Practical Use of Generative Techniques in Software Development Projects: an Approach that Survives in Harsh Environments

Joern Bettin
Equinox Ltd. Software Architects
Level 12, Equinox House, 111 The Terrace,
PO Box 10 168
Wellington, New Zealand
+64 4 499 9450
joern.bettin@equinox.co.nz

Abstract

The forces pulling on real-life software projects can easily corrode the best attempts at introducing generative techniques. No matter how desirable generative techniques are to improve the technical quality of a software solution, unless the new techniques are introduced with proper diligence, they are not likely to become state-of-the-practice. Techniques that work in a research environment may not scale up to larger, less structured project environments. What does seem to work, in my experience, is combining the use of a UML tool with template based code generation techniques.

1. Introduction

I was first introduced to generative techniques in the early nineties, in a non-OO implementation environment. At the time we built a template based code generator called RUOM[1] (Rapid User Object Method) that took its input from a text based, non-graphical object modelling tool. Fortunately we did not have to reinvent the wheel and leveraged the power of the LANSA 4GL and CASE environment that already provided a template language. The result was a highly configurable code generation engine that allowed the user to generate a fully working implementation from a highly abstract design object model. Over the following two years I used this tool set in projects for three different customers. In all cases I was able to train users within the customer organisation to properly use the tools to produce applications, however I noticed that most organisations lacked the skills to become proficient in the configuration of the code generation engine. This is an area that requires specialist skills and a level of discipline not found in most software development organisations. I suspect that this is one major reason for the continuing lack of industrial use of generative techniques.

Since then I have worked on a number of OT projects of varying sizes, using a number of different tools, and until now have not come across any set of OO modelling and development tools that could match the level of productivity achievable with LANSA RUOM. The potential offered by OO languages can only be harnessed by the use of a highly integrated development environment that integrates abstract [meta-] modelling, flexible code generation for arbitrary target languages, traditional IDEs, and deployment tools. Recently I have used a UML modelling tool (Rose[2]) and a template based code generator (GenIt Architect[3]) to increase productivity and quality in a Java development project.

The following sections describe my experience with this combination of tools.

2. Model-driven development with today's tools

Most commercial UML modelling tools promote modelling at the same level of abstraction as the implementation. Essentially this approach interprets the UML as a graphical notation that provides a view into the implementation source code. In particular if the round-trip-engineering features of the typical commercial UML tools are used, the resulting models contain repeated instances of design patterns that have been used in the implementation. This use of the UML can lead to frustrated development teams where the UML tool is mainly used as a drawing tool for post-implementation documentation.

Our motivation to use the UML is congruent with [Harrison-00], where the UML design models are implementation language independent, and where the model can be used to generate implementation structures in potentially more than one target language. The achievable difference in the level of abstraction between UML model and implementation code directly corresponds to a gain in productivity, and depends to a large degree on the quality of the code generation techniques used to map from UML model to target language(s).

The effort of maintaining accurate UML models is more than offset by the leverage gained by automated

[1] RUOM is part of the LANSA product from LANSA Inc.

[2] Rose is a product from Rational Software Corporation

[3] GenIt Architect is a product from Codagen Technologies Corp.

generation of structural Java code, SQL DDL, and XML deployment descriptors. The abstract UML models can be used as reliable documentation by subsequent projects. Through the use of appropriate code generation techniques the models can be re-used in other projects in significantly different target implementation environments.

Today's UML modelling tools still lack critical features that were built into LANSA RUOM back in 1994 such as:

1. A 100% reliable impact analysis that is able to calculate precisely which artifacts need to be re-generated due to model changes since the last generation. This type of feature is invaluable, as it minimises the impact of a re-generation and it provides the development team with an advance plan of what items might require intensive testing after re-generation.
2. An industry-strength implementation of the object-relational mapping is required if a relational database has to be used. In particular if new applications are being developed, the architect should be able to define O-R mapping rules at a very high level of abstraction. Manually mapping each persistent property of an object model to entities in a separate data model is unacceptable and undesirable. With GenIt Architect we have completely automated the mapping so that the database structure is continuously synchronised with the structure of the object model at generation time. We still don't have the degree of automation that was achieved with LANSA RUOM, where the data content of development and test database tables automatically got migrated to the new table structures.
3. Full integration with a build environment. LANSA RUOM provided a "builder queue" that automatically scheduled all build activities identified by the impact analysis and that provided the user with precise details of any problems incurred during a build. Once the user had fixed the identified build problems, the RUOM builder queue would allow rescheduling of all remaining build activities.
4. An industry-strength approach to non-destructive re-generation that allows evolutionary prototyping. A generative environment may use generative techniques at multiple levels of abstraction, and typically software engineers may want to add hand-crafted code at each of these levels in order to implement behaviour that can't be automatically generated economically (automation would cost more than hand coding). In OO languages problems with non-destructive re-generation can largely be avoided by structuring the code base in such a way that hand-crafted code is located in classes that don't contain generated behaviour. However in the absence of a reliable impact analysis feature that precisely calculates the scope of a regeneration in advance, existing hand-crafted code could still have unexpected effects on system behaviour.

My experience with the GenIt Architect tool has been very positive. The integration with UML modelling tools is of high quality, and the template language environment allows any skilled software engineer to perform code template changes. Provided that the mapping between UML model and generated implementation artifacts is meticulously documented, it is possible to use GenIt Architect to substantially raise the level of abstraction of the UML model over the level of abstraction of the generated artifacts.

3. Summary of a project

The first project where I used GenIt Architect was a project to build a complex commercial application within the New Zealand electricity industry. The overall approach made use of modelling and meta-modelling techniques, formal specifications of component architecture standards, a commercial UML modelling tool (Rational Rose), and the GenIt Architect template-based code generator.

In this project my main objective was to achieve re-useable, implementation independent design models. One success factor was my previous insight into the domain: I knew the customer, and had performed a high-level requirements analysis before the project started.

Another success factor was the complex nature of the domain. The customer required a flexible implementation of a pricing system for electricity transmission charges. The underlying pricing methodology is likely to change at least on a yearly basis, and depends on the structure of the electricity transmission grid, which changes as soon as a new customer is connected to the grid. This meant that we had a strong incentive to design a solution that allows dynamic pricing methodology and grid changes with minimal – if any - change to the code base.

The result is a system design that relies on a careful balance of generic and generative techniques. The generative techniques have allowed us to use implementation patterns that would have been prohibitive to implement by hand, and they provide the platform for an implementation independent design model.

From design to implementation

For maximum clarity and precision we chose a compact representation of the mapping between UML design model and the Java implementation. The basic idea is straight forward: use the UML to model each *allowable* construct that can appear in the UML *design model* in an *architecture model*, and then use the UML to model all the corresponding Java implementation constructs in an *implementation model*.

In the UML each model element is only allowed to be assigned to one stereotype. We however needed *multi-dimensional stereotypes* to express the mapping between design model and implementation model with sufficient precision. Tagged values could have been used to

classify model elements along multiple dimensions, but we already used tagged values to capture additional model element properties required to fine-tune code generation, and preferred the visual UML notation of stereotypes.

In summary our modelling and meta-modelling techniques led to a highly compact and precise specification of implementation patterns. The abstract nature of the notation however raises issues with using UML tools that only have restricted meta-modelling capabilities. In other words the UML extension mechanisms of stereotypes and tagged values are insufficient to allow usage of UML as a complete architecture description language.

Mapping to a Java, XML and SQL implementation

The actual mapping of the architecture model to the implementation model makes use of patterns similar to those described by [Harrison-00] supplemented by additional patterns that reflect the project-specific application architecture.

Most classes in the architecture model correspond to a pattern of a Java interface, an abstract Java class, a concrete Java class, and a Java collection class [used for the implementation of associations], SQL DDL [for persistent classes], and corresponding XML definitions in the implementation.

4. Conclusion

Although today's tools still lack features that are required in the industrial use of generative techniques, they allow significant improvements over handcrafted OO implementations. In my opinion however, non-technological factors determine whether a project team embraces generative technology or not.

Unless these factors are taken into account, I predict that generative technologies will not be adopted by mainstream software development organisations in the foreseeable future. The appendix explores this topic.

5. Appendix

On the one hand generative technology has an extensive track record, on the other hand it is only rarely used in typical industrial software development projects. This indicates that there is a fundamental problem.

Vendors of mainstream software development tools tend to paint a simplistic picture of how software should be developed. Their target audience is the mass-market of all organisations that develop software. It is in a vendor's own interest not to acknowledge that many complex problems should best be addressed by a fine-tuned, domain-specific development environment [and methodology].

Software developers try to keep up with constantly changing implementation technologies and standards. Most software developers consciously only invest in skills that they perceive as increasing their marketability, i.e. mainstream implementation tools and technologies. Learning how to drive a domain-specific development environment falls outside their field of interest, even if it would be beneficial to their employer.

Large software service providers have a strong incentive to support the initiatives of development tool vendors and to accommodate the interests of their own software developers.

The customer hears the same assertions from all directions, and is led to believe that his/her project will take 18 months, requires a team of 50 people, and will cost X million dollars.

Any small consulting organisation that tries to use non-mainstream tools and techniques to develop software of higher quality, in less time, and with fewer people faces huge challenges. Why should the customer trust such an organisation - when at the same time several large software service providers tell him/her that the project requires a much larger effort?

Getting buy-in from a development team

The first hurdle is to convince an entire software development team that there is something to gain by introducing generative technology and domain-specific techniques. In my experience you need to start small, with a single project and a team of 10 people or less. On the team you need to have at least one mentor who knows how to use and configure the tools required. Ideally the mentor already has in-depth knowledge of the problem domain of the project.

The mentor needs a certain amount of lead-time to perform a first-cut domain analysis and then to proceed with domain design and implementation to set up a prototype end-to-end development environment. This environment is used as a proof-of-concept, to demonstrate the potential of the tools and techniques, – seeing is believing.

The next step is to involve two or three additional team members in scaling up the prototype to a level that can be used in a real project. It is advisable not to attempt automated generation of too many artifacts on the first project. The clock is ticking, and the team will quickly get impatient.

The most important objective is to ensure that the development environment is at least as productive as an environment with traditional development techniques. This will entice team members to use the new tools and techniques. In the perception of the team, an implementation technology independent design model resulting from the new approach will only be viewed as a side effect and not as a critical project artifact. The new approach has to pay for itself purely through increased productivity. This is particularly true if the performance

bonus scheme of the company is largely based on project performance.

Getting buy-in from the customer

Whether the customer needs to buy into the approach depends on the nature of the project. In the case of a fixed-price contract the customer won't have a large interest in the method used to deliver the solution. In projects where the customer pays by the hour or similar effort-dependent measures, the situation is reversed. Similarly the customer will have a large interest in the approach taken if members of the customer's organisation largely perform the project.

Buy-in from the customer can typically only be obtained if a pre-configured prototype development environment can be demonstrated in a pilot project (4-6 weeks) specifically set up for this purpose. Again, as with buy-in from a development team, the customer will judge success largely by productivity and the tangible deliverables of the pilot project. The more intangible quality aspects of the solution will not be relevant to the customer at that stage.

The future

Given the limited acceptance of generative techniques in the majority of software development organisations, I have the following questions:

1. What can be done to promote the use of generative techniques in an environment dominated by tool vendors focused on general purpose programming languages, software service providers, and the large number of "traditional" software developers?
2. Generative techniques can only be made accessible to a larger audience if there is support from off-the-shelf tools: [meta-]modelling tools, template languages, and transformation engines. What techniques can vendors of such tools use to compete against vendors of traditional development tools?
3. Is it realistic to expect that the majority of software developers can be trained in the use of generative techniques? The discipline that is required in a generative development environment seems to be foreign to most developers. Are our universities teaching the right skills and attracting the right people in the field of software engineering?
4. When confronted with a fully configured highly automated development environment, many developers are reluctant to use it. They resent the lack of implementation freedom. How can technically oriented software developers be encouraged to learn more about the problem domain instead of tinkering with implementation options?
5. I suspect that introducing highly automated domain-specific development environments into a large team environment only works incrementally, with one small team leading by example. Does anyone have practical experience in training a traditional software development team of 100 people or more in generative techniques? What are the issues? How long does it take?

References

[Harrison-00] William Harrison, Charles Barton, Mukund Raghavchari. Mapping UML Designs to Java. *Conference Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), 178-187 (October 2000).