Measuring the potential of domain-specific modelling techniques

Jorn Bettin
SoftMetaWare Ltd.
Wellington, New Zealand
+64 27 448 3507
jorn.bettin@softmetaware.com

Abstract

A small-scale example application is used to introduce a domain-specific notation for object-oriented user interfaces. This paper compares traditional software development (no abstract modelling), standard Unified Modelling Language (UML)-based software development, and software development based on domain-specific modelling. To allow a comparison, Lines of Code (LOC) serve as a measure of development effort, and Atomic Model Elements (AME) are introduced to measure the modelling effort.

1. Introduction

The development of any complex software system requires practical techniques to ensure conformance with architectural decisions. The UML provides notational means to specify object oriented software designs, but the code generation features of most commercial UML modelling tools rely on design models that mirror the level of abstraction of the implementation [BET 01]. 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.

To meet the goal of ensuring architectural conformance and to avoid the issues of using UML tools in the traditional way, domain-specific notations allow the specification of software at a higher level of abstraction, and allow to automatically generate significant parts of the implementation, thereby enforcing consistent use of design patterns.

2. Practical example: A simple commerce system

The example used in this paper deals with the application of domain-specific techniques to automate technical software design and implementation in the presentation layer. The example also provides a good illustration of the symbiotic relationship between frameworks and template-language based code generation. The open source Naked Objects GUI framework and a model-driven template-language based code generator forms the foundation of implementation of the example.

Naked Objects is a Java framework that allows the automated construction of a Java GUI which exposes Java business entities to the end user in an object-oriented paradigm. A detailed description of Expressive Objects can be obtained from www.nakedobjects.org. For the purposes of this paper, it is sufficient to know that the source code of the example makes

use of Naked Objects and that the framework – without any further coding effort – provides a standardised high quality user interface for the classes of the example.

The sample application discussed in this paper is a simple commerce system that allows customers to order products, thus the key business entities are *Customer*, *Order*, and *Product*.

3. Traditional UML model of the application

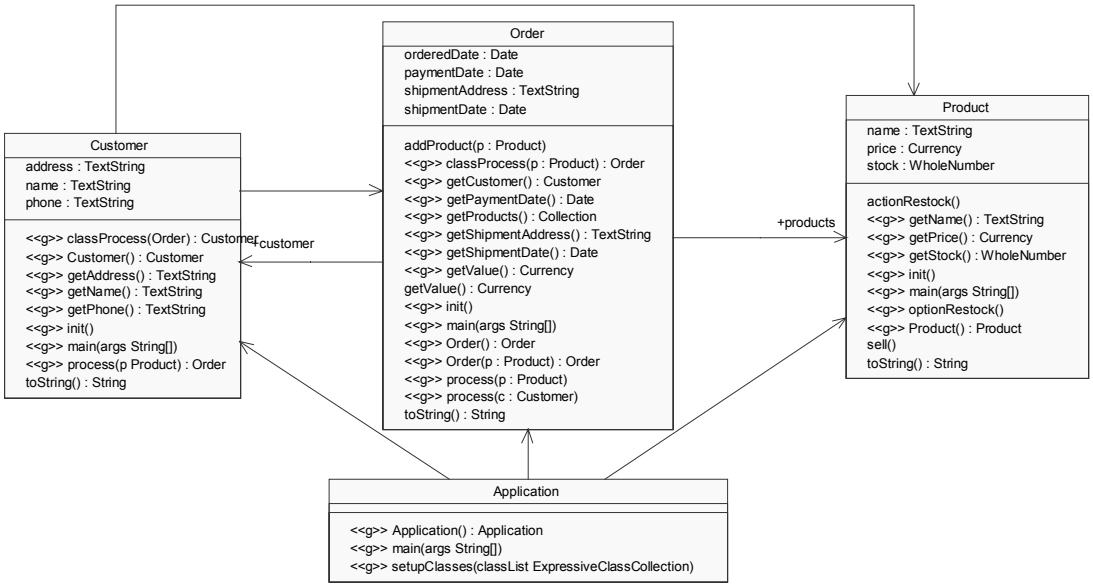

Figure 1

In figure 1 operations of stereotype <<g>> can be automatically generated to 100%, all other operations contain non-trivial business behaviour that needs to be partially implemented by hand. If operations of stereotype <<g>> are removed from the diagram, the remainder constitutes the domain model that determines the structure of the objects made available in the generated GUI.

4. Modelling object-oriented user interfaces

The drag-and-drop functionality of Naked Objects is easily specified in a simple behavioural modelling notation for the domain of object-oriented user interfaces. The notation has been chosen so that it can be understood by end-users. The visual representation of the notation, as described below, relies on the UML purely because this allows standard UML tools to be used to capture specifications.

In the context of our simple commerce system example, the visual notation for drag-and-drop behaviour is used as shown in figure 2 to specify user interface functionality of the application. This allows automatic generation of the glue code to implement drag-and-drop behaviour through the Naked Objects framework. In a small example it may not seem relevant whether drag-and-drop behaviour is specified directly in the code or through a visual model. In a larger application however, the large number of types and possible drag-and-drop operations rises significantly, so that the ability to specify allowable drag-and-drop operations visually in a format that can be reviewed by potential users of the application becomes very useful.

Behavioural Notation for Drag & Drop

An association with name **<name>** of stereotype **<<drop on object>>**, with navigability from class **A** to class **B** specifies that an object of type **A** can be dropped onto an icon representing an object of type **B**, and results in the creation of a new object of type **<name>**.

An association of stereotype **<<drop on class>>**, with navigability from class **A** to class **B** specifies that an object of type **A** can be dropped onto the icon representing the collection of objects of type **B**, and results in the creation of a new object of type **B**.

An association of stereotype **<<show>>** from the Application class to a class **A** specifies that an icon representing the collection of objects of type **A** is included in the main application window.

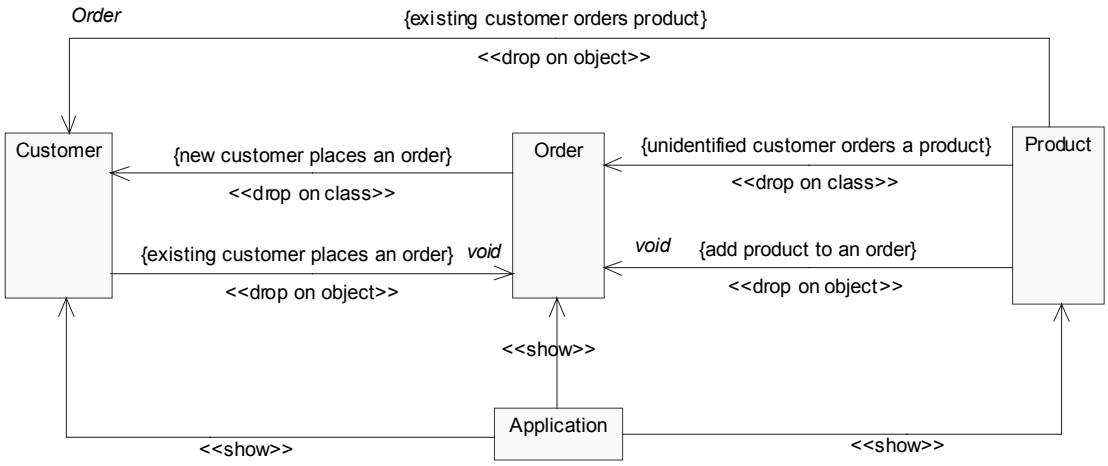

Figure 2

5. Model-driven development

With a template-language based code generator the behavioural specification can easily be translated into code automatically.

The same domain model can be used [with a different set of code templates and a different GUI implementation technology such as Swing] to generate other user interfaces, streamlined for specific delivery channels. This is of major importance, as it allows the core domain implementation to be reused even if another delivery channel needs to be bolted on.

Similarly if care has been taken in carving out the key, invariant abstractions relevant in the domain, the object-oriented user interface provided by Naked Objects should be fairly resilient against variations in business processes.

6. Metrics

Measuring software is fraught with difficulties in practice. One just needs to look at the attempts to define metrics for reuse to see that it is anything but an exact science. Nevertheless, software metrics can provide some objectivity, especially when different approaches to software development need to be compared. It is instructive to compare the effort required by the application developer to implement the solution using three scenarios:

- **Scenario 1: Manual coding** - no use of modelling tools whatsoever. The currently prevailing way of software development.
- **Scenario 2: Naïve UML modelling**, skeleton generation, and manual coding. Software development as recommended by the vendors of the current generation of UML tools
- **Scenario 3: Domain-specific modelling**, and minimised manual coding.

It is advisable to use a method of measuring implementation effort that is accepted by a broad audience ranging from IT managers, over hard-core programmers, to UML enthusiasts. Lines of code (LOCs) have been around for a long time, their advantages and drawbacks are well documented in [HES 96] and [POU 97], and the measuring process is simple: it can be specified unambiguously, so that when applied to the same code by different people, the results are guaranteed to be identical.

The LOCs technique can be extended to count the effort to produce graphical models by introducing an equivalent of a line of code, the *atomic model element* (AME). Measuring the effort then becomes a matter of counting AMEs. LOCs measure program size, and can arguably be used as an indicator of implementation effort - the usual concerns about the usefulness of LOCs apply.

For the purpose of this paper "implementation effort" is defined as the data entry effort for progam specifications measured in keyboard strokes and mouse operations. AMEs have been designed so that the data entry effort (in terms of keyboard strokes) for one AME using a typical visual modelling tool corresponds roughly with the data entry effort for a LOC.

Counting Atomic Model Elements

The following rules lead to very generous estimates of the UML modelling effort, but the results are nevertheless highly interesting:

- Each class, operation, or attribute used in a UML diagram
- Each link between model elements in a UML diagram [dependencies, associations, method invocations, etc.]
- Each role name used in a UML diagram
- Every stereotype used in a UML diagram

is counted as an atomic model element

Counting Lines of Java Code

Lines of Java code have been counted as follows:

- Each semicolon counts as one line of code, covering package definitions, import statements, variable definitions, and all statements within Java methods.
- Each class header counts as one line of code.
- Each method header counts as one line of code.

No extensive scientific analysis of keyboard strokes and mouse operations has been undertaken. The AME counting rules have consciously been chosen so that in the assessment of any experienced programmer or modeller, there is no question that the data entry effort for one average AME is at most that of entering one average LOC. The conservative counting rules should at least reassure hard-core programmers that the modelling effort has not been underestimated in the comparison. The application code is exactly identical in all three scenarios, and the skeleton generation facilities of today's UML tools are fully taken into account.

	Manual coding	Naïve UML modelling		Domain-specific modelling	
Manually coded LOCs	126	71		20	
Automatically generated LOCs	0	55		106	
Total LOCs	126	126		126	
Manual code [%]	100	56		16	
Static model [AMEs]	0	61		21	
Dynamic model [AMEs][1]	0	56	0	51	20
Total AMEs	0	117	61	72	41
Sum of manually coded LOCs and AMEs	126	188	132	92	61
Effort compared against manual coding [%]	100	149	105	73	48

Table 1

The results summarised in table 1 clearly show that the traditional use of UML modelling tools is not necessarily a step forward. Looking at the figures, and even acknowledging the conservative measurement of AMEs, it is very hard to argue the case for the use of UML tools and skeleton generation. In practice application developers quickly resort of "code-first, model-second" mode to increase the speed of development. In this context the heavily advertised "round-trip engineering" features of UML tools are often resorted to, because they allow developers to meet management-imposed UML design documentation requirements with less effort than in "model-first, code-second" mode. However the application developer will still need to manually arrange the reverse engineered UML diagram elements into an appealing format. This artistic effort is of questionable value, especially if the application evolves rapidly, and UML models quickly become outdated.

In contrast, a domain-specific modelling approach allows the use of abstract modelling to reduce the total development effort. Besides the obvious gain in productivity there are several advantages that can't be claimed by either the manual coding camp, nor the traditional UML camp:

- The level of abstraction of the model is higher, and the model is therefore simpler and easier to understand. By carefully designing domain-specific modelling notations, even non-software literate domain experts can at least read and validate a model.
- The model specification constitutes the main "source" of the application, implementation language source code loses importance - only certain parts of it still need to be maintained

[1] To account for different levels of documentation requirements, a separate scenario is broken out that does not include dynamic models, and in the case of domain-specific modelling only includes those dynamic models that serve as specifications for automated code generation.

and understood by the application developer. As a corollary, the model specification always accurately reflects the implementation.

- The model is implementation language independent, and is no longer tied to Java, C++, Oracle or other software technologies. Thus the model can easily withstand the constant churn of software technologies. If technological change does not necessitate a change of the model, the only changes affecting the model are changes to the domain - the evolution of the business. With model-driven development environments hooked into the domain-specific model, these domain changes can be implemented faster than ever before.

7. Conclusions

What conclusions can be drawn from the small sample application illustrated in this paper?

The sample application made use of a UI framework, and it shows how domain-specific techniques and the use of frameworks go hand in hand. The coding rules and discipline imposed by a framework such as Naked Objects are the ideal starting point to further raise the level of abstraction and to introduce high-level visual modelling notations. A code generator is much better at conforming to the strict rules than an application developer. Conversely without frameworks, the generation gap between model and implementation can become so wide that the complexity of the code generator skyrockets and generation becomes impractical.

The example only focused on generating code for Naked Objects from a model. In a real application more code would be required to address other architectural concerns such as persistence, time dependencies, distribution etc. The same techniques can be applied, starting from the same model, to automatically generate implementation code to cover these concerns. If this is done, the metrics shift even further in favour of domain-specific model-driven development: the model does not increase in complexity, but more and more code is automatically generated. It is easily conceivable to reduce the development effort to between 5 and 20% of manual coding depending on the type of application.

With frameworks and code generators significantly reducing implementation costs and with abstract models taking the place traditionally held by source code (the ultimate specification of application functionality), the value of abstract models will increase and the value of source code will diminish significantly. When this happens on a large scale, it will constitute a similar paradigm shift as the change of focus from assembler to higher-level languages three or four decades ago. Today it is hard to predict what the consequences of this paradigm shift will be; comparing the maximum complexity that could be handled in assembler language with the maximum complexity that can be handled in modern OO languages might give an indication of the order of magnitude.

8. References

[BET 01] Joern Bettin. Raising the Level of Abstraction of Design Models. *Conference Companion of the Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), 37-38 (October 2001).

[HES 96], Brian Henderson-Sellers. *Object-Oriented Metrics - Measures of Complexity.* Prentice Hall, (1996)

[POU 97], Jeffrey S. Poulin. *Measuring Software Reuse – Principles, Practices, and Economic Models.* Addison-Wesley, (1997)