Generating enterprise applications from models – experience and best practices

Vinay Kulkarni and Sreedhar Reddy

Tata Research Development and Design Centre, Pune, INDIA

{vinay.vkulkarni, sreedhar.reddy } @ tcs.com

Abstract: Modern business systems need to cater to rapidly evolving business requirements in an ever-shrinking window of opportunity. Modern business systems also need to keep pace with rapid advances in technology. Model-driven development approach addresses these issues by separating the technology concerns from functionality by providing a set of modeling notations for specifying different layers of a system namely user interface, application functionality and database, and a set of code generators that transform these models into platform-specific implementations. We have used this approach extensively to construct medium and large-scale enterprise applications resulting in improved productivity, better quality and platform independence. We discuss this experience and the best practices that evolved there from. Large-scale applications benefited from a centralized model repository that provided a single point of control for integration, change management and consistent generation of various code and document artifacts. They also benefited from an architecture-aware special-purpose language for specifying business logic, a set of abstractions for partitioning the models into well-defined modules organized into workspaces with a well-defined integration policy, and a model-driven testing approach for independent unit testing of client and server sides. This has resulted in improved productivity, better quality and smoother integration. Small-to-medium scale projects were predominantly Java or .Net-centric and favoured a light-weight code-centric development approach. These projects were primarily interested in the benefits of code generation that MDD provides. The ability to extract models from annotated Java or C# code which then drive code generation through customisable code generators, and the ability to weave the generated code into hand-written code benefited these projects most.

Introduction

Faced with the problem of developing large and complex applications, industrial practice uses a combination of non-formal notations and methods. Different notations are used to specify the properties of different aspects of an application and these specifications are transformed into their corresponding implementations through the steps of a development process. The development process relies heavily on manual verification to ensure the different pieces integrate into a consistent whole. This is an expensive and error-prone process demanding large teams with broad-ranging expertise in business domain, architecture and technology platforms. In this paper, we present a model-driven development approach that addresses this problem by providing a set of modeling notations for specifying different layers of a typical business application system namely user interface, application functionality and

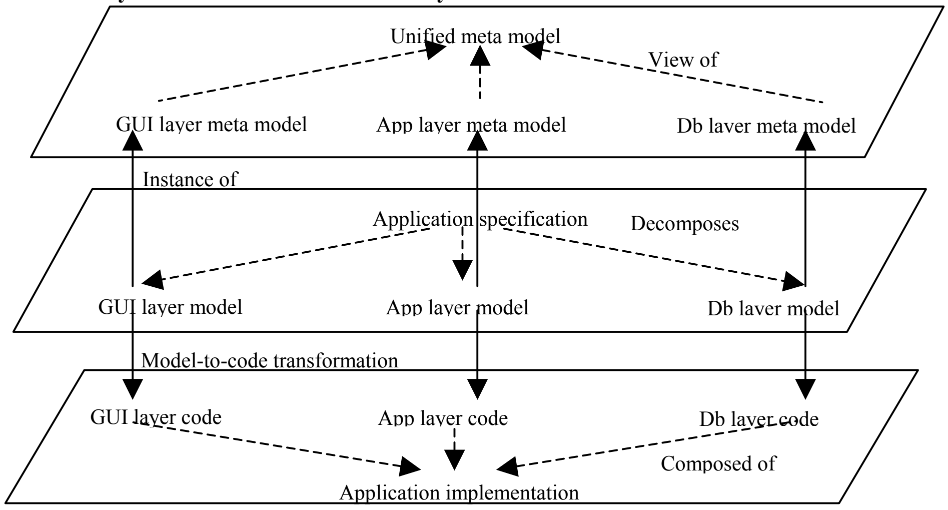

Fig. 1. Model based development approach

database; a set of code generators that transform these models into platform-specific implementations; and an abstraction for organizing application specification into work-units and an associated tool-assisted development process [4].

The development of an application starts with an abstract specification that is to be transformed into a concrete implementation on a target architecture [3]. The target architecture is usually layered with each layer representing one view of the system e.g. Graphical User Interface (GUI) layer, application logic layer and database layer. The modeling approach constructs the *Application specification* using different abstract views - *GUI layer model*, *App layer model* and *Db layer model* each defining a set of properties corresponding to the layer it models as shown in Fig. 1. Corresponding to these specifications are the three meta models - *GUI layer meta model*, *App layer meta model* and *Db layer meta model* which are views of a single *Unified meta model*. Having a single meta model allows us to specify integrity constraints to be satisfied by the instances of related model elements within and across different layers. This enables independent transformation of *GUI layer model*, *App layer model* and *DB layer model* into their corresponding implementations namely *GUI layer code*, *App layer code* and *Db layer code*. These transformations can be performed either manually or using code generators. The transformations are specified at meta model level and hence are applicable for all model instances. If each individual transformation implements the corresponding specification and its relationships with other specifications correctly then the resulting implementations will glue together giving a consistent implementation of the specification as depicted in Fig. 2. Models can be kept independent of implementation technology and the application specifications can be targeted to multiple technology platforms through model-based code generation. Construction of application specification in terms of independent models helps divide and conquer. Automated code generation results in higher productivity and uniformly high quality. Modeling helps in early detection of errors in application development cycle. Associated with every model are a set of rules and constraints that define validity of its instances. These rules and constraints could include rules for type checking and for consistency between specifications of different layers.

Experience

The model-driven development approach described above has been used to develop several large business applications, a representative set of which is summarized in the table below [2]. The column *Domain model* refers to the domain classes and not to the implementation classes.

Project	Specifications			Generated code		Technology
	Domain model (no of classes / screens)	Size (kloc)	Kind	Size (kloc)	Kind	Platforms
Straight Through Processing system	334 / 0	183	Business logic, Business rules, Queries	3271	Application layer, Database layer, Architectural glue	IBM S/390, Sun Solaris, Win NT, C++, Java, ICS, MQ Series, WebSphere, DB2
Negotiated dealing system	303 / 0	46	Business logic, Queries	627	Application layer, Database layer, Architectural glue	IBM S/390, Win NT, C++, CICS, MQ Series, COM+, DB2
Distributor management system	250 / 213	380	Business logic, Business rules, Queries, GUI	2670	Application layer, Database layer, GUI layer, Architectural glue	HP-UX, Java, JSP, Weblogic, Oracle, EJB
Insurance system	105 / 0	357	Business logic, Business rules, Queries	2700	Application layer, Database layer, Architectural glue	IBM S/390, Sun Solaris, C++, Java, CICS, DB2, CORBA

We discuss our experience in using this approach in these projects and the best practices that evolved out of that experience. Several projects had a product-family nature wherein a product-variant needed to be quickly put together and customized to meet the specific requirements of a customer. Model-driven development approach helped in quickly retargeting the application functionality on multiple technology platforms. This was achieved using a relatively unskilled workforce as the technology and architecture concerns were largely taken care of by the tools. The tool-assisted component-based development process helped in early detection of errors that would otherwise have led to late-stage integration problems. Also, all the projects reported significant improvements in productivity and quality.

Best practices

Best practices for large-scale projects

These projects typically have an average team size of 50 or more and run for about two or more years.

Prototyping phase

We have found that no two business applications have exactly the same architectural requirements and hence the same requirements on the tools that deliver into these architectures. So an upfront prototyping phase wherein a representative sample of the target application is developed and tested with a representative usage profile is critical to flesh out the architectural requirements early in the life cycle. Tools can then be customized to deliver into the validated architecture, before the project proceeds into design and implementation phases. Tool customization typically involves defining new meta-models or extending exiting meta models, defining custom model editors if any, and implementing the code generators that deliver into the chosen architecture.

The alternative of starting with an existing architecture (and the corresponding toolset) with the hope that it will suffice has proven to be a bad practice. Discovering an architectural problem when the project is in full swing leads to costly delays and wasted efforts, requiring retooling, retraining and model porting.

We have found that the time spent in initial architecture prototyping and tooling does not lead to any significant overall delays, as the project team can concurrently develop analysis models which are not impacted by the architectural requirements.

Repository-centric development

Large-scale applications by their nature have a large number of components with large development teams working on them concurrently. It is necessary to carefully plan and control the project so that all the development artifacts are consistent with each other and the requirements are implemented consistently across all the parts. A centralized model repository is found to be an invaluable aid in this effort. The repository provides the single point of control for coordinating integration, change management and consistent generation of various code and document artifacts. Recognizing this benefit many projects have even automated their document production from the repository by capturing the descriptions as annotations in the model. Generation from the model ensures that all the documents and their cross references are consistent.

Special-purpose language for business logic

We have designed a special-purpose high-level language for specifying business logic of typical business applications that are database centric, transactional and client-server in nature. Business logic specifies the computations to be performed by the application. The language is tailored for the architecture and frees the application developer from low-level implementation concerns such as memory management, pointers, resource management, etc. Another big advantage, especially for product-lines, is that it is easily retargetable to programming languages of choice such as Java, C++, C# etc. So, despite the initial hurdle of project teams having to learn a new language, all the large projects have found this language to be extremely useful, not only for its simplicity (leading to productivity and better quality), but also because it makes it possible for them to deliver their applications in different target languages.

Component-based development process

Large projects have large teams working concurrently. It is extremely important to have a well-defined process that ensures work products developed by different teams integrate smoothly while at the same time ensuring adequate separation between the development of these work products. We addressed this issue by providing two abstractions: *component* (not to be confused with a deployment component which is not necessarily the same) and *workspace*. Development work products of a project are divided into a set of components. A component has two parts – a model part and a code part. A component has an interface that exposes artifacts i.e. model elements such as classes, operations, queries, etc. that other components can use. A component has to explicitly declare dependencies on other components whose artifacts it wants to use, and it is only allowed to use the artifacts that are exposed in their interfaces. A set of constraints are defined at the meta-model level that check that the consumer-supplier relationships are correctly honored. These constrains can be arbitrarily complex going beyond the typical type consistency checks available in the coding world. For instance, a window should display data that is consistent with respect to the parameters being passed to the operations invoked from the window.

Each component has an associated workspace in which it is developed. Component workspaces synchronize themselves with a shared workspace using the check-in/check-out protocol. On each check-in/check-out the models are validated against the integration constraints.

Projects that used this process had a significant reduction in integration problems which are usually a major source of head-ache in large projects.

Model-driven testing

In a typical client-server application, GUI design is the one that undergoes most number of changes. If there were a way to separate GUI development and testing from the server side, it would tremendously reduce the change cycle-times. The testing should include not only look and feel, but other aspects such as tab logic, enabling-disabling logic, intra- and inter-field validations, window navigation, checking that right set of objects get created and passed as parameters to server methods, etc.

In our approach, since GUI is completely modeled, it is possible to automatically generate a client-only application (with appropriate stubs for server methods) from the models that can be tested for all the above mentioned properties. This has been found to greatly speed up GUI development. Test suites are also generated to unit test the server methods. A test-data generator generates optimal test data by processing the constraints captured in the object model to exercise the server methods.

The approach has resulted in a significant reduction in the number of bugs discovered during integration testing.

Synchronization of code and models

One major complaint about the model-driven development approach above was regarding the overhead associated with small changes. For instance, if an attribute needed to be added to a class, it had to be added in the model first followed by code generation to bring it into code, thus resulting in much longer cycle times than in direct coding approaches.

We have discovered that even a simple round-tripping approach that only reflects minor changes - such as changes to attributes and method signatures – back into models goes a long way towards addressing the cycle-time problem. This is because an overwhelming majority of the changes that people wanted to make directly in code were indeed such simple ones.

Best practices for small-to-medium scale projects

These projects typically have an average team size of 5 or less and run for about six months.

Code-centric development

Small projects find model-driven development a heavy weight approach to use due to the associated steep learning curve that requires a high initial investment in terms of time and effort that they can not afford. Instead, they prefer to use more traditional code-centric approaches wherein models are used primarily as documentation aids if at all. At the same time they want the benefits of code generation that MDD provides.

A majority of these projects are either Java or C# centric. Since a significant amount of code generation happens directly from class models and since class models (sans associations) can be extracted from Java (or C#), we can provide the benefits of code generation to these projects by requiring them to suitably annotate the Java (or C#) code with tags (or attributes in C#) to provide additional information on the class models. These annotated class models are then transformed to a form suitable for the consumption of model-based code generators.

Customizable code generators

Also unlike in large projects, small projects typically can not afford a separate prototyping phase to validate their architecture upfront. Instead they start with the best guess and expect to be able to change the architecture later on if required. Which means that they should have the flexibility to be able to change the code generators quickly as the need arises. They need open, extensible, easy-to-use, template-driven code generators that they can quickly customize.

Weaving of generated code into hand-written code

Since large parts of these applications are hand-written directly in the target language, it is imperative that there exists a good code weaving mechanism, as in AOP [1], that weaves the generated code into the hand-written code.

Open issues

Despite the many acknowledged benefits, the projects have also reported a few problems with the approach. In a model-driven development approach, part of the specification is in model form and part of it in code form. However debugging support is available only at the code level leading to difficulties in debugging. Also, the cycle-time required to effect a small change and verify its correctness was found to be significantly greater for the model-based approach than the traditional approach. However, the fact that a model-level change gets automatically reflected at multiple places in a consistent manner was appreciated.

Conclusions

We have presented a model-driven development approach that has been successfully used to develop and maintain several large-scale business applications. We have discussed the best practices that have emerged from our experience in these projects. We have also discussed the needs of the small-to-medium scale projects and how they can be met. Despite some short-comings the model-driven development approach has proven to be extremely beneficial in the development of large-scale business applications, especially product-lines. It has resulted in improved productivity, quality and a better handle over change management.

References

1. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Longtier and John Irwin. Aspect oriented programming. ECOOP'97 LNCS 1241, pp 220-242. Springer-Verlag. June 1997.
2. MasterCraft – Component-based Development Environment' Technical Documents, Tata Research Development and Design Center.
3. Sreenivas A, Venkatesh R and Joseph M,. Meta-modelling for Formal Software Development in Proceedings of Computing: the Australian Theory Symposium (CATS 2001), Gold Coast, Australia, 2001. pp. 1-11
4. Vinay Kulkarni and Sreedhar Reddy, UML Modeling Languages and Applications, «UML» 2004 Satellite Activities, Lisbon, Portugal, October 11-15, 2004, Revised Selected Papers. Lecture Notes in Computer Science 3297, pp 118 – 128, Springer 2005, ISBN 3-540-25081-6,