

Добавим в окно еще одну составную панель, для чего из меню **Composite** выберем пункт **AddressPane** и расположим этот объект чуть ниже панели **NamePane**. Протестируем это окно. Обратите внимание на одну деталь: если мы попали в поле **First Name** панели **NamePane** и затем нажмем **[Tab \leftrightarrow]**, курсор не покидает эту панель. Единственный способ перебраться в какое-нибудь поле панели **AddressPane** состоит в том, чтобы с помощью мыши явно поместить курсор в одно из них. Теперь использование клавиши **[Tab \leftrightarrow]** приводит к обходу объектов только внутри этой группы полей. Как заставить курсор перемещаться из одной составной панели в другую? Закроем тестируемое окно и откроем редактор порядка табуляции. В нем нажмем на кнопку **Set Tabbing Order** и выберем сначала объект **NamePane**, а затем **AddressPane**. Нажмем кнопку **OK**. Протестируем окно снова. Клавиша **[Tab \leftrightarrow]** теперь перемещает курсор непрерывно и внутри панелей, и между ними.

Составные виджеты можно создавать самостоятельно. Рассмотрим этот процесс на простом примере.

- 1) Откроем новое окно редактора **WBP** и расположим в окне **New Window** три радиокнопки (**RadioButton**).
- 2) Выберем все три кнопки.
- 3) Из меню **File** выберем пункт **Composite Pane** \triangleright **Create**. **WBP** откроет новый редактор на этой группе радиокнопок, трактуя их как единое целое. Сохраним новый объект типа **CompositePane** под каким-либо

именем, например, `ThreeRadioButtons`. При этом появится диалоговое окно с вопросом о замене первоначальных трех радиокнопок на созданный составной объект. Как правило, на этот вопрос стоит отвечать утвердительно. Закроем окно для составной панели и вернемся в окно, в котором начали ее создавать.

Но как и в случае со счетчиком, только «рисунок» составной панели не достаточен для ее эффективного использования. Мало объединить виджеты в новую составную панель, следует написать методы для ее активных элементов. При этом желательно, чтобы составная панель, выступая как единое целое, была полезна и для других панелей вашего приложения, а также могла повторно использоваться в будущем.

Чтобы понять, как надо программировать составные панели, откроем окно просмотра иерархии классов на классе `OkCancelPane`. Его экземпляр — составная панель из горизонтально расположенных кнопок с именами `OK` и `Cancel`. Посмотрим на методы экземпляра `ok` и `cancel`:

cancel

```
self event: #cancel.
```

ok

```
self event: #ok.
```

Единственная строка каждого из методов просто пересылает соответствующее событие владельцу составной панели `OkCancelPane` (как правило — окну приложения). Обратим еще внимание в `OkCancelPane` на метод класса

supportedEvents

```
"Возвращает события, поддерживаемые панелью."
```

```
^super supportedEvents add: #ok; add: #cancel; yourself.
```

Можно сказать, что произошел экспорт событий элементов составной панели на более высокий уровень — уровень составной панели, после чего события, происходящие внутри составных объектов, становятся доступными для других объектов приложения. Далее уже нет никакой разницы между написанием методов для составной панели и для любого другого виджета: методы реализуют реакцию на события, происходящие в составной панели.

Созданные пользователем панели можно сделать доступными для использования при последующем построении интерфейсов, используя специально для этого предназначенное пользовательское подменю `Custom Panes` меню `Add`. Изначально это подменю пусто. Чтобы добавить сюда желаемые панели, надо выбрать пункт `Add Custom Pane...` из меню `Add`, и `WBP` предоставит список всех имеющихся компонентов. Из этого списка надо

выбрать нужный виджет, и он будет добавлен в пользовательское подменю Custom Pane из меню Add. С помощью пункта Remove Custom Pane. . . можно удалить объект из этого подменю. Здесь удобно хранить созданные самостоятельно подклассы классов WBP. Но для подобных целей есть еще одно средство — *альбомы*, описанию которых посвящен следующий раздел.

17.2.6. Альбом

В WBP существует очень интересное и удобное вспомогательное средство — **Scrapbook** (Альбом). Пользователь может иметь несколько альбомов, но в каждый момент времени в редакторе доступен только один. В альбомах можно накапливать часто используемые средства управления, панели и окна, чтобы позже их можно было легко и быстро отыскать и использовать. **Scrapbook** разделен на главы. Каждая глава содержит одну или несколько «страниц»; страница содержит один объект и умеет возвращать свое содержимое. Изначально альбом содержит следующие главы: **Buttons** (Кнопки), **Quick Reference** (Быстрая ссылка), **Radio Buttons** (Радиокнопки) и **Widget Sets** (Набор виджетов). Всегда можно открыть Scrapbook ▷Retrieve. . . и пролистать страницы текущего альбома, чтобы найти нужную панель, кнопку или средство управления. После этого остается только нажать кнопку ОК, чтобы загрузить курсор этим виджетом и одновременно закрыть альбом. Затем можно обычным способом размещать виджет в создаваемом окне. Объекты, хранящиеся в главе **Quick Reference**, доступны непосредственно из меню Scrapbook ▷Quick Reference. Для доступа к остальным главам требуется открыть альбом и выбрать нужную главу.

В альбом можно добавлять новые объекты. Для этого следует выделить сохраняемый объект и выбрать в меню пункт Scrapbook ▷Store. . . Появится диалоговое окно, показывающее имена всех глав в альбоме. Можно создать новую главу, нажав на кнопку New Chapter и указав ее имя, или выбрать главу из списка (WBP не допустит сохранения нового объекта, пока не определена глава). Затем в поле ввода с меткой Page Name надо задать имя страницы для нового объекта. Один объект можно размещать в разных главах. Перед закрытием альбома инструмент WBP автоматически сохранит его новое состояние.

Альбом включает средства для удаления страниц и глав. В меню Scrapbook есть пункты, позволяющие объединить сохраненный альбом с текущим, загрузить ранее сохраненный альбом и создать новый альбом. Текущий альбом является объектом в образе системы и нет необходимости сохранять его явно. Однако если вы желаете сохранить альбом в отдельном файле, можно использовать пункт меню Scrapbook ▷Save. . . .

17.3. Возможности быстрого макетирования

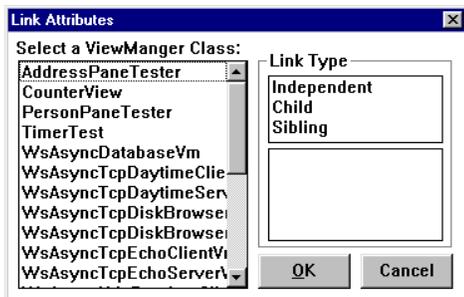


Рис. 17.6. Окно свойств связи.

Одна из сфер применения языка Смолток — быстрое макетирование (или, как еще говорят, быстрое прототипирование) приложений. При этом инструменты типа WBP уменьшают число тех ситуаций, когда нужно «вручную» писать методы, и позволяют создавать панели и меню, которые автоматически связываются с разнообразными окнами или операциями. Большинство простых задач решается почти без программирования. Фактически, даже тот, кто досконально не знает смолтоковских систем, может быстро создать работающий прототип приложения. Некоторые из таких возможностей рассматриваются ниже.

17.3.1. Связывание через LinkButton и LinkMenu

С помощью кнопок и меню окна приложения часто приходится открывать вторичные и диалоговые окна. WBP имеет встроенную возможность, делающую программирование таких кнопок и меню довольно простым. Для решения этой задачи используются кнопки LinkButton. Откроем новое окно WBP и выполним перечисленные ниже действия.

- 1) Добавим кнопку LinkButton в окно. Это можно сделать, выбирая кнопку Link из палитры или пункт меню Add > Button > LinkButton.
- 2) Расположим кнопку в окне.
- 3) Дважды щелкнув на кнопке LinkButton левой кнопкой мыши, откроем связанное с ней диалоговое окно (рис. 17.6). В левой списковой панели этого окна отобразятся окна (подклассы ViewManager), которые определены в вашей системе и могут быть связаны с кнопкой, а в правой панели — характер устанавливаемых отношений между окном, содержащим кнопку LinkButton, и связываемым окном. В тот момент, когда вы выбираете тип отношений, в нижней части правой панели появляется пояснение к данному типу отношений.
- 4) Выберем в панели окон приложение CounterDemo, а в панели отношений зависимости — строку Independent (Независимое) и нажмем на кнопку ОК.

Возвратившись в WBP, увидим, что надпись на кнопке изменилась на имя выбранного класса. При тестировании окна нажатие на кнопку откроет окно CounterDemo. В данном случае окно CounterDemo не будет зависеть от вызвавшего его окна, поэтому можно перемещаться между окнами и закрывать их в любом порядке.

Связывание с пунктом меню вместо кнопки работает точно так же, но процесс создания происходит немного иначе:

- 1) Откроем для строящегося окна редактор Menubar.
- 2) Создадим в окне редактора Menubar (см. рис. 17.3) само меню и его пункты, затем выберем один из пунктов меню и в верхней половине правой панели Item attributes нажмем на кнопку Link... Появится диалоговое окно для установки связей (рис. 17.6), которое мы уже использовали для LinkButton.

Далее действуем по описанной выше методике.

17.3.2. Операции через ActionButton и ActionMenu

Чтобы выполнить некоторые простые действия, можно воспользоваться кнопкой ActionButton или пунктом меню, связанным с кнопкой Action... Эти кнопки подобны ранее описанным LinkButton. Главное отличие состоит в том, что с ActionButton можно сопоставить практически любое действие. Можно рассматривать возможности кнопки LinkButton как частный случай возможностей кнопки ActionButton.

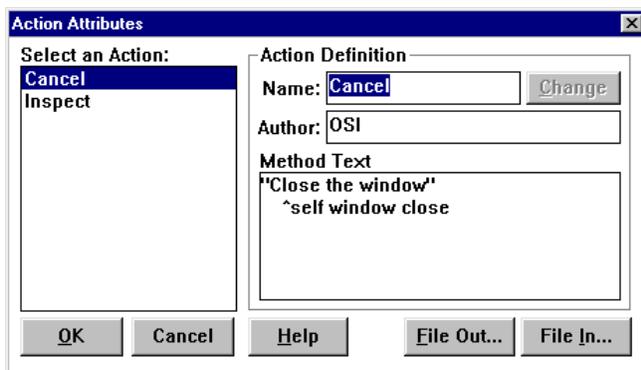
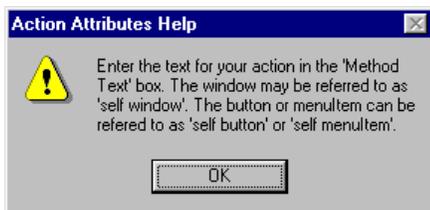


Рис. 17.7. Окно свойств действия.

Создание и программирование кнопки ActionButton напоминает работу с LinkButton. Откроем новое окно WBP и сделаем следующее:

- 1) Добавим кнопку `ActionButton` в строящееся окно. Это можно сделать, используя пункты меню `Add > Button > ActionButton` или выбирая кнопку АСТ из палитры.
- 2) Расположим кнопку в окне.
- 3) Дважды щелкая на кнопке `ActionButton` левой кнопкой мыши, откроем связанное с ней диалоговое окно (рис. 17.7). Действия, для которых методы уже существуют, перечисляются в левой списковой панели. В правой панели с меткой `Action Definition` можно определить новое действие. Для этого следует:
 - указать имя нового действия (одно слово);
 - указать автора метода (необязательно);
 - определить метод для нового действия либо выбирая уже существующий метод из левой списковой панели, либо непосредственно набирая в панели `Method Text` текст нового метода, либо загружая метод из файла с помощью кнопки `File In . . .`

Кнопку `File Out. . .` применяют для сохранения текста метода в файле, чтобы его можно было использовать позже, загружая из файла. При нажатии на кнопку `Help` отображается диалоговое окно, содержащее следующую информацию:



«Введите в текстовую панель `Method Text` текст метода. На само окно можно сослаться посредством `self window`. На кнопку или пункт меню можно сослаться посредством `self button` или `self menulitem`».

Для иллюстрации сказанного рассмотрим пример, в котором с помощью кнопки `ActionButton` будем отображать окно с сегодняшней датой. Выполним следующие действия:

- 1) расположим в редактируемом окне кнопку `ActionButton` и дважды щелкнем по ней, открывая диалоговое окно;
- 2) в поле ввода с меткой `Name` введем имя метода `showDate` и нажмем кнопку `Add`, после чего новое имя появится в левой панели;
- 3) заполним (если есть желание) поле `Author`;
- 4) в панели `Method Text` введем текст метода (комментарий вводить не обязательно):

"Отобразить на экране диалоговое окно с сегодняшней датой."
`MessageBox message: 'Today is ', Date today printString`

- 5) нажмем сначала кнопку **Change**, запоминая введенный текст метода, а затем нажмем кнопку **OK**.

Остается протестировать созданное окно и убедиться, что все работает правильно. Это хороший пример того, как можно создавать небольшие методы непосредственно в **WBP**, не пользуясь браузером иерархии классов. Совершенно аналогично создаются методы, связанные с пунктами меню. Но будьте внимательны: инструмент не предусматривает прямого способа удаления метода, созданного для кнопки **ActionButton**. Придется обратиться к браузеру иерархии классов и с его помощью удалить из класса **WBAction** не нужные для дальнейшей работы методы экземпляра (все они имеют имена с префиксом **action**).

17.3.3. Преобразование виджетов

Иногда при создании пользовательского интерфейса бывает очень полезно поэкспериментировать с разными типами виджетов для решения проблем, подобных следующей: как лучше представить пользователю список элементов: в виде списковой панели или в виде набора радиокнопок?

В **WBP** в таком случае виджеты можно подвергать морфингу, преобразуя их из одного типа в другой, с помощью команды **Edit ▷ Morph** или ее клавиатурного эквивалента — **[Ctrl]+[M]**. По этой команде появляется диалоговое окно со списком всех классов панелей, известных **WBP**. Система преобразует тип текущего виджета к типу выбранной панели. При переходе от «старого» типа к «новому» всегда сохраняется столько информации об объектах и их поведении, сколько имеет смысл сохранить для «нового» типа. Любая информация, специфичная для «старого» типа виджетов и не соответствующая «новому» типу, будет утеряна, и если придется вернуться к исходному типу, то придется заново вводить потерянную информацию.

Поэтому другой, более предпочтительный способ морфинга — вызов через подменю **Morph** контекстного меню виджета. В подменю **Morph** попадают только те классы панелей, в которые можно с минимальными потерями преобразовать текущий виджет.

Рассмотрим простой пример. Откроем новое окно **WBP** и выполним следующие действия:

- 1) выберем панель **RadioButtonGroup** командой меню **Add ▷ Composite ▷ RadioButtonGroup** и разместим выбранную панель в окне;
- 2) дважды щелкнем по панели, для того чтобы открыть диалоговое окно, которое используется для создания радиокнопок внутри панели **RadioButtonGroup**;

- 3) определим кнопки **Red**, **Green**, **Blue**, вводя их имена в верхнем поле ввода и нажимая после каждого ввода кнопку **Add**; нажмем кнопку **OK**, закрывая диалоговое окно;
- 4) сохраним окно под каким-либо именем;
- 5) выберем объект **RadioButtonGroup** и каким-либо способом преобразуем его в **ListBox** (через мгновение **RadioButtonGroup** преобразуется в **ListBox** с тем же содержимым);
- 6) протестируем окно и убедимся, что все работает правильно.

Пользуясь морфингом, вы убедитесь, что трансформация панелей — одна из мощных возможностей, реализованных в **WBP**.

17.4. Методы **createViews** и **open**

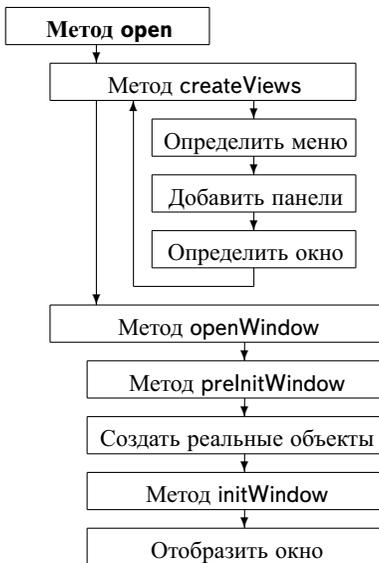


Рис. 17.8. Метод **open**

Метод **createViews**, сгенерированный **WBP**, нельзя редактировать. Любые его изменения будут проигнорированы, если потом снова воспользоваться инструментом **WBP**. Но желание изменить этот метод все же появляется. Часто возникают ситуации, когда требуется управлять такими свойствами окна приложения, которые невозможно определить в **WBP** (например, отключением пунктов меню или кнопок, добавлением панелей с динамическими данными). В таких ситуациях следует не менять метод **createViews**, а воспользоваться специально разработанными для этих целей методами **preInitWindow** и **initWindow**, которые выполняются при создании окна приложения. Порядок, в котором эти методы вызываются, очень важен. Чтобы понять место и значение этих методов, посмотрите на рис. 17.8, который показывает все методы, вызываемые окном приложения, с того момента, когда ему послано сообщение **open** и до появления окна на экране. Метод **open**, концентрируя в себе самое главное, вызывает сначала метод **createViews**, а затем метод **openWindow**. Последний вызывает методы **preInitWindow** и

`initWindow`, позволяющие в различных фазах инициализации выполнить дополнительный код, определяемый программистом.

Метод `preInitWindow` редко используется в приложениях. Он нужен для добавления тех панелей и меню, которые не может обрабатывать **WBP**, но которые должны определяться до построения «реального» окна. *Smalltalk Express* при построении окон и управлении ими существенно использует операционную систему. Любая используемая панель является не только объектом смолтоковской системы, но и объектом (структурой данных) операционной системы. Когда создается окно с помощью метода `open`, метод `preInitWindow` выполняется после того, как все панели в окне были созданы как смолтоковские объекты, но прежде, чем будут созданы «реальные» объекты операционной системы. По этой причине, например, блокировка и активизация пунктов меню или кнопок на данном этапе создания окна в методе `preInitWindow` производиться не может.

Для дополнительной настройки виджетов используется метод `initWindow`, который, как видно из рис. 17.8, действует после того, как *все* объекты окна уже созданы, но перед их визуальным отображением на экране. Поэтому, если внутри метода `initWindow` попытаться добавить в окно новые виджеты, этого не произойдет. Но метод `initWindow` идеально подходит для решения таких задач, как определение состояния виджетов или установка содержимого панелей с динамическими данными. Эти данные не могут быть определены в **WBP**, поскольку они не известны до начала выполнения приложения.

Для примера предположим, что надо сделать недоступным пункт `editObject` из меню `Object`, основываясь на состоянии некоторой переменной экземпляра (скажем, `theObject`), которая получает свое значение только в момент открытия окна. Пусть переменная инициализируется и окно открывается методом `openOn:`, который может иметь следующий вид:

openOn: anObject

```
theObject := anObject.  
self open.
```

Информацию о состоянии переменной `theObject` для организации блокировки надо использовать до отображения окна, поэтому воспользуемся методом `initWindow`, который создадим в следующем виде:

initWindow

```
(theObject isEditable)  
ifFalse: [(self menuTitled: 'Object') disableItem: 'editObject'].]
```

Пункт меню `editObject` будет недоступен, если объект не поддерживает редактирование. Совершенно аналогично можно поступать и с кнопками.

ГЛАВА 18

ИНТЕРФЕЙС ДЛЯ FINANCIALBOOK

Вспользуемся инструментом WBP и создадим интерфейс пользователя для домашней бухгалтерии — экземпляра класса `FinancialBook`, который будет моделью данного приложения. Эта модель более сложная, чем словари, которые были моделями в предыдущих примерах. Назовем класс окна приложения именем `WBPFinancialBook`. Всю информацию для своих панелей интерфейс будет получать от модели, хранящейся в его единственной переменной `financialBook`. Приложение должно решать следующие задачи:

- отображать общую информацию о доходах и расходах;
- отображать страницу для доходов и страницу для расходов, которые позволят: просматривать все статьи доходов и расходов; вводить новые статьи доходов и расходов; добавлять новую сумму в статьи.

18.1. Эскиз приложения

Домашняя бухгалтерия	
Всего получено:	<input type="text"/>
Всего потрачено:	<input type="text"/>
Сумма наличных:	<input type="text"/>
<input type="button" value="Доходы"/>	<input type="button" value="Расходы"/>
<input type="button" value="Выход"/>	

Рис. 18.1. Эскиз основного окна приложения

Начнем работу с создания эскиза приложения и его окон. Наше приложение будет открывать три окна: основное окно, отображающее общую информацию о финансовом состоянии, и два вторичных окна, которые будут открываться из основного окна. Первое из них будет страницей доходов, а второе — страницей расходов. Эти окна одинаковы по своей функциональности, поскольку должны однотипно управлять одинаковыми по структуре словарями `incomes` и `expenditures`, но будут отличаться некоторой (несущественной) информацией, указывающей пользователю на то, с каким словарем он имеет дело.

18.1.1. Эскиз основного (первичного) окна

Итак, основное окно должно только отображать общую информацию и позволять пользователю открывать страницу доходов или расходов. Никаких действий пользователя по отношению к модели основное окно не

поддерживает, но оно должно корректно закрываться, сохраняя текущее состояние модели приложения.

Будем отображать в основном окне в соответствующих полях ввода три суммы: общую сумму доходов (она хранится в переменной `totalIncome` объекта `financialBook`), общую сумму расходов (она хранится в переменной `totalExpenditure` объекта `financialBook`) и сумму наличных денег (которую объект `financialBook` возвращает в ответ на сообщение `cashOnHand`). Сопроводим поля ввода метками, отражающими характер отображаемой информации. Чтобы открывать вторичные окна, воспользуемся двумя кнопками с метками, указывающими на тип открываемого окна. Еще одну кнопку используем для того, чтобы закрывать окно. Таким образом, основное окно должно иметь вид, представленный на рис. 18.1, а его функциональные возможности определяются событиями и связанными с ними методами, приведенными в табл. 18.1.

Таблица 18.1. События основного окна приложения

Виджет	Событие	Имя метода	Цель метода
InputField1	getContents	totalIncome:	Отобразить общую сумму доходов.
InputField2	getContents	totalExpenditure:	Отобразить общую сумму расходов.
InputField3	getContents	cashOnHand:	Отобразить сумму наличных.
Button1	clicked	openIncomes:	Открыть вторичное окно на аргументе, который является словарем доходов.
Button2	clicked	openExpenditures:	Открыть вторичное окно на аргументе, который является словарем расходов.
Button3	clicked	close	Закреть первичное окно, сохраняя модель.

18.1.2. Эскиз окна выполняемой операции

Оба вторичных окна будут экземплярами класса `DlgFinancialBook`, который сделаем подклассом в `WindowDialog`. Значит возвращение в основное окно до закрытия вызванного им вторичного окна будет невозможно. Моделью вторичного окна будет словарь, который вычисляется по модели первичного окна и всего приложения в целом. Поскольку основное назна-

чение этого окна — обеспечить ввод новой информации и в модель приложения, и в конкретный словарь, кроме переменной экземпляра `dictionary`, хранящей модель вторичного окна, определим переменные экземпляра `sum`, `selectedItem` и `type`, которые будут хранить соответственно итоговую сумму, введенную во вторичном окне, выделенный в словаре элемент и «тип выполняемой операции».

Таблица 18.2. События вторичного окна приложения

Виджет	Событие	Имя метода	Цель метода
<code>aWindow</code>	<code>activate</code>	<code>titlesWindow</code>	Определить соответствующую выполняемой операции заголовок окна и метку списковой панели.
<code>ListPane</code>	<code>getContents</code>	<code>getList:</code>	Отобразить содержимое списковой панели.
	<code>select</code>	<code>itemSelected</code>	Выделить выбранный пользователем элемент списка.
<code>EntryField1</code>	<code>getContents</code>	<code>getAmount:</code>	Отобразить сумму, связанную с выбранным элементом списка.
<code>Button1</code>	<code>clicked</code>	<code>addAmount:</code>	Открыть окно подсказчика, в котором пользователь введет добавляемую им сумму.
<code>Button2</code>	<code>clicked</code>	<code>ok:</code>	Закрыть вторичное окно, сохраняя сделанное.
<code>Button3</code>	<code>clicked</code>	<code>close:</code>	Закрыть вторичное окно, отказываясь от введенной информации.

Окно должно отображать список источников доходов или причин расходов и сумму, связанную с выделенным элементом списка. Кроме того, окно обеспечивает возможность добавить к выделенной записи новую сумму, а также возможность закрыть себя с указанием на сохранение введенной информации или ее отмену. Поэтому внешний вид вторичного окна приложения должен быть примерно таким, как на рис. 18.2. Строка заголовка вторичного окна `typeOfPage` и метка его списковой панели `typeOfList`: будут зависеть от того, для каких целей открывается вторичное окно. Как мы определим позже, строка заголовка окна может быть строкой «Страница доходов» или строкой «Страница расходов», а строка панели `typeOfList`, соответственно, строкой «Источники доходов:» или строкой «Причины расхо-

дов:». Чтобы вторичное окно можно было тестировать независимо от первичного окна и использовать, если потребуется, самостоятельно, по умолчанию определим заголовок в виде строки Книга учета, а метку списковой панели в виде строки Список ключей:.

Так как определяется вторичное окно приложения, его класс, переменные и все методы должны рассматриваться как частные. Не предполагается, что пользователи приложения будут обращаться к классу и его экземплярам явно, не используя для этого первичного окна. Тем не менее мы постараемся сделать реализацию вторичного окна как можно меньше зависящей от первичного. А функциональные возможности вторичного окна определим событиями и связанными с ними методами, приведенными в табл. 18.2.

Чтобы иметь возможность добавлять в списковую панель новые записи, установим в нее всплывающее меню с одним единственным элементом — Новая запись (для чего воспользуемся редактором меню).

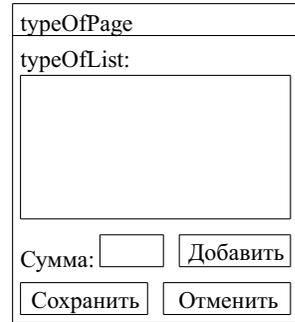


Рис. 18.2. Эскиз вторичного окна приложения

18.2. Реализация приложения

Приложение будем строить, пользуясь инструментом WindowBuilder Pro, который достаточно подробно описан в предыдущей главе. Поэтому мы не будем здесь отвлекаться на описание того, как выполнять в WBP ту или иную стандартную операцию.

Таблица 18.3. Свойства первичного окна приложения

Виджет	Текст	Стиль	Имя
aWindow	Домашняя бухгалтерия		mainView
StaticText1	Всего получено:	leftJustified	
StaticText2	Всего потрачено:	leftJustified	
StaticText3	Сумма наличных:	leftJustified	
EntryField1	0	readonly	incomes
EntryField2	0	readonly	expenditures
EntryField1	0	readonly	cashOnHand
Button1	Доходы	pushButton	
Button2	Расходы	pushButton	
Button2	Выход	pushButton	

18.2.1. Предварительная реализация основного окна

Откроем инструмент **WBP**. В новом окне размером 300×200 разместим по три виджета типа **StaticText**, **EntryField** и **Button** так, как это показано на рис. 18.1. Выровняем их и установим для каждого виджета его свойства. Сначала установим свойства для внешнего представления виджета так, как они описаны в табл. 18.3, а затем зададим обработчики событий, описанные в табл. 18.1. Выделим само окно, откроем редактор порядка табуляции и установим следующий порядок: **Доходы** → **Расходы** → **Выход**.

Теперь следует определить обработчики событий и другие методы. Для этого откроем из **WBP** окно просмотра иерархии классов. Поскольку класс окна приложения еще не назван, появится диалоговое окно, в котором введем для этого класса имя **WBPFFinancialBook**, а также укажем, что это подкласс класса **WindowDialog**, чтобы пользователь не мог изменять размеры окна. В открывшемся окне просмотра иерархии классов добавим в определение класса **WBPFFinancialBook** переменную экземпляра **financialBook** и сохраним следующее определение класса:

```
WindowDialog subclass: #WBPFFinancialBook
  instanceVariableNames: 'financialBook '
  classVariableNames: ' '
  poolDictionaries: 'ColorConstants WBConstants '
```

В соответствии с нашими установками в окне **WBP** система уже создала в этом классе заголовки необходимых методов. Но прежде чем завершить их, напишем метод класса, создающий его новый экземпляр.

new

```
"Возвращает новый экземпляр окна с пустым экземпляром
класса FinancialBook в качестве модели."
^ super new financialBook: (FinancialBook new).
```

Теперь по порядку реализуем все необходимые методы экземпляра. Начнем с методов доступа к единственной переменной экземпляра (метод установки значения мы уже использовали в методе **new**), а затем реализуем все обработчики событий виджетов основного окна.

financialBook

```
"Возвращает значение переменной financialBook."
^ financialBook.
```

financialBook: aFinancialBook

```
"Присваивает переменной financialBook новое значение."
^ financialBook := aFinancialBook.
```

Обработчики события #getContents

totalIncome: aPane

"Обработчик события #getContents поля ввода с именем 'incomes', отображающего общую сумму доходов."

```
aPane contents: (financialBook totalIncome) printString.
```

totalExpenditure: aPane

"Обработчик события #getContents поля ввода с именем 'expenditures', отображающего общую сумму расходов."

```
aPane contents: (financialBook totalExpenditure) printString.
```

cashOnHand: aPane

"Обработчик события #getContents поля ввода с именем 'cashOnHand', отображающего наличную сумму денег."

```
| cash |
```

```
cash := financialBook cashOnHand.
```

```
aPane contents: cash printString.
```

```
cash <= 0
```

```
ifTrue: [ (self paneNamed: 'OpenExpenditures') disable ]
```

```
ifFalse:[ (self paneNamed: 'OpenExpenditures') enable ].
```

В конце последнего метода происходит проверка суммы наличных, и кнопка расходов основного окна блокируется (делается недоступной для пользователя), если это число не является положительным.

Обработчики события #clicked

close: aPane

"Обработчик события #clicked кнопки Выход.

Закрывает первичное окно приложения."

```
self close.
```

Поскольку класс вторичных окон приложения еще не создан, для предварительного тестирования первичного окна напомним методы-заглушки, которые вместо вторичных окон приложения открывают окна сообщений.

openIncomes: aPane

"Обработчик события #clicked кнопки Доходы. Открывает вторичное окно для страницы доходов."

```
^ MessageBox message: 'Страница доходов'.
```

openExpenditures: aPane

"Обработчик события #clicked кнопки Расходы. Открывает вторичное окно для страницы расходов."

```
^ MessageBox message: 'Страница расходов'.
```

В соответствии с эскизом (см. рис. 18.1) и заявленной в таблицах 18.1 и 18.3 функциональностью предварительная реализация основного окна завершена. Остается его протестировать и убедиться, что все работает.

18.2.2. Реализация вторичного окна

В новом диалоговом окне размером 300×370 разместим по одному виджету `ListPane` и `EntryField`, два виджета `StaticText`, которые будут метками двух первых виджетов, и три виджета типа `Button` так, как это показано на рис. 18.2. Выровняем в окне их расположение и размеры, установим для каждого виджета свойства, определяющие его внешний вид в соответствии со следующей таблицей.

Таблица 18.4. Свойства вторичного окна приложения

Виджет	Текст	Стиль	Имя
<code>aWindow</code>	Книга учета		<code>dialogWindow</code>
<code>StaticText1</code>	Список ключей:	<code>leftJustified</code>	<code>titleLabel</code>
<code>StaticText2</code>	Сумма:	<code>leftJustified</code>	
<code>EntryField</code>		<code>readonly</code>	<code>amount</code>
<code>Button1</code>	Добавить	<code>pushButton</code>	<code>add</code>
<code>Button2</code>	Сохранить	<code>pushButton</code>	<code>ok</code>
<code>Button2</code>	Отменить	<code>pushButton</code>	<code>cancel</code>

Установим следующий порядок прохождения кнопок окна: **Добавить** → **Сохранить** → **Отменить**. Теперь следует задать обработчики событий виджетов, перечисленные в табл. 18.2. Затем откроем окно просмотра иерархии классов и в появившемся диалоговом окне введем для этого класса имя `DlgFinancialBook`. В текстовой панели окна просмотра иерархии классов добавим в определение класса `DlgFinancialBook` четыре переменные экземпляра: `dictionary`, `selectedItem`, `sum`, `type` и сохраним изменения:

```
WindowDialog subclass: #DlgFinancialBook
  instanceVariableNames: ' dictionary selectedItem type sum '
  classVariableNames: ''
  poolDictionaries: 'ColorConstants WBConstants '
```

Сначала определим метод класса, создающий новый экземпляр для словаря (модели окна), указанного первым аргументом, и операции, указанной вторым аргументом. По смыслу приложения параметр `aString` может принимать всего два значения: `'incomes'` и `'expenditures'`. Этот аргумент нами используется для корректного представления заголовка вторичного окна и метки его списковой панели.

newOn: aDictionary forOperation: aString

"Создает новый экземпляр класса со словарем aDictionary в качестве модели, который используется для выполнения операции, определяемой строкой aString."

^super new setDictionary: aDictionary forOperation: aString.

Напишем еще один метод класса, создающий экземпляр с аргументами по умолчанию. Он может оказаться полезным при тестировании вторичного окна из среды инструмента WBP.

new

^self newOn: (Dictionary new) forOperation: 'incomes'.

Первыми среди методов экземпляра реализуем метод инициализации переменных экземпляра setDictionary:forOperation: и необходимые далее методы доступа к переменным dictionary и sum.

setDictionary: aDictionary forOperation: aString

"Инициализирует переменные получателя."

dictionary := aDictionary.

sum := 0.

type := aString.

dictionary

"Возвращает значение переменной dictionary."

^dictionary.

dictionary: aDictionary

"Устанавливает значение переменной dictionary."

dictionary := aDictionary.

sum

"Возвращает значение переменной sum."

^sum.

sum: aNumber

"Устанавливает значение переменной sum."

sum := aNumber.

Теперь реализуем обработчики событий виджетов вторичного окна.

Обработчики события #activate

Метод экземпляра titleWindow: нужен для того, чтобы, в зависимости от выполняемой в окне операции над моделью, определить заголовок самого вторичного окна и метку (виджет StaticText1 с именем 'titleList') содержащегося в окне списка.

titleWindow: aPane

"Определяет заголовок вторичного окна и метку содержащегося в окне списка."

type = 'incomes'

ifTrue: [aPane labelWithoutPrefix: 'Страница доходов'.

(self paneNamed: 'titleList') contents: 'Источники доходов:'].

type = 'expenditures'

ifTrue: [aPane labelWithoutPrefix: 'Страница расходов'.

(self paneNamed: 'titleList') contents: 'Причины расходов:'].

Рассмотрим выражение (self paneNamed: 'titleList') contents: . . . Когда в WBP создавалось окно, некоторые его панели получили имена (их мы вводили в поле ввода окна WBP с меткой Name:). Теперь к панели можно обращаться по имени, пользуясь сообщением paneNamed:, посылаемым владельцу панели. В этом выражении определяется текст, который отображается виджетом с именем 'titleList'.

*Обработчики события #getContents***getList: aPane**

"Обработчик события #getContents панели ListPane с именем 'listItem'. Отображает в панели ключи словаря."

aPane contents: dictionary keys asSortedCollection.

getAmount: aPane

"Обработчик события #getContents поля EntryField с именем 'amount'. Отображает в поле значение выбранного ключа."

selectedItem isNil

ifFalse: [aPane contents: (dictionary at: selectedItem) printString].

*Обработчик события #select списковой панели***itemSelected: aPane**

"Обработчик события #select панели ListPane. Запоминет выбор пользователя и изменяет значение, отображающееся в поле ввода с именем 'amount'."

selectedItem := aPane selectedItem.

self changed: #getAmount:.

*Обработчики событий #clicked***addAmount: aPane**

"Обработчик события #clicked кнопки Добавить. Запрашивает добавляемую сумму, затем изменяет модель и отображает новое значение суммы, соответствующее выбранному ключу."

```

| amount addAmount |
selectedItem isNil ifTrue: [^ self].
amount := dictionary at: selectedItem.
addAmount := Prompter prompt: 'Введите добавляемую сумму:'
    defaultExpression: String new.
(addAmount isNil or: [addAmount isNumber not])
    ifTrue: [^ self].
dictionary at: selectedItem
    put: (amount + (addAmount roundTo: 0.01)).
sum := sum + addAmount.
self changed: #getAmount:.

```

ok: aPane

"Обработчик события #clicked кнопки Сохранить.
 Не меняет созданной модели. Закрывает окно."
 self close.

cancel: aPane

"Обработчик события #clicked кнопки Отменить.
 Создает недопустимую модель (nil). Закрывает окно."
 dictionary := nil.
 self close.

Напомним, что выражение `Prompter prompt: aString defaultExpression: defaultString` по нажатию кнопки ОК возвращает объект, являющийся результатом *вычисления* ввода пользователя. Введенная пользователем информация должна возвращаться приложению методом `addAmount:` как число.

Построение меню списковой панели

Осталось встроить в списковую панель меню с единственной опцией и определить для нее метод экземпляра. Закроем окно просмотра иерархии классов и вернемся в окно WBP с вторичным окном. Чтобы построить меню списковой панели `ListPane`, выделим ее и вызовем редактор меню. Напомним, что редактор меню можно вызвать, пользуясь пунктом `Menubar` из всплывающего меню рабочей области окна WBP.

В левом верхнем поле ввода напечатаем имя пункта Новая запись, а в правом верхнем поле ввода с меткой `Selector:` — имя метода экземпляра `addNewRecord`, который будет выполняться при выборе пользователем данного пункта всплывающего меню списковой панели. Закроем редактор меню, нажимая ОК. Меню создано.

Снова откроем окно просмотра иерархии классов на классе `DlgFinancialBook` и определим метод экземпляра:

addNewRecord

"Пользователь выбрал пункт меню Новая запись. Метод запрашивает у пользователя новый ключ, сохраняет его в модели, отображает новую модель в окне."

| key |

key := Prompter prompt: 'Введите ключ новой записи:'
 default: String new.

(key isNil or: [dictionary includesKey: key]) ifTrue: [^ self].

selectedItem := key.

dictionary at: key put: 0.

self changed: #getList;;

 changed: #getList: with: #selection: with: key;

 changed: #getAmount:.

Построение вторичного окна завершено. Протестируйте его. Сохраните образ системы, а если необходимо, сохраните все созданные классы в виде отдельных файлов (пользуясь пунктом File Out... из меню Classes).

18.2.3. Окончательная реализация основного окна

После построения вторичного окна можно завершить реализацию основного окна, заменяя методы-заглушки `openIncomes:` и `openExpenditures:` полнофункциональными методами. В этих методах открывается вторичное окно, в котором происходит ввод новой информации в соответствующий словарь. Когда вторичное окно закрывается, проверяется, какой словарь хранится в нем. Если это nil (пользователь нажал во вторичном окне кнопку Отменить), ничего не происходит. В противном случае производятся соответствующие изменения в модели предметной области первичного окна. Здесь нам потребуются методы доступа ко всем переменным экземпляра *FinancialBook*, которые читатель должен был создать самостоятельно.

openIncomes: aPane

"Обработчик события clicked кнопки Доходы. Открыть вторичное окно для страницы доходов. После закрытия изменить, если надо, значение переменной financialBook."

| dic sum answer |

answer := (DlgFinancialBook newOn: (financialBook incomes)
 forOperation: 'incomes') open.

dic := answer dictionary.

dic isNil ifFalse: [sum := answer sum.

 financialBook incomes: dic.

 financialBook totalIncome:

```
(financialBook totalIncome + sum)].
self changed: #totalIncome;;
changed: #cashOnHand:.
```

openExpenditures: aPane

```
"Обработчик события clicked кнопки Расходы. Открыть
вторичное окно для страницы доходов. После закрытия
изменить, если надо, значение переменной financialBook."
| dic sum answer |
answer := (DlgFinancialBook
           newOn: (financialBook expenditures)
           forOperation: 'expenditures') open.
dic := answer dictionary.
dic isNil ifFalse: [sum := answer sum.
                   financialBook expenditures: dic.
                   financialBook totalExpenditure:
                     (financialBook totalExpenditure + sum)].
self changed: #totalExpenditure;;
changed: #cashOnHand:.
```

18.3. Как организовать работу приложения

Созданный нами интерфейс имеет недостаток: он не сохраняет свою модель в случае закрытия окна. Рассмотрим один из вариантов исправления этой ситуации. Когда создавался класс `FinancialBook`, мы определили глобальную переменную `FamilyFinancialBook1999`, в которой решили хранить собственную книгу по учету доходов и расходов, и поместили ее в системный словарь `Smalltalk`. После создания интерфейса нам бы хотелось иметь возможность открывать интерфейс на этой книге и в ней сохранять новую информацию. Напишем в классе `WBPFFinancialBook` еще один метод класса, решающий поставленную задачу.

openOnMyFamilyBook

```
"Открывает приложение на глобальной
переменной FamilyFinancialBook1999."
| book |
book := super new
       financialBook: FamilyFinancialBook1999.
book open.
^ book
```

Теперь для работы с финансовой книгой достаточно выполнить выражение `WBPFinancialBook openOnMyFamilyBook`, а, выходя из системы, сохранить образ. Можно найти и другие способы сохранения приложением своей новой модели, например, в файле.

В заключение отметим, что в книге мы рассмотрели далеко не все возможности смолтоковских систем и их инструментов (в том числе *Smalltalk Express* и *WBP*), не все базовые классы. Но мы надеемся, что у вас теперь есть достаточный опыт работы со смолтоковской системой, чтобы самостоятельно решать те проблемы, которые возникнут перед вами при создании собственных приложений.

18.4. Задания для самостоятельной работы

1. В классе `WBPFinancialBook` напишите метод класса `openOn: aFinancialBook`, который открывал бы новый экземпляр приложения с объектом `aFinancialBook` в качестве модели.
2. Создайте необходимые методы в классе `WBPFinancialBook` и измените существующие так, чтобы при запуске приложение запрашивало пользователя о том, из какого файла взять модель приложения, а при закрытии основного окна приложения — о том, в каком файле сохранить модель приложения. Для этого самостоятельно изучите классы `File` и `ObjectFiler`.
3. Если были выполнены задания для самостоятельной работы, приведенные в конце гл. 12, и создан класс `FinancialBook`, учитывающий даты получения конкретных доходов и расходов по каждой статье (ключу словаря), измените приложение `WBPFinancialBook` так, чтобы вторичное окно могло проинформировать пользователя о всех суммах и датах, относящихся к выбранному ключу.

ПРИЛОЖЕНИЕ

МИР ЯЗЫКА СМОЛТОК

Воздержанность в еде рождена или заботой о здоровье, или невозможностью много съесть.

Франсуа де Ларошфуко

1. Коммерческие реализации

Коммерческие реализации сегодня определяют основное направление развития языка Смолток и построенных на его основе систем. Описать их все сложно, поскольку о некоторых системах трудно получить достоверную информацию. Однако основные характеристики систем, оказавших наибольшее влияние на развитие Смолтока, мы приведем, наряду с информацией о некоторых новых реализациях Смолтока.

Из-за недостатка места мы опустили здесь многие известные смолтокоподобные системы, такие как ObjectStudio или Smalltalk/MT, поскольку, хотя они используют «почти» Смолток, их язык и библиотеки слишком сильно отличаются как от *Smalltalk-80*, так и от стандарта языка.

1.1. Smalltalk-80

Система Smalltalk-80 появилась на рынке программ в 1983 году. Так как образ системы имел машиннезависимый формат, она могла быть реализована на любой платформе, удовлетворяющей следующим требованиям [4]:

- монохромный растровый дисплей;
- указывающее устройство (мышь) с тремя кнопками;
- клавиатура, возвращающая коды ASCII, или клавиатура ALTO;
- жесткий диск;
- миллисекундный таймер;
- часы реального времени с разрешением в одну секунду.

На выбранной платформе должна реализовываться только виртуальная машина Smalltalk-80. Первоначально она существовала для рабочих станций Xerox и машин фирм, участвовавших в рецензировании. Хотя цена на систему Smalltalk-80 и ее требования к аппаратным средствам не позволили ей получить широкого распространения, она сразу же нашла применение в

создании крупных систем, требовавших графического интерфейса пользователя и возможностей по сложной обработке данных различной природы. Система постоянно развивалась. Большой вклад в этот процесс внесла лаборатория искусственного интеллекта Фэйрчайлда, где была усовершенствована виртуальная машина для процессоров Motorola 680x0.

Выделение из корпорации Xerox фирмы ParcPlace привело к более эффективному распространению системы. Стал шире набор техники, для которой существовала виртуальная машина. Фирмы, участвовавшие в тестировании и рецензировании, тоже не остались в стороне от совершенствования системы. Так, фирма Tectronix, выпускавшая графические рабочие станции и цветные принтеры, добавила в Smalltalk-80 поддержку цвета. Фирма Apple Computers, компьютеры которой с первых моделей отличались хорошей графикой и звуком, стала развивать звуковые возможности системы. Несколько членов группы по проблемам обучения, принимавших активное участие в создании Смолтока, перешли в Apple, где занялись работами над пользовательским интерфейсом. Эти усилия завершились созданием графических пользовательских интерфейсов машин Lisa и Macintosh. В Apple была создана система Fabrik [9] — одна из первых систем визуального программирования, работавшая в среде Smalltalk-80.

1.2. VisualWorks

Убеждение, что смолтоковские системы должны сосуществовать на компьютерах с другими приложениями и выполняться под управлением установленного на графической станции менеджера окон, потребовало существенной переработки Smalltalk-80. Эта переработка началась с выпущенной в 1989 году системы *ObjectWorks* и успешно завершилась в 1991 году выпуском системы *VisualWorks*. Серия продуктов *VisualWorks* обладает большими возможностями, и после слияния ParcPlace и Digitalk стала одним из основных продуктов объединенной компании ObjectShare.

Долгое время версия *VisualWorks 2.x* была одной из самых распространенных в мире смолтоковских систем. Ее библиотека классов восходит к библиотеке Smalltalk-80 и удивительно стабильна. В последующей версии *VisualWorks 3.0* фактически сохранились без изменений все классы системы, не относящиеся к виджетам или графике, в то время как библиотека классов пользовательского интерфейса была существенно изменена. Инструменты *VisualWorks*, связанные с разработкой приложений баз данных, поддержкой жизненного цикла программы, и прочие утилиты не уступают инструментам системы *IBM VisualAge*. Но в ней нет таких мощных средств визуального программирования как в системе *VisualAge for Smalltalk*.

Система VisualWorks 3.0 — расширяемый комплекс приложений [23], состоящий из ядра среды (собственно VisualWorks) и множества подключаемых дополнений:

VisualWave Developer. Средство разработки интерактивных web-приложений, поддерживающих CGI, NS API, IS API и взаимодействующих через сеть с апплетами. Поддерживает фреймы и JavaScript.

VisualWave Server. Объектно-ориентированный web-сервер приложений. Может работать совместно практически со всеми распространенными http-серверами, включая Apache.

Distributed Smalltalk. Обеспечивает классы и средства разработки для создания распределенных приложений.

DLL & C Connect. Обеспечивает доступ из VisualWorks к функциям разделяемых библиотек и создание новых примитивов.

Advanced Tools. Содержит набор полезных утилит, включающий про-файлеры, документаторы, новые браузеры с расширенной функциональностью, верификаторы кода, генератор лексических анализаторов и среду для создания расширенных числовых классов, поддерживающую системы чисел, не обладающие линейной упорядоченностью (например, комплексные или пополненные бесконечностью).

Database Connect. Обеспечивает интерфейсы к СУБД IBM DB2, Oracle, Informix и Sybase. Эти интерфейсы включают поддержку схем баз данных и нестандартных свойств, обеспечивающих повышение эффективности доступа к данным. Сюда же включено дополнение *Database Application Creator* — визуальная среда создания БД и отображения БД на объекты, построения GUI для приложений баз данных. В архивах общедоступного кода есть пакеты, обеспечивающие доступ к базам данных через ODBC, интерфейс к СУБД MySQL и непосредственную работу с файлами dBase.

COM Connect. Поддерживает работу с OLE-приложениями.

Фирма ObjectShare на основе VisualWorks 3.0 создала некоммерческую версию системы. Эта полнофункциональная система, которая отличается от обычной только тем, что при старте выводит окно с текстом лицензии, в котором указано, что данная версия не может использоваться для создания коммерческих приложений, а только для ознакомления и обучения. Кроме того, в нее не включена часть пакетов, в частности те, которые поставляются на условиях сублицензирования. В остальном (и, прежде всего, в формате образа) система полностью совместима с коммерческой версией. Некоммерческая версия распространялась с виртуальными машинами для

Windows 95/NT, Linux i386 и Power Macintosh. В поставку системы входили дополнения *DLL & C Connect*, *Advanced Tools* и *Database Connect* (Oracle и Sybase). Из общедоступных смолтоковских архивов были добавлены приложения и утилиты, адаптированные под данную версию VisualWorks,

Сегодня вся смолтоковская продукция фирмы ObjectShare куплена и распространяется фирмой Cincom, которая выпустила новые версии VisualWorks: сначала версию 5i.x, а затем версию 7.x¹.

Версии 5i.x и 7.x представляют существенно измененный язык Смолток, поддерживающий пространства имен, константные объекты и ограничения доступа. Сделано все, чтобы ядро системы (как оно описано в [2, 5]) осталось неизменным. Эти версии сохранили обычную для VisualWorks организацию графического интерфейса пользователя. Современные версии VisualWorks могут работать на следующих платформах:

- Sun Solaris SPARC
- IBM AIX
- HP/UX
- Linux i386/SPARC/PPC
- SGI Irix
- Apple PowerMac
- Apple Mac OS/X
- Windows 9x/NT/2k/XP.

1.3. Smalltalk/V

Эта система начала свое существование как система *Methods*, реализовавшая язык Смолток для IBM PC в среде с текстовым оконным интерфейсом. С появлением более мощных и стандартно снабжавшихся графическими мониторами компьютеров IBM XT была создана графическая версия, получившая название Smalltalk/V. Фирма Digitalk сумела найти удачный компромисс между сохранением богатых системных смолтоковских классов, обеспечивающих уникальные свойства, и необходимостью максимально уменьшить систему, предназначенную для бедных ресурсами PC-совместимых компьютеров. В Smalltalk/V были существенно сокращены классы графики, упрощена поддержка интерфейса и множественных процессов, а также полностью убрана вся поддержка многоплатформенности. С появлением IBM AT с процессором 80286, способным адресовать до 16М памяти, была выпущена версия Smalltalk/V 286, использовавшая все преимущества нового процессора.

Сохранение в Smalltalk/V 286 без изменений внешнего протокола для неграфических классов, описанных в [4], обеспечило высокую переносимость программ из других диалектов. Пользовательский интерфейс систе-

¹ Некоммерческие варианты можно найти на сайте <http://www.cincom.com/>.

мы строился на основе парадигмы «модель — панель — диспетчер» (упрощенный вариант парадигмы «модель — вид — контроллер» из Smalltalk-80) и позволял использовать широко распространенную двухкнопочную мышь вместо обязательной для Smalltalk-80 трехкнопочной, и даже работать без мыши. Были добавлены новые средства, позволяющие напрямую обращаться к портам процессоров Intel 80x86, вызывать программные прерывания и создавать внешние выполнимые модули, реализующие новые примитивы. Smalltalk/V продавался по невысоким ценам и в 90-ые годы был одной из самых широко распространенных смолтоковских систем.

На распространение MS Windows фирма Digitalk ответила выпуском системы Smalltalk/V for Windows. В ней была существенно изменена организация пользовательского интерфейса: он стал максимально приближенным к интерфейсу, используемому в среде Windows. Система получила удобные средства доступа к Windows API и возможность реализовывать новые примитивы с помощью DLL. Система Smalltalk/V for Windows быстро развивалась и после выхода второй версии долго считалась лучшей средой разработки приложений для Windows.

Были выпущены версии Smalltalk/V for OS/2 и Smalltalk/V for Mac. Версия Smalltalk/V for OS/2 является настоящим OS/2-приложением, использующим все преимущества этой быстрой и надежной операционной системы, но благодаря концептуальной близости OS/2 Presentation Manager и MS Windows библиотека классов этой версии практически совпадает с библиотекой классов версии для Windows, что обеспечивает высокую переносимость приложений.

Широкое распространение *Smalltalk/V for Windows* и *OS/2* как в корпоративной, так и в университетской среде обеспечило создание большого числа вспомогательных приложений, разнообразных инструментов и дополнений к базовой библиотеке классов. На основе Smalltalk/V for Windows была создана и система *Smalltalk Express*.

1.4. Visual Smalltalk

Система Visual Smalltalk и *Visual Smalltalk Enterprise* представляют собой результат дальнейшего развития Smalltalk/V. Это 32-разрядные системы, способные работать под управлением OS/2 и всех платформ Win32.

С точки зрения программиста, важнейшей особенностью этих систем является великолепно сделанная поддержка визуального программирования, когда в визуальном редакторе *Workbench* можно строить не только пользовательский интерфейс приложения, но его структуры данных и логику работы [22]. Система построена на основе технологии сборки приложе-

ния из визуальных и невизуальных компонентов, графически соединяемых между собой в визуальном редакторе приложения. Система обеспечивает взаимодействие с другими программами как с помощью COM/DCOM, так и SOM/DSOM.

Система *Visual Smalltalk Enterprise* отличается от базовой системы Visual Smalltalk тем, что содержит средства взаимодействия с базами данных, поддержку протокола TCP/IP и средства групповой работы над проектом. Дополнительно в систему могут устанавливаться средства поддержки коммуникационных протоколов мэйнфреймов.

В систему *Visual Smalltalk Enterprise 2000* включено средство визуального программирования *WindowBuilder Pro*, перенесенное в 32-разрядную среду из *Smalltalk Express*. Это позволяет легко переносить в *Visual Smalltalk* приложения *Smalltalk Express*, а программист при создании новых программ может выбирать между Workbench и WindowBuilder.

1.5. IBM VisualAge for Smalltalk

IBM VisualAge for Smalltalk [21] — мощная профессиональная система разработки прикладных программ, активно поддерживающая новейшие технологии проектирования и создания приложений. Сегодня из всей серии продуктов *VisualAge* это, пожалуй, наиболее развитая система. В ее основе лежит язык IBM Smalltalk [6], на котором написана большая часть систем разработки *VisualAge*. Он отличается от языка Smalltalk-80 [5] наличием экземплярных переменных класса (см. с. 44). Достоинством системы является встроенная платформонезависимая разделяемая библиотека объектов *Code Library* с механизмами управления версиями и поддержкой групповой работы. В результате, понятия частных классов и частных сообщений поддерживаются не как соглашения между программистами, а как обязательные ограничения системы. Отметим, что использование управления версиями необходимо изучить еще до начала работы с данной системой.

Библиотека классов в *VisualAge* в части, не связанной с графикой и пользовательским интерфейсом, является сильно расширенной библиотекой системы Smalltalk-80. Вместе со стандартными универсальными классами во всех категориях присутствуют специализированные классы, оптимизированные под специальные категории задач и/или под различные критерии производительности. Кроме того, библиотека содержит многочисленные классы, поддерживающие компонентную модель, доступ к БД и сетевые протоколы.

Библиотека классов графики и графического интерфейса пользователя сильно отличается от соответствующих классов систем Smalltalk-80

и VisualWorks, но, как и они, независима от платформы. Она основана на концепциях, структурах данных и функциональных вызовах XWindow System и OSF/Motif, являясь тщательно сделанной объектной «оберткой» над их стандартными библиотеками. Система позволяет создавать приложения и без графического интерфейса — консольные или демоны².

VisualAge поддерживает широкий спектр средств взаимодействия процессов, включающий сигналы, семафоры и именованные каналы в стиле Unix/POSIX, сетевые средства взаимодействия, DDE, распределенные объектные технологии, причем поддерживается как SOM/DSOM (реализация CORBA от IBM), так и менее функциональная (но более распространенная) COM/DCOM. Кроме того, в *VisualAge Web Connection* встроена поддержка интерфейсов CGI, NS API, IBM API и ISAPI, а также средств для взаимодействия с сервлетами и апплетами языка Java.

С первых версий эта система поддерживает компонентное (сборочное) программирование — способ составления программ их сборкой в визуальном редакторе из визуальных и невидимых компонентов с графическим заданием связей между атрибутами, событиями и операциями.

Сегодня *IBM VisualAge for Smalltalk* — полнофункциональная среда разработки объектно-ориентированных приложений, включающая в себя средства моделирования предметной области, поддерживающие стандартный язык описания объектной модели UML, а значит, и прямую генерацию кода по модели, выделение транзакций из модели и обратное проектирование (реинжиниринг) модели по коду. Собственный объектный брокер *Object Extender* обеспечивает отображение объектных моделей на схемы реляционных баз данных, прямой доступ к базам данных IBM DB2 и Oracle, а также доступ к любым базам, поддерживающим хотя бы один из интерфейсов, — ODBC или JDBC. Кроме того, для управления *Code Library* появились ENVY-подобные и ENVY-совместимые браузеры, упростившие управление версиями.

VisualAge работает на следующих программно-аппаратных платформах:

- IBM AIX и OS/2
- Sun Solaris SPARC
- HP/UX
- Windows 9x/NT/2k/XP

Для системы *VisualAge for Smalltalk* поставляется множество расширений, обеспечивающих эффективную интеграцию приложений как в Internet, так и в корпоративную среду, включая поддержку IBM SNA. Поддерживается непосредственное взаимодействие с приложениями для AS/400, S/390 и другими (новыми и унаследованными) платформами IBM.

² Сервисы в терминологии Windows NT.

1.6. Dolphin Smalltalk

Еще одной попыткой реализовать Смолток для Win32 является система *Dolphin Smalltalk* фирмы ObjectArt, которая по своей структуре, по уровню развития среды и инструментов примерно соответствует системе Visual Smalltalk. Она содержит средства построения интерфейса пользователя, поддержку TCP/IP, ODBC, OLE и DirectX. Для этой системы уже существуют две объектные базы данных. Пользовательский интерфейс системы поддерживает все особенности и нововведения операционной системы Windows. Существуют средства формирования автономных приложений и web-апплетов, для функционирования которых нужен специальный plug-in для браузера. Используемая при этом технология аналогична подходу, реализуемому языком *Java*, но без свойственной последнему безопасности.

Система *Dolphin Smalltalk* является переносимой Win32-реализацией, но по языку и библиотеке классов она совместима с [5], [2]. К интересным особенностям среды разработки относится иерархическая организация категорий и возможность относить один метод к нескольким категориям. В системе это используется для того, чтобы разнести методы стандартных классов как по категориям, указанным для них в [5], так и по протоколам, введенным в стандарте [2].

Отметим, что сегодня система *Dolphin Smalltalk* является одной из самых дешевых смолтоковских систем. Ее версия (с несколько ограниченными возможностями) доступна через Internet бесплатно.

2. Некоммерческие реализации

С одной реализацией языка Смолток, ставшей уже некоммерческой — *Smalltalk Express*, вы познакомились достаточно подробно, работая над материалом учебника. Как уже упоминалось, сегодня существуют специально созданные мощные некоммерческие реализации и некоммерческие версии самых современных и мощных Смолток-систем, что можно считать признаком популярности языка. Фактически это означает следующее:

- язык становится доступным для той широкой аудитории, включая научные и образовательные учреждения, которая не может широко использовать коммерческие версии;
- реализации языка, распространяемые вместе с исходными текстами по лицензии GNU или лицензиями, подобными ей, позволяют широко экспериментировать с языком, виртуальной машиной, библиотеками, инструментами разработки и расширениями, что технически затруднено или прямо запрещено лицензиями коммерческих реализаций.

2.1. GNU Smalltalk

Основная задача при разработке системы *GNU Smalltalk* состояла в том, чтобы создать систему, полностью реализующую язык (включая байткод), описанный в [4]. Уже в версии 1.0 задача была успешно решена. В *GNU Smalltalk* полностью реализована библиотека классов (с некоторыми дополнениями), описанная в [4]. Созданная система является консольным приложением, использующим вместо графического интерфейса пользователя цикл «read – eval – print».

Более совершенная система *GNU Smalltalk 1.1* также не имеет графического интерфейса, но для нее были созданы дополнения, содержащие графический интерфейс пользователя для OSF/Motif. Средств разработки программ в консольной версии нет, а графическая версия содержит только их минимальный набор (стандартные браузеры и инспекторы). Потому, в отличие от большинства программ GNU, *GNU Smalltalk* сегодня не может конкурировать с другими реализациями языка Смолток, как коммерчески, так и бесплатно распространяемыми.

2.2. Squeak

Еще одна общедоступная реализация Смолтока создана Дэном Ингаллом (Dan Ingalls), Тэдом Кэлером (Ted Kaehler), Джоном Мелони (John Maloney), Скоттом Валласом (Scott Wallace) и Аланом Кеем (Alan Kay) [8] во время их работы в компании Apple Computer, Inc., которой и принадлежат права на *Squeak*. Лицензия на эту систему, предоставляемая Apple Computer, накладывает минимальные ограничения на использование и изменение системы и по своей сути сравнима с лицензиями GNU и BSD.

Squeak — открытая переносимая реализация Smalltalk-80 с виртуальной машиной, полностью написанной на самом Смолтоке, что облегчает процесс отладки и изменений как программ пользователя, так и самой виртуальной машины и системных библиотек Смолтока. Восходящая к оригинальному Smalltalk-80, эта система использует для построения интерфейса как классическую триаду MVC [11], так и новую систему визуального построения приложений *Morphic*, основанную на идеях, развитых в незавершенном проекте SELF фирмы Sun и Стэнфордского университета (см., например, [19], [20]).

Эффективность *Squeak* обеспечивает конвертер “Смолток → ANSI C”, который позволяет получить систему, сравнимую по производительности с коммерческими реализациями Смолтока. Особенностью этой системы является чрезвычайно компактный формат объектов и усовершенствованные алгоритмы сборки мусора, позволяющие, с одной стороны, работать на

платформах с малым количеством памяти, а с другой стороны, минимизирующие задержки на сборку мусора, что позволяет использовать *Squeak* даже в некоторых задачах реального времени.

Squeak создавался для компьютеров семейства Apple Macintosh, и поэтому, в отличие от исходного Smalltalk-80, работает с цветной графикой (в том числе трехмерной), поддерживает звуковые устройства, включая синтез речи, имеет ряд серьезных усовершенствований в утилитах и инструментах программирования. Ядро этой системы, возможно, является самым компактным из всех 32-разрядных реализаций Смолтока и при адаптации объектной библиотеки может работать в 1 М памяти. Кроме Mac'ов, *Squeak* реализован на всех основных разновидностях Unix, OS/2, BeOS, Windows 9x/NT и компьютерах Acorn, Amiga и различных PDA (Palm Pilot, Zaurus и другие) — всего более 30 программно-аппаратных платформ.

Именно *Squeak* в последнее время становится основной платформой для экспериментирования с языком Смолток.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- Apple
 - Lisa, 308
 - Mac, 308, 316
- CORBA, 313
- Digitalk, 240, 308, 310
- Dolphin Smalltalk, 314
- ENVY, 313
- Fabrik, 308
- GDI, 160
- GNU Smalltalk, 315
- IBM Smalltalk, 22, 312
- Methods, 310
- Morphic, 315
- ObjectArt, 314
- ObjectShare, 308, 309
- ObjectWorks, 308
- ParcPlace, 308
- SDI-интерфейс, 55
- SELF, 18, 315
- Smalltalk-80, 55, 307
- Smalltalk/V, 5, 240, 310, 311, 322
- Squeak, 315
- Visual Smalltalk, 5, 311, 312, 314
- Visual Smalltalk Enterprise, 311
- VisualAge for Smalltalk, 5, 36, 89, 94, 308, 313
- VisualWorks, 5, 36, 47, 89, 94, 231, 237, 308-310, 312
- Xerox, 307
- абстрагирование, 17
- байткод, 61
- блок, 42
 - без параметров, 43
 - с оператором \wedge , 42, 47
 - с параметрами, 42
- виджет, *см.* панель
- виртуальная машина, 80, 307
- владелец панели, 248
- выполнение выражения, 22, 60
- выражение, 22
- глиф, 176, 178
- графический сегмент, 182
- добавление
 - класса, 66
 - метода, 69
- журнал системы, 81, 82
 - сжатие, 82
- иерархия
 - классов, 16, 20, 23, 109
 - метаклассов, 23
- имя
 - категории, 231
 - класса, 15, 230
 - метода, 29, 229
 - панели, 249
 - переменной, 25, 43, 229
 - пула, 28
- инкапсуляция, 14, 17
- инструмент **WBP**, 266, 294, 312
 - Scrapbook**, 287
 - Set Framing Parameters**, 281
 - Tab Order Editor**, 284

- выбор виджета, 271
- выбор виджетов, 280
- запуск, 267
- изменение размера, 268
- изменение свойств, 273
- метод `createViews`, 292
- метод `open`, 292
- морфинг виджетов, 291
- палитра виджетов, 269
- панель свойств, 268
- перемещение виджета, 271
- редактор меню, 275
- создание окна, 268
- составные панели, 284
- сохранение окна, 272
- тестирование окна, 274
- удаление виджета, 270
- установка виджета, 270
- интерфейс, 13
 - класса, 25
 - объекта, 187
 - экземпляра, 26
- кисть, 181
- класс, 15, 22
 - `ApplicationWindow`, 242
 - `Array`, 35, 91, 120, 129
 - `Association`, 26
 - `Association class`, 28
 - `Bag`, 120, 135, 138
 - `Behavior`, 24, 32
 - `Bitmap`, 161, 168, 170
 - `Boolean`, 48, 99
 - `Button`, 56
 - `ByteArray`, 120
 - `Character`, 33, 112, 113
 - `Class`, 23–25
 - `ClipboardManager`, 85
 - `Collection`, 37, 103, 119, 121, 122, 124
 - `Commander`, 174, 183
 - `Compiler`, 65
 - `Context`, 42, 102
 - `ControlPane`, 56
 - `CounterDemo`, 266
 - `CursorManager`, 85, 86, 183, 184
 - `Date`, 114
 - `DialogBox`, 245
 - `Dictionary`, 121, 135
 - `Directory`, 85, 144, 145
 - `EntryField`, 56
 - `False`, 48, 99
 - `File`, 102, 144, 146
 - `FileStream`, 140, 144
 - `FinancialBook`, 198, 294
 - `FixedSizeCollection`, 35, 120, 126, 129
 - `Float`, 34, 39, 116
 - `Font`, 178, 179
 - `Fraction`, 34, 39, 116
 - `GraphBox`, 56
 - `GraphicsMedium`, 161, 168–171, 177
 - `GraphicsTool`, 161, 174–177
 - `GraphPane`, 171
 - `GroupPane`, 56
 - `IdentityDictionary`, 121, 138
 - `IdentitySet`, 238
 - `IndexedCollection`, 120, 126, 127
 - `Integer`, 34, 39, 100, 106, 116, 118
 - `Interval`, 121, 129
 - `LanguageTranslator`, 259
 - `LargeNegativeInteger`, 117
 - `LargePositiveInteger`, 117
 - `ListBox`, 56
 - `Magnitude`, 28, 37, 112
 - `Menu`, 256
 - `MessageBox`, 61, 245
 - `MetaClass`, 23–25
 - `MetaClass class`, 24
 - `NotificationManager`, 85
 - `Number`, 34, 39, 102, 116–118
 - `NumberArray`, 205
 - `NumericalMatrix`, 217
 - `Object`, 20, 23, 25, 37, 87–89, 91, 92, 95–97
 - `Object class`, 23, 25
 - `ObjectFiler`, 147

- OrderedCollection, 126, 131, 150, 157
- Pen, 85, 174, 179, 180
- PhoneBook, 251
- Point, 161, 162
- Printer, 168, 173
- Process, 150–152
- ProcessScheduler, 150, 153
- Prompter, 61
- RadioButton, 56
- Random, 147
- ReadStream, 141
- RecordingPen, 174, 181–183
- Rectangle, 161, 163
- Screen, 85, 168, 171
- ScrollBar, 56
- Semaphore, 85, 150, 151, 155
- Set, 120, 135, 238
- SimpleSharedQueue, 158, 159
- SmallInteger, 117
- SortedCollection, 121, 133
- StaticPane, 56
- StoredPicture, 168, 172, 181
- String, 33, 120, 130, 170
- SubPane, 55, 240
- Symbol, 34–35, 130
- SystemDictionary, 84, 121, 138
- TextPane, 176
- TextTool, 174, 176–180
- TextWindow, 85, 176
- Time, 45, 115
- TopPane, 240
- Triangle, 204
- True, 48, 99
- UndefinedObject, 48
- ViewManager, 240, 242–243
- WBPFInancialBook, 294
- Window, 37, 55, 161, 168, 171, 240
- WindowDialog, 244
- WordIndex, 191
- WriteStream, 141, 143
 - абстрактный, 37
 - базовый, 20
 - добавление, 66
 - удаление, 67
- ключевое слово, 35, 39
- Кобол, 116
- константы
 - символьные, 33
 - числовые, 34
- контрольная точка, 75, 79
- меню, 57
- метакласс, 23, 24
- метафайл, 172
- метод, 13, 29
 - доступа к переменной, 201
 - по имени, 231
 - класса, 29
 - примитивный, 52
 - рекурсивный, 106
 - частный, 200
 - экземпляра, 29
- механизм
 - зависимости объектов, 97
 - связи между панелями, 249
- модель
 - предметной области, 240
 - приложения, 242
- модульность, 17
- набор, 119
- наследование, 15, 18
 - одиночное, 20
- обработчик события, 249
- образ системы, 81
 - сохранение, 81
- объект, 12, 13, 19
 - возвращаемый, 14
 - глобальный, 25
 - литеральный, 33, 92
 - получатель, 14
- ограничение доступа, 17
- окно, 55
 - Debugger, 62, 194
 - Walkback, 62, 194
 - Debugger, 74, 77

- Transcript, 55
- Transcript, 46, 47
- Walkback, 74, 75
- иерархии классов, 56, 62, 63
- инспектора, 62, 71
- модальное, 61
- просмотра диска, 63
- просмотра класса, 62, 70
- просмотра методов, 62, 72
- просмотра сообщений, 62, 73
- рабочее (Workspace), 55
- оператор
 - возврата значения, 30
 - присваивания, 30
- описание переменной, 45
- очередь, 157
- ошибка
 - библиотеки классов, 74
 - времени выполнения, 74
 - времени компиляции, 61
 - проектирования, 74
 - синтаксическая, 74
- память растра, 169
- панель, 55, 241
 - текстовая, 59
- переменная, 15, 43
 - блока, 43, 45, 46
 - временная, 45, 46
 - глобальная, 43
 - именованная, 44
 - индексированная, 44, 91
 - класса, 25, 43, 44
 - Dependents, 87, 96
 - RecursionInError, 87
 - RecursiveSet, 87
 - класса, экземплярная, 43, 44
 - локальная, 28, 44
 - метода, 43, 45
 - общая, 28, 44
 - CharacterConstants, 86, 141
 - Clipboard, 85
 - ColorConstants, 86
 - CurrentProcess, 85, 153
 - Cursor, 85, 183
 - CursorConstants, 86
 - Disk, 85, 145
 - Display, 85, 168, 171
 - KeyboardSemaphore, 85, 154
 - Notifier, 68, 85
 - PendingEvents, 155
 - Processor, 85, 150–153, 155
 - Smalltalk, 84, 138
 - Sources, 85
 - SysFont, 85
 - Terminal, 85
 - Transcript, 46, 47, 85
 - Turtle, 171
 - UserInterfaceProcess, 154
 - WinConstants, 86
 - WinEvents, 241
 - WinEventsExtra, 241
 - экземпляра, 28, 43
 - пиксел, 161
 - подкласс, 15
 - поиск метода, 31, 49–52
 - полиморфизм, 18, 39, 52
 - прерывание
 - Смолтока, 156
 - приоритет
 - процесса, 152
 - сообщений, 40
 - протокол
 - класса, 25
 - экземпляра, 26
 - процесс, 150
 - активный, 150
 - блокированный, 153
 - готовый к выполнению, 153
 - мертвый, 153
 - новый, 151
 - пассивный, 152
 - синхронизация, 151
 - псевдопеременная, 43, 48
 - false, 48
 - nil, 36, 37, 48, 89
 - self, 31, 45, 49
 - super, 49, 52

- true, 48
- пул, 28, 43
 - CharacterConstants, 86, 141
 - ColorConstants, 86, 174–176
 - CursorConstants, 86, 184
 - FileConstants, 86
 - WinConstants, 86, 174, 184
- растр, 168
 - цветной, 169
- рекурсия
 - объектно-ориентированная, 108
- селектор, 29, 73
- семафор, 155
- сжатие
 - журнала системы, 82
 - исходные тексты, 82
- система координат, 161
- словарь
 - пула, 43
 - системы, 25, 84, 121, 138
- Смолток, 18
- событие
 - операционной системы, 246
 - связь с методом, 249
 - смолтоковское, 182, 243, 246
 - панели, 246
- сообщение, 13, 14, 37
 - арифметическое, 38
 - бинарное, 40
 - каскадное, 41
 - ключевое, 39
 - унарное, 39
- среда графики, 161
- среда, управляемая событиями, 240
- суперкласс, 15
- тело метода, 29, 30
- триада MVC, 239
- удаление
 - класса, 67
 - класса окон, 68
 - метода, 70
 - экземпляра, 68
- форма курсора, 183
- функция API, 160
 - GDI, 160
 - BitBlt, 175
- черепашья графика, 179
- числа
 - рациональные, 116
 - с плавающей запятой, 116
 - с фиксированной запятой, 116
 - целые, 116–118
- шаблон сообщения, 29, 45
- экземпляр класса, 15

ЛИТЕРАТУРА

1. *Budd T.* A Little Smalltalk. — Reading, MA: Addison-Wesley, 1987.
2. Draft American National Standard for Information Systems — Programming Languages — Smalltalk. — Revision 1.9. — NCITS, 1997.
3. *Goldberg A.* Smalltalk-80: The Interactive Programming Environment. — Reading, MA: Addison-Wesley, 1984.
4. *Goldberg A., Robson D.* Smalltalk-80: The Language and its Implementation. — Reading, MA: Addison-Wesley, 1983.
5. *Goldberg A., Robson D.* Smalltalk-80: The Language. — Reading, MA: Addison-Wesley, 1989.
6. IBM Smalltalk User's Guide. — IBM, 1998.
7. *Ingals D.* Design Principles Behind Smalltalk // Byte Magazine, v. 6, № 8, 1981.
8. *Ingals D., Kaehler T., Maloney J., Wallace S., Kay A.* Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself // Proceedings of OOPSLA'97. — ACM, 1997.
9. *Ingals D., Wallis S., Yu-Ying Chow, Ludolph F., Doile K.* Fabrik: A Visual Programming Environment // Proceedings of OOPSLA'88. — ACM, 1988.
10. *Kay A.* The Early History of Smalltalk // ACM SIGPLAN Notices, v. 28, № 3, 1993.
11. *Krasner G., Pope S.* A Description of the Model-View Controller User Interface Paradigm in Smalltalk-80 System. — Mountain View, CA: ParcPlace Systems Inc., 1988.
12. *Lalonde W., Pugh J.* Smalltalk/V: Practice and Experience. — Prentice Hall, 1994.
13. *Lewis S.* The Art and Science of Smalltalk. — Prentice Hall, 1995.
14. *Linderman M.* Developing Visual Programming Applications Using Smalltalk. — New York: SIGS Books, 1996.
15. *McClure S.* Object Tools: Smalltalk Market Accelerates. — IDC Inc., 1995.
16. *McClure S.* Smalltalk Strengths Stand Out. — IDC Inc., 1997.
17. *Shafer D.* WindowBuilderPro for *Smalltalk Express*. — <http://omega.pef.czu.cz/pef/ki/oo/st/>.
18. Smalltalk-80: Bits of History, Words of Advice / Ed. G. Krasner. — Reading, MA: Addison-Wesley, 1983.
19. *Ungar D., Smith R.B.* SELF: The power of simplicity // Lisp and Symbolic Computation, v. 4, №3, 1991.
20. *Ungar D., Smith R.B.* Programming as experience: The inspiration for SELF // Proceedings of ECOOP'95 Conference. — Aarhus, 1995.

21. VisualAge for Smalltalk User's Guide. — IBM, 1997.
22. Visual Smalltalk Enterprise User's Guide. — ObjectShare, 1997.
23. VisualWorks Application Developer's Guide. — ObjectShare, 1998.
24. *Бадд Т.* Объектно-ориентированное программирование в действии. — СПб.: Питер, 1997.
25. *Безруков Д.И., Голосов А.О.* Система Smalltalk-80 и объектно-ориентированное программирование // Искусственный интеллект: Кн. 3. Программные и аппаратные средства: Справочник / Под ред. В.Н. Захарова, В.Ф. Хорошевского. — М.: Радио и связь, 1990.
26. *Буч Г.* Объектно-ориентированное проектирование с примерами применения. — М.: Конкорд, 1992.
27. *Выгодский М.Я.* Справочник по элементарной математике. — М.: Наука, 1964.
28. *Иванов А.Г.* Министерство обороны США выбирает Смолток // PC Week (RE), № 26, 1998.
29. *Иванов А.Г., Карпова А.В., Кремер Ю.Е., Семик В.П.* Объектно-ориентированная среда программирования // Системы и средства информатики. Вып. 3 — М.: Наука, 1993.
30. *Иванов А.Г., Кремер Ю.Е.* Язык Smalltalk: концепция объектно-ориентированного программирования // Компьютер Пресс, № 4, 1992.
31. *Кей А.* Программное обеспечение ЭВМ // В мире науки (Scientific American: Издание на русском языке), № 11, 1984.
32. *Кнут Д.Э.* Искусство программирования для ЭВМ. Т. 2. Получисленные алгоритмы. — М.: Мир, 1977.
33. *Пейперт С.* Переворот в сознании: Дети, компьютеры и плодотворные идеи. — М.: Педагогика, 1989.
34. *Рэддинг А.* С++ и Smalltalk: Чей лоб крепче? // ComputerWorld Россия, 5 декабря, № 14, 1995.
35. Смолток. Объектно-ориентированная система программирования. Руководство пользователя, часть 1, 2, 3. — М.: Ин-т проблем информатики РАН, 1995.
36. *Фролов А.В., Фролов Г.В.* Графический интерфейс GDI в MS Windows. — М.: ДИАЛОГ-МИФИ (Библиотечка системного программиста; т. 14), 1994.
37. *Фути К., Судзуки Н.* Языки программирования и схемотехника СБИС: пер. с япон. — М.: Мир, 1988.

ОГЛАВЛЕНИЕ

Предисловие	3
ЧАСТЬ I. ОБЩИЙ ОБЗОР	
Глава 1. Основы ООМП	12
1.1. Методология ООП	12
1.2. Смолтоковская реализация ООМП	20
Глава 2. Синтаксис языка Смолток	33
2.1. Определение объекта	33
2.2. Сообщения	37
2.3. Блоки	42
2.4. Переменные в языке Смолток	43
2.5. Псевдопеременные	48
2.6. Методы и примитивные методы	52
Глава 3. Среда <i>Smalltalk Express</i>	54
3.1. Основные операции с окнами	55
3.2. Специальные окна системы	62
3.3. Окна, предназначенные для отладки	73
3.4. Управление системой в целом	80
ЧАСТЬ II. БИБЛИОТЕКА КЛАССОВ	
Глава 4. Протокол класса Object	87
4.1. Проверка функциональности объекта	88
4.2. Равенство объектов	88
4.3. Индексированные переменные	91
4.4. Печать и сохранение объектов	92
4.5. Обработка ошибочных ситуаций	92
4.6. Обработка сообщений	95
4.7. Механизм зависимости	96
Глава 5. Управляющие структуры	99
5.1. Условные выражения	99
5.2. Итерации	102
5.3. Рекурсия	106
5.4. Задания для самостоятельной работы	111
Глава 6. Величины	112
6.1. Класс <i>Magnitude</i>	112
6.2. Класс <i>Character</i>	112

6.3. Дата и время	114
6.4. Числа: большие, малые и всякие	116
Глава 7. Наборы на любой вкус	119
7.1. Протокол класса <code>Collection</code>	119
7.2. Индексированные наборы	126
7.3. Неиндексированные наборы	135
Глава 8. Потоки и файлы	140
8.1. Протокол класса <code>Stream</code>	141
8.2. Особенности класса <code>FileStream</code>	144
8.3. Потоки генерируемых элементов	147
Глава 9. Независимые процессы	150
9.1. Процессы и управление ими	151
9.2. Семафоры	155
Глава 10. Графика в <i>Smalltalk Express</i>	160
10.1. Класс <code>Point</code>	162
10.2. Класс <code>Rectangle</code>	163
10.3. Графическая среда	168
10.4. Графические инструменты	174
10.5. Работа с курсорами	183
 ЧАСТЬ III. ПОСТРОЕНИЕ НОВЫХ КЛАССОВ	
Глава 11. Основы строительства	185
11.1. Принципы построения нового класса	185
11.2. Многократное использование кода	189
11.3. Использование отладчика	191
Глава 12. Домашняя бухгалтерия	198
12.1. Цель и определение класса	198
12.2. Создание нового экземпляра	199
12.3. Методы доступа к информации	200
12.4. Методы ввода информации	202
12.5. Задания для самостоятельной работы	203
Глава 13. Классы <code>Triangle</code> и <code>NumberArray</code>	204
13.1. Постановка задачи	204
13.2. Поведение объектов <code>NumberArray</code>	205
13.3. Поведение объектов <code>Triangle</code>	209
13.4. Задания для самостоятельной работы	213
Глава 14. Матрицы	215
14.1. Основные математические определения	215
14.2. Постановка задачи	217
14.3. Протоколы класса <code>NumericalMatrix</code>	218
14.4. Задания для самостоятельной работы	226

Глава 15. Особенности создания кода	228
15.1. Что такое смолтоковский стиль	228
15.2. Соглашение об именах	229
15.3. Доступ к переменным экземпляра	231
15.4. О структурировании методов	233
15.5. Использование комментариев	234
15.6. На что еще стоит обратить внимание	235
ЧАСТЬ IV. ПОСТРОЕНИЕ ПРИЛОЖЕНИЙ	
Глава 16. Интерфейс пользователя	239
16.1. Класс Window	240
16.2. Класс ViewManager	242
16.3. Панели и события	245
16.4. Пример: телефонная книга	250
16.5. Пример: переводчик	258
16.6. О цикле разработки приложений	264
Глава 17. WindowBuilder Pro	266
17.1. Построение простого счетчика	266
17.2. Инструменты WBP	280
17.3. Возможности быстрого макетирования	288
17.4. Методы createViews и open	292
Глава 18. Интерфейс для FinancialBook	294
18.1. Эскиз приложения	294
18.2. Реализация приложения	297
18.3. Как организовать работу приложения	305
18.4. Задания для самостоятельной работы	306
ПРИЛОЖЕНИЯ	
Мир языка Смолток	307
1. Коммерческие реализации	307
2. Некоммерческие реализации	314
Предметный указатель	317
Литература	322

Кирютенко Юрий Александрович
Савельев Василий Александрович

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ
Язык Smalltalk

Книга издана в авторской редакции

Ответственный редактор *Н.Г. Карсева*
Технический редактор *П.С. Корсунская*
Компьютерная верстка *В.А. Савельева*

Подписано в печать

Печать офсетная. Бумага газетная.

Гарнитура «Таймс». Формат 60x84 1/16.

Усл. печ. л. 19,06. Тираж 300.

ЗАО «Издательское предприятие «Вузовская книга»
125993, Москва, А-80, ГСП-3, Волоколамское шоссе, д. 4,
МАИ, Главный административный корпус, к. 301а.
Т/ф (499) 158-02-35. E-mail: vbook@mai.ru, vbook@mail.ru