

OOPSLA '98 Metadata and Dynamic Object-Model

Pattern Mining Workshop

Michel Tilman, System Architect Unisys, (mtilman@acm.org)

Context

This paper describes some experiences in implementing a meta-driven, reflective object-oriented framework. We use the framework to build applications in administrative environments. These applications often share a common business model, and typically require a mix of database, document management and workflow functionality.

The framework uses a repository to store meta-information about end-user applications. This includes object model, object behavior, constraints, specifications of application environments (initial views on the shared model), query screens, layout definitions of overview lists and forms, authorization rules, workflow process templates and event-condition-action rules. Fully operational end-user tools consult this meta-information at run-time, and adapt themselves dynamically to the application specifications in the database. Thus we effectively separate specifications of a particular organization's business model from the generic functionality offered by the end-user tools. Rather than coding or generating code, we develop end-user applications by building increasingly complete specifications of the business model. These specifications are available for immediate execution.

Dealing with change

Users and developers are becoming increasingly aware that change is a constant factor and that most applications are never truly finished. Hence the need to make design for change a primary goal of most, if not any, architecture. This goal applies equally well to end-user applications and to development tools.

To support this development process we built a framework. Frameworks tend to go through several iterations before converging to a stable product. But even mature frameworks need to evolve, although often at other time-scales than the end-user applications and the development tools. Sound object-oriented software engineering practices often enable developers to localize the impact of changes. Yet some architectural designs or requests for change are not properly met without additional techniques.

Reflection has many uses. For one, it enables us to dynamically 'reason about' and 'manipulate' applications. But when the reflective facilities are sufficiently expressive, we get additional advantages. True reflection empowers us to intervene at the right spot, in order to achieve the right balance between minimal intervention and the required impact. For these reasons we opted for Smalltalk as the development environment for our framework.

The reflective architecture of our framework is separate from the reflective facilities of the underlying development language. Thus there is no reason why we could not have used a more static language, such as C++ or Java. Yet we are convinced that we would not have been able to assimilate and incorporate lessons learned from past experience quite as well in any of these languages and environments. In fact, the very objectives of the initial Smalltalk environment have always been a major source of inspiration for our framework: a framework that evolves, designed to build applications and tools that evolve, relying heavily on meta-information, bootstrapping and reflection.

And even if, in absolute terms, only a very small percentage of our code actually uses Smalltalk reflection, it is always reassuring to have that extra headroom, just in case it is needed.

Building a meta-repository based framework

Although we designed our framework from the start to be driven by a repository of meta-information, it has evolved considerably over the past few years. The evolution happens essentially as follows: when we add hotspots to the framework, we determine whether these hotspots are more related to generic and re-usable functionality, such as printing or entering data, or are more business-related, such as business rules and models of data and processes. For instance, at the framework level we provide hotspots to add

new editors for entering data. At the application-development level we specify which editor we want to use when designing a form layout.

Thus we reify high-level specifications from business-specific hotspots in the framework components and adapt the framework to interpret these specifications. We store these specifications in a central repository to make them dynamically available. A few of these hotspots are driven by what is typically regarded as regular data, such as the login user, but most specifications represent meta-information about the end-user applications: e.g. object model, business rules and list and form layouts.

Thus our basic architecture uses a repository containing object- and meta-level information. In an initial phase we developed end-user tools consulting the meta-information, and development tools producing the meta-information. What the user perceives as an application is actually a representation of the meta-information suitably filtered by the tools.

We wanted to keep the number of tools as small as possible, so we started to bootstrap the system. In this process we discard original hard-wired development tools in favor of tools developed in the system itself. Thus the development tools (and in fact, the end-user tools) consult and produce the meta-information. This approach has another important benefit. Just as end-users tailor the applications to their proper needs, application developers can now easily customize and personalize the development tools as well. To achieve this goal, we had to model the meta-information explicitly in the system itself.

But there are other ways of bootstrapping the system. We also wanted to keep the kernel meta-model and repository as small as possible. Thus we focused on giving application developers the means to e.g. define their own types of constraints and even enhance the basic meta-model, using the available tools. Expressing the meta-model in itself and storing it into the repository opens up the field for some of these exciting possibilities (although this is still in an exploratory phase).

Putting the reflection to work

Once more we can take a look at what makes Smalltalk work. Smalltalk reifies a large and relevant part of its meta-structure and tools, and turns this reified information into first-class objects. The net -and important- result is that object- and meta-level objects freely communicate.

In our framework we model most relevant aspects of the meta-information explicitly, such as object model, script rules, workflow process templates and application environments. We represent objects as dynamically typed structures. The variable state pattern takes care of mapping object properties onto the actual values. Meta-information can be treated as regular data and associated with object-level information. Authorization rules, for instance, use both object- and meta-level information to achieve the right degree of expressiveness.

In a similar way we model scripts explicitly. For instance, object behavior is implemented by means of scripts associated with object types. Adding thesaurus keywords to classify and query methods, or implementing collaboration contracts can be achieved through mere modeling, without the need for additional tools.

In fact, in some cases where we are investigating possible additions to the framework functionality, such as query-by-example, it has become possible to build a simplified prototype in the system itself, instead of building a hard-wired prototype.

Dealing with specifications

Using specifications has many benefits. For one, specifications narrow the gap between the user's perception of his real-world business activities and the effort required to get the tools supporting his job. Specifications are often also more concise and easier to analyze and reflect upon.

We found another useful yardstick to the measure how effective the specifications really are. In the course of the project we changed the nature of the specifications several times, particularly with regards to meta-model, application environments, stored queries and layouts.

This is roughly analogous to, say, changes in an API. The applications using the API must be modified, typically manually, one by one. In our case we write scripts to convert the specifications from one format into another. Only in a few cases did we intervene manually. For instance during our recent conversion of stored queries, about 10 queries were not converted correctly, out of a total of 800. Rather than spending some more time to get the conversion script right, we corrected the few 'bad' queries manually. In practice, converting specifications does not turn out to be harder than converting regular data in traditional database applications.

We have the additional benefit that, as of now, our repository contains only about 300 hundred scripts in constraints, methods and event-condition-action rules.

Bridging environments

As we shift more responsibilities to the meta-repository, we may lose some of the support from our regular development tools. In some cases we need to re-create some of this functionality in our development tools. For instance, Envy, despite many shortcomings, provides some useful tools to transfer and compare applications across different libraries. Since we typically use several repositories in various phases of the development process (prototyping, testing and delivery in production environment), our application developers need appropriate import / export tools. We are currently enhancing our existing import / export tools to better support this process.

Similarly, we need extra consistency maintenance tools, for instance to detect conflicts between methods stored in the repository and 'primitive' methods in the Smalltalk class library.

Testing

About a year ago we started building a testing tool, based on Kent Beck's test framework. But given our framework approach, it is easy to configure a test application in the system itself.

For instance, in this application we list the object types and properties to be tested, and add queries and layouts. With just a few scripts we can enumerate the queries. We execute each query, once for each overview list layout (the query specifies the 'where' clause, the list layout the 'select' clause). For each query execution we open the form tool on the result. Then we browse through the list, going through all available form layouts for each object.

Having set up such a generic testing application, we only need to supply the appropriate meta-information.

Performance issues

Given the dynamic nature of our framework, we must pay ample attention to performance. Detecting and optimizing typical usage patterns becomes extremely important because the end-user has so many opportunities to adapt the applications. In contrast to more hard-wired applications, however, we need to apply these optimizations only once, at the framework level. If done appropriately, most end-user applications and development tools ultimately benefit.

This is particularly relevant for the query generator component. The queries created explicitly (in the query editor and list tool) and implicitly (in the form tool) by the user are expressed in a high-level query language. There are usually many ways to translate a given query expression into a SQL statement, hence it is important to get the query generator right. We had the additional problem that our (old) database server exhibits some highly unpredictable behavior. We ended up with pre-processing the queries based on simple statistical data. In addition to this generic strategy, the query generator consults domain-specific hints. This is similar to, say, the Smalltalk virtual machine, which is optimized for typical usage patterns, but may be influenced by application-specific policies, such as the memory management policy.

This approach hints that the very dynamic nature of the architecture can be applied to solve at least some of its own weaknesses, since the tools can reflect to a large degree on the application specifications. The authorization rules, for instance, when executed single-mindedly, tend to be rather computation intensive in some of our applications. By analyzing the rules, our framework components dynamically extend the queries with authorization-related constraints to reduce the number of queries. And in several cases the tools deduce automatically that some rules need not be checked at all. Currently, we only have a couple of object types for which a 'pure' object model (i.e. without any redundant information) continues to present some problems, performance-wise, with regards to the authorization rules.

A simple real-world example

The following script is a simple example of the scripting language we may use when developing applications.

```
City select: [ :city | true ] load: #(name zipcode state)
```

In this example we retrieve all instances of the object type 'City' (hence the true 'where' clause), requesting only the name, zip code and state properties to be retrieved, i.e. we are only interested in partial information.

In the process we:

- use the Smalltalk reflective facilities to convert this expression into a high-level query expression
- convert the query expression into an SQL statement
- set up the appropriate caches
- set up a streamer and define a suitable variable-state template to represent the objects retrieved from the database in a compact form
- retrieve and cache the objects.

In this process we dynamically consult both object-model and meta-model. The total time is less than 1.3 seconds when using an Access database through ODBC, for a total of more than 2800 objects (in this example the client runs on a Pentium 233 MHz MMX with 64 MB of memory).

Given more time, we could further optimize some steps, in particular the query generation and streamer components in steps 2 and 4, and the ODBC-interface library.

Combining the dynamic and the static

Occasionally we encounter generated SQL-statements that go way out of bounds. Thus we keep refining our generation strategy. Yet it is important to note that potential performance problems result from complex queries rather than from large tables. Currently we do not have tables containing more than, say, a couple of hundred thousand records. Hence it is not always easy to predict how our approach would scale up to million-record databases, in particular with regards to queries created by the end-user in the query editor.

We could provide the option to attach optimized queries to stored queries, and to list and form layouts. In case the user modifies the layouts and queries, we would invalidate the optimized query (this can be configured in the system itself, using the business rules). In fact, there is no reason why most of these optimized queries could not be generated by the framework itself: a simple user-assisted tool could explore the most relevant alternatives and return statistical information about the results. Thus the user himself would effectively be able to optimize most of the queries.

We do not provide this hybrid strategy right now (though it would not be too hard to implement), but it is always reassuring to know we still have several tricks in store in case the going gets tough. In fact, the sky is the limit.

References

Michel Tilman and Martine Devos, A Repository-based Framework for Evolutionary Software Development (Mohamed Fayad and Ralph E. Johnson, ed.), Wiley Computer Publishing, 1998

Michel Tilman, A Repository-based Framework for Evolutionary Software Development, MetaData Pattern Mining Workshop, University of Illinois, 1998 (<http://www-cat.ncsa.uiuc.edu/~yoder/Research/metadata/UoI98MetadataWkshop.html>)