MODEL-DRIVEN DEVELOPMENT USING STANDARD TOOLS

Julián Garrido, Mª Ángeles Martos and Fernando Berzal
Dept. Computer Science and AI, ETSIIT, University of Granada, Granada 1807, Spain

Keywords: Model-driven development, application generator, reflection, persistence service, OO model, O/R mapping.

Abstract: This paper describes a model-driven software development tool suitable for the rapid development of enterprise applications. Instead of requiring new specialized development environments, our tool builds on top of a conventional programming platform so that it is suitable for the progressive adoption of model-driven development techniques within a software development organization.

1 INTRODUCTION

Due to its large potential, model-driven software development has drawn a lot of attention during the last few years (Mellor, 2003), let it be in the form of OMG's Model Driven Architecture (Frankel, 2003), Microsoft's Software Factories (Greenfield, 2004), or plain code generation, e.g. (Herrington, 2003).

Even though hype is quite common when talking about tools purported to provide "order of magnitude" benefits, more modest but important productivity and quality benefits can still be achieved using model-driven software development techniques, such as quick prototyping, improved application portability, and reduced maintenance costs.

Learning a new tool or technique actually lowers programmer productivity initially and eventual benefits are to be achieved only after the learning curve is overcome (Glass, 2003). In order to reduce the steep learning curve associated to the introduction of a new development approach, we propose a model-driven software development tool built on top of the Java programming platform.

We introduce our system architecture in the next section. Once the stage is set, we will describe the parts that comprise the input needed to generate a typical business application using our MDD tool. We will do so in the order the user would usually design them.

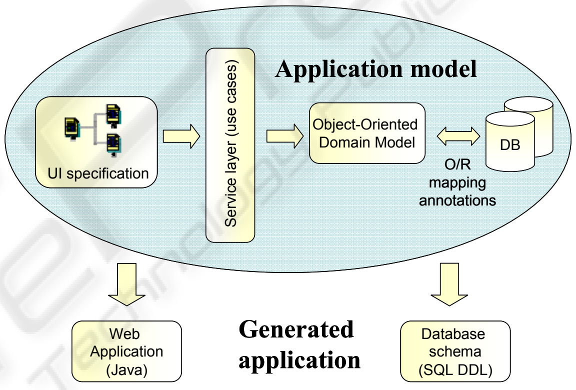

Figure 1: Overall system architecture.

2 SYSTEM ARCHITECTURE

Figure 1 shows the overall design of our MDD tool, which clearly resembles the layered architecture of many enterprise applications (Fowler, 2003).

In order to use our application generator, the user must provide an application model, shown in Figure 1 as a shaded oval. From that model, our generator will automatically create a stand-alone application. In our current prototype, the generator creates a Java web application, although it can easily be extended to create applications for a windows-based desktop environment or a mobile device, for example. As we will see, our generator can also create a suitable database schema for storing data in a relational database (for green-field projects) or can work with existing databases (in production environments where the database is already supporting other applications).

The application model needed by our MDD tool requires three components:

1. An object-oriented domain model describing our problem domain, designed using conventional object-oriented design techniques, and annotated for O/R mapping purposes.
2. A service layer providing a programmer-friendly "external" interface to our system (separate from the domain model for improving testability and simplifying the specification of the user interface).
3. An abstract model of the user interface describing the presentation layer for the system. This UI specification includes a navigation map describing how the user can move around in our application and independent descriptions of all the interaction contexts needed by the application (i.e. the different windows or web pages that will be presented to end-users).

The next three sections describe these three components in greater detail.

3 OO DOMAIN MODEL

Domain modeling is central to software design: "Through domain models, software developers are able to express rich functionality and translate it into a software implementation that truly serves the needs of its users" (Evans, 2004). Hence, an object-oriented domain model will serve as the foundation for our MDD tool.

Our tool requires a set of valid Java classes as the input describing our domain model. These classes can be developed using standard IDEs, such as Eclipse, and reused among projects once you establish a reusable asset library within your organization. Archetype patterns, as described by (Arlow, 2004), are ideal candidates for such a library.

In our MDD tool, the classes describing the domain model are enriched with metadata (Java annotations in our current prototype) so that the generated application can interoperate with a relational database, probably by using an existing O/R mapping tool such as Hibernate. Our tool includes a persistence service that provides the basic CRUD functionality needed to persist data (and a reflective O/R mapping tool so that you do not have to configure anything to get started).

These annotations, although not strictly needed by a domain model, are needed when putting domain-driven design in practice (Nilsson, 2006). In our system, they can be used to generate a database schema suitable for storing the information in our domain model and also to provide the mapping needed by our domain to work with existing databases.

Basically, annotations tell the O/R mapping tool the table within the database that will be used to store the instances belonging to a particular class. They specify which table column(s) will hold each instance variable. Finally, they describe how the relationships between classes will be represented within the relational database (using foreign keys and referential integrity constraints to represent both associations and inheritance relationships).

The annotations can also be helpful when mapping data types, since the database type system will not match that of the programming platform.

The following code snippet shows how an annotated `Order` class might look like in our system:

```
@Persistent()
public class Order{

   @Key()
   @Persistent (type=Types.INTEGER)
   int id;

   @Persistent
     (name="total",precision=2)
   Decimal price;

   @Persistent(name="order_details")
   Vector<Product> products;
}
```

The above example shows how orders are persisted. In case we are using a relational database, individual orders will be stored in the `order` table, where the order `id` will act as the primary key in a namesake table column and the order price will be stored in its `total` column. Order details will be stored in a separate `order_details` table representing the many-to-many association between orders and products.

In case our programming environment does not support generic types (any JDK prior to JDK 5), we would need a third annotation to indicate the type of the objects included in our `products` collection, i.e.

```
   @Reference(Product.class)
   @Persistent(name="order_details")
   Vector products;
```

This `@Reference` annotation is also needed to support bidirectional associations in our system.

4 SERVICE LAYER: USE CASES

Once we have defined a suitable domain model for the application we want to generate (which is arguably the hardest part of the problem, something that MDD cannot automate), the next step would consist of organizing the functionality of the desired application according to its use cases, following the usage-centered design proposed in (Constantine, 1999).

In our system, we will model each use case in a separate Java class whose methods will correspond to the steps the user must perform to complete the use case and whose instance variables will hold the objects required during the execution of the use case.

This usage-centered approach, always from the end-user's perspective, improves our system testability (Martin, 2005) and, in some sense, it is somewhat similar to how test fixtures are developed in Fit to automate testing (Mugridge, 2005). As in Fit, our approach supports iterative software development and, integrated within our complete MDD solution, it provides almost immediate feedback to changing requirements and allows for quick prototyping.

In order to illustrate how our system works in practice, we will show the implementation of a simple use case:

```
public class CustomerRegistration
              extends Task
{
 Customer client;

 public void newCustomer
             (string name)
 {
   client = new Customer();
   client.name = name;
 }
 …
 public void registerCustomer()
 {
   persistenceService.store(client);
 }
}
```

As shown above, we have also resorted to annotated classes in order to describe use cases for maintaining consistency with the approach we used to define our domain model:

- Each use case is created as a subclass of `Task`, which provides access to the persistence service needed to store data.
- The use case implementation is always performed in terms of the domain model we have previously developed, so that the coding effort is kept to a minimum in the service layer (a good design practice).

5 PRESENTATION LAYER: UI

Once we have defined our system use cases as described in the previous section, our tools help us automate the creation of the user interface for the application. Using a pure object-oriented approach, our previous work might be more than enough to generate a working user interface, as "naked objects" demonstrate (Pawson, 2002). However, we provide higher flexibility in the construction of the presentation layer for our application.

Following the same approach described above, we use an annotated set of classes to describe the user interaction contexts for our application (i.e. the set of screens the user will be presented with when using the generated application). Provided with such description, our MDD tool will be able to generate a commercial-quality complete working application.

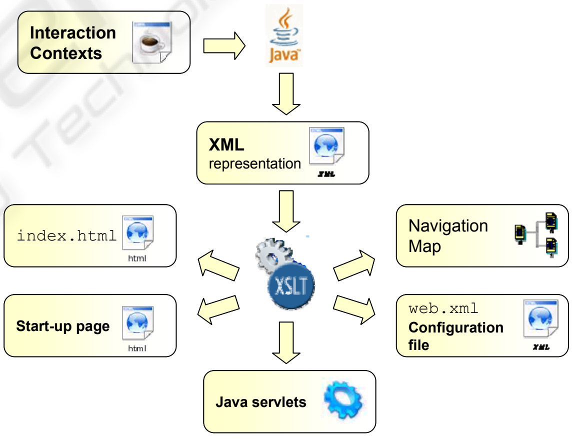

Figure 2: Provided the UI specification, our system automatically generates all the elements needed by a complete Java web application (an intermediate XML representation is used in our current system).

The code snippet below shows the description of the final interaction context the end-user would face for completing the customer registration use case from the previous section:

```
@Interface
  (task={CustomerRegistration.class})
public abstract class NewCustomer
  extends InteractionContext
{
  @Input()
  Customer client;

  @Action
    (target={CustomerAccount.class})
 public abstract void registerCustomer ();
}
```

This abstract description of the interaction context links the user interface with the system use cases. Please, note the use of the `task` property to refer to the use case implementation, the specification of the `target` property to define the application navigation map, the correspondence between the abstract actions in the user interface and the methods in the use cases, and the mapping between the data in the interface and the data consumed by the use case.

Each interaction context is annotated with metadata to describe user input (`@Input`), application output (`@Output`), user selections (`@Selection`, for data collections), individual user actions (`@Action`), and use case entry points (`@EntryPoint`). Additional annotations provide some control over the visual presentation of user interface controls, the overall organization of the application interface (e.g. task groups to provide hierarchical menus), and alternative navigation paths to be followed in the presence of errors.

6 CONCLUSIONS

Our application generator improves programmer productivity since it allows her to work at a higher level of abstraction, without having to deal with the elaborate details of the presentation and data access layers in conventional layered architectures.

The input needed by our generator is as far from implementation details as current technologies allow. By focusing on the external view of the system, its phenotype (Davis, 2003), our tool lets developers work at the analysts' abstraction level, blurring the line between requirements specification techniques and implementation technologies.

Our tool automates most, if not all, of the routine work needed to create a working application, thus avoiding the mistakes a manual process would entail and improving the overall application quality.

Moreover, the separation of concerns in our system makes applications truly portable, since they do not depend on the particular persistence mechanism employed nor on the specific technology used to develop a friendly user interface.

Even though our current prototype always works with relational databases and generates web interfaces, nothing prevents us from adding new profiles so that the same application can be targeted to different platforms (i.e. Web, windows, or mobile interfaces) and use alternative persistence mechanisms (e.g. XML databases). Because of our tool modular design and the reusable nature of specifications in MDD, application migration to new implementation technologies is almost trivial.

Our working application generator shows how model-driven development is something more than mere hype: it improves programmer productivity, helps addressing user needs, provides portable applications, and removes many sources of error that are present in hand-coded applications (hence improving quality).

REFERENCES

Arlow, J., Neustadt, I., 2004. *Enterprise patterns and MDA*, Addison-Wesley, ISBN 0-321-11230-X.

Constantine, L., Lockwood, L., 1999. *Software for use*, Addison-Wesley, ISBN 0-2101-92478-1.

Davis, A., 2003. System phenotypes, *IEEE Software* 20:4, July/August 2003.

Evans, E., 2004. *Domain-driven design: Tackling complexity in the heart of software,* Addison-Wesley, ISBN 0-321-12521-5.

Fowler, M., 2003. *Patterns of Enterprise Application Architecture*, Addison-Wesley, ISBN 0-321-12742-0.

Frankel, D., 2003. *Model Driven Architecture^{TM}: Applying MDA to enterprise computing*, Wiley Publishing, ISBN 0-471-31920-1.

Glass, R., 2003. Facts and fallacies of software Engineering, Addison-Wesley, ISBN 0-321-11742-5.

Greenfield, J., Short, K., 2004. *Software factories: Assembling applications with patterns, models, frameworks, and tools*, Wiley Publishing, ISBN 0-471-20284-3.

Herrington, J., 2003. *Code generation in action*, Manning Publications, ISBN 1-930110-97-9.

Martin, R., 2005. The test bus imperative: Architectures that support automated acceptance testing, *IEEE Software* 22:4, July/August 2005.

Mellor, S., Clark, A., Futagami, T. (guest editors), 2003. Model-driven development, *IEEE Software* 20:5, September/October 2003.

Mugridge, R., Cunningham, W., 2005. *Fit for developing software: Framework for integrated tests*, Prentice Hall, ISBN 0-321-26934-9.

Nilsson, J., 2006. *Applying Domain-Driven-Design and Patterns*, Addison-Wesley, ISBN 0-321-26820-2.

Pawson, R., 2002. Naked objects, *IEEE Software* 19:4, July / August 2002.