

Investigating User Interface Engineering in the Model Driven Architecture

Jacob W Jespersen* and Jesper Linvald

IT-University of Copenhagen, Glentevej 67, 2400 NV, Denmark

*corresponding author: jwj@itu.dk

Abstract: This paper describes a model-based approach to user interface engineering that we are currently investigating. The investigation is part of a project that utilizes OMG's Model Driven Architecture (MDA) to generate domain specific J2EE applications from a purely declarative application description. We discuss basic conditions for engineering a user interface in this context, primarily the central role of the *user interface model* and the model's relation to the concrete system. Finally, we argue that the MDA may be able to bridge some of the gaps between HCI and System Engineering under certain circumstances.

Keywords: HCI, model-based interface development, user interface specification, user interface modelling, Model Driven Architecture, program generation

1 Introduction

The Model Driven Architecture (MDA) is an approach to IT system specification that separates the specification of system functionality from the specification of that functionality on a specific technology platform (OMG, 2001). Notably, the MDA prescribes a Platform Independent Model (PIM) and a Platform Specific Model (PSM), which, as the names suggest, are two different ways to describe a given system.

A common use of the MDA – and the one we are pursuing – is to specify the functionality of a system in a domain specific language. Such a specification constitutes a PIM, the Platform Independent Model of the wanted system. Details about the domain we address are omitted here, but are available in a separate paper (Borch et al., 2003).

For a PIM to be realized as a running system, the declarations it contains must be turned into *artefacts* (often software programs, database tables, etc.) that can be deployed on a computing platform. The artefacts needed to realize a given PIM on a given platform constitute a corresponding PSM for the platform. In the MDA terminology, it is said that a PIM is *transformed* into a PSM.

The model-based approach to user interface engineering is not a new concept in HCI, yet it has not been successful in commercial environments (Szekely, 1996). One likely cause is the generally

low adoption of model-based approaches in commercial system engineering (OMG, 2003).

Possibly, model-based user interface development can benefit from the awareness that OMG's Model Driven Architecture gets, and leverage advances in modelling and transformation technology.

In this paper we sketch our current understanding of user interface engineering in the specific MDA context and outline a basic approach that we consider relevant from an HCI perspective.

2 HCI in the MDA

An important motivation for the MDA in general is the prospect for *separation of concerns* when modelling a system. Ideally, design and specification activities within each set of concerns are independent of other concerns. The typical example of the separation is that of system functionality in terms of the application domain versus concerns for the computing platform's performance, scalability, transaction support, security mechanisms, etc.

It is not clear that HCI concerns *per se* have a role in the MDA. Most HCI concerns cannot easily be described independently from other concerns for a system; usually, every aspect of a system has the potential to impact its users. We do not see that the MDA requires new HCI virtues, however, the attempt to model (i.e. make recipes for) systems in independent 'slices' to the benefit of core system engineering will affect associated HCI activities.

2.1 User interface models

In this paper, we discuss some implications of the MDA for user interface specification and engineering activities. In brief, we believe that the MDA puts additional focus on the means to create user interface specifications as *models*. While it is trivial to model the common graphical user interface components (window, icon, menu and widget) on their own, it is not straightforward to model user interfaces at higher abstraction levels and to cover all aspects, e.g. visual design, task guidance, error handling, and help facilities.

Yet, a *user interface model* is fundamental to circumscribe user interfaces in the MDA. Such models must encompass completely the intended structure and behaviour of a user interface, its temporal as well as permanent features, and it must be *bound* to the data it represents. Binding is discussed later on, but a further qualification of the user interface model concept is beyond the scope of this paper. See (Szekely, 1996) for a discussion of the nature of user interface models.

3 Our approach

We set out with the ambition to generate complete Java 2 Enterprise Edition (J2EE) applications (Sun Microsystems, 1999) in a fully automated way from a declarative system specification in a domain specific language. Our program generator tool reads the system specification (the PIM) and generates the required J2EE artefacts (the PSM) that can be installed and run on a J2EE application server. We chose J2EE as it provides a realistic, yet manageable platform for running a distributed application.

3.1 The concrete user interface

Among the generated artefacts is the client application that allows users to use the system. The choice of client technology posed a principal design question. First and foremost, the choice was between a standard client (e.g. an internet browser) and a custom-designed ‘fat’ client application. Traditionally, fat clients are considered expensive to develop and maintain, but this may no longer be the case if the fat client is automatically generated (qua the MDA) from a user interface model in a sufficiently expressive language. To incorporate changes to a generated fat client in this case, the system engineer would need only to update the model to reflect the changes and then generate a new application.

We decided to utilize the MDA for the client application as well, and make our generator tool

generate a fat client that runs on top of the Java Foundation Classes (Sun Microsystems Inc., 2003), which provide basic WIMP style user interaction services.

3.2 When to *bind* the user interface

The second principal design question on the client-side was that of *time of binding*, i.e. when is the user interface tied to the data it represents?

Regardless of presentation issues and visual design (which we do not discuss in this paper) the client application obviously needs data to support its operation. Data may come from any source; it can be placed directly in the model, or the model can contain references to the location where the data exists, such as a database. Either way, the model must directly or indirectly specify the data that goes into the user interface. *Binding* is loosely the process of making a connection between *one that holds* data (e.g. a database) and *one that uses* data (e.g. a user interface component).

In principle, binding can take place at any time during the user interface model transformations (PIM-to-PSM) or, ultimately, at some point during runtime. Generally speaking, *early binding* benefits runtime performance (allowing for low response times) but freezes specific choices; *late binding* adds levels of indirection, which provides variability but incur the cost of time-consuming look-ups (as well as an extra complexity to the engineering part).

There are compelling reasons to differentiate binding strategies dependent on the kind of data involved, e.g. to incur look-up overhead only when necessary. However, in order to reduce complexity we have so far not employed more than one binding strategy. For the time being, we use only late runtime binding, as this is a single binding strategy that allows us to bind to all relevant kinds of data, as explained next.

3.3 What to bind in the user interface

We identified three fundamentally different kinds of data that need representation in the user interfaces of the generated client applications, and thus need to be bound:

Type 1. The business data from the domain. This data is residing in a database on the server.

Type 2. Data about the state of the system itself, which is used e.g. to show system progress when the user is waiting for the system. This data is available by system reflection or similar service.

Type 3. Static data for help facilities, graphics, etc. This data is available locally or on the server.

Binding to the data about the state of the system's operations (Type 2) is what demands the late runtime binding strategy, since this data cannot effectively be bound before the system actually exists.

Now, deciding on the kind of data to bind is obviously only one side of the story; specifying the proper user interface components, these components' static and temporal relationships, and which components should represent which data is the necessary other side. In its current incarnation, our generator tool does only a simple mapping from user interface model element to user interface component. Details of our presentation specification style are omitted in this paper.

We use the ubiquitous Model-View-Control pattern (Krasner & Pope, 1988) to integrate data, presentation and user controls when the client application runs.

4 Concluding remarks

We believe the MDA holds some promises from an HCI perspective, primarily – as is routinely said about model-based development – because it may directly lower the cost of constructing and maintaining high-quality systems, and thus benefit users and the people involved in bringing the systems to them.

But paradoxically, the modelling aspect of the MDA seems to be a limiting factor to it being an all-round approach when it concerns user interfaces. First of all, it takes a considerable effort to reach mature *ways* to express user interface models, which we loosely call *modelling styles*. If a system engineering team is not sure that it can re-use a particular modelling style, it can be too risky to devote resources to craft and/or adopt it.

Secondly, by definition, the MDA models are the result of abstractions “suppressing selected detail to establish a simplified model” (OMG, 2003). Thus, in the course of finding modelling styles, compromises are struck between generality and individuality. As design philosophies and practices look at the world differently, it follows that no single modelling style can aspire to become universal. For example, even if user tasks and domain models are natural ingredients in a user interface model, it is not obvious whether a task orientation or a domain orientation should dominate, and current model-based development tools have chosen differently in this respect (Szekely, 1996).

Finally, one can argue that all systems are profoundly unique. Then every system will require a separate user interface modelling style, in which case the MDA has little to offer.

4.1 Domain specific modelling styles

On the other hand, it seems that when groups of systems have commonalities at a level that allows their user interface models to share a common modelling style, the MDA promises can hold.

In our case, generation of the client application complements the generation of the complete system, as previously described. Consequently, besides a user interface modelling style, we have also a ‘core system modelling style’ (i.e. the *style* of the PIM) that covers the specific business domain we target. So it is not surprising that the J2EE applications coming from our tool have commonalities that we can exploit when choosing the modelling style for the user interfaces. For example, since we work in a specific business domain, we know in advance the *types* of user tasks that the generated user interface must support. Thus, our user interface models can refer directly to these domain task types, and we need not model the task types explicitly. By referring to domain specific task types, we obviously make our user interface models domain specific as well. That is not a problem in our case, since we use our own equally domain specific generator tool.

4.2 System reflection at runtime

We found that the issue of binding the user interface components to the core system was a little less trivial than expected. We wanted the client application to have the means to query the system server (at the J2EE application server) about current processing status in order to inform the user during long waits. A trivial problem, yet it required us to either control the order of artefact generation (to make sure the server parts existed before the client did) or to postpone binding until runtime, where we can reasonably expect that everything exists.

Again, this aspect turns up in our case only because the core system is also being generated; usually, model-based user interface development tools can assume known data types from the core system.

Of course, other solutions can be found to solve such binding problems, but so far we haven't had reason to do so. On the contrary, the availability of a *system reflection* interface has turned out to be a valuable asset for the user interface designer.

4.3 Further work

We continue to aim at getting the most out of model-based user interface engineering in a way that is aligned with the MDA approach. Our investigation has proved valuable to realize a working user interface modelling style (and corresponding generator capabilities) that seems suitable to the specific business domain we generate systems for.

It is evident that because we generate user interfaces in a specific domain, we have been able to avoid a lot of complexity. For example, our user interface models need not cope with the modelling of user interfaces in general; we have merely identified suitable user interface constructs to act as 'types' and then determined the required variability of each type. Only the chosen variability of the types is available for specification in the model; types are implicit due to the chosen modelling style.

We see a substantial reliance on a specific domain as the most straightforward way to realize the promises in the MDA on the short term. This holds also for the model-based user interface development.

In domain specific environments it seems feasible to agree on specialized user interface modelling styles, which allows the corresponding models to be semantically rich and focused descriptions suitable for fully automated transformations to concrete user interfaces.

In this case, the MDA has the potential to successfully bridge some of the gap between HCI and system engineering.

References

- Borch et al., 2003, A Model Driven Architecture for REA based systems, In the Proceedings of the Workshop on Model Driven Architecture: Foundations and Application, TR-CTIT-03-27, University of Twente
- Krasner & Pope, 1988, A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. Journal of Object Oriented Programming, 1988. vol. 1, no. 3, pp. 26-49, <http://citeseer.nj.nec.com/krasner88description>
- OMG, 2001, Object Management Group, Model Driven Architecture, <http://www.omg.com/mda/>
- OMG, 2003, MDA Guide Version 1.0
- Sun Microsystems Inc., 2003, Java Foundation Classes, Cross Platform GUIs & Graphics, <http://java.sun.com/products/jfc>
- Sun Microsystems, Inc., 1999, Simplified Guide to the Java 2 Platform, Enterprise Edition, http://java.sun.com/j2ee/j2ee_guide.pdf
- Szekely, P.: Retrospective and Challenges for Model-Based Interface Development. In Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces, (Vanderdonckt, J. Ed.). Namur University Press, Namur, 1996. <http://citeseer.nj.nec.com/szekely96retrospective>