Domain-Specific Modelling for Cross-Platform Product Families

Steven Kelly & Risto Pohjonen

MetaCase Consulting, stevek@metacase.com, rise@metacase.com

Abstract. Domain-specific modelling has proved its worth for improving development speed and dependability of applications. By raising the level of abstraction away from the code and towards the structure and behaviour of an application, it also offers good possibilities for generating the same application from the same models, but for a wide variety of client platforms. This paper examines one example of domain-specific modelling for an embedded application, and how that was extended to a mobile platform.

1 Introduction

This paper presents a modelling language for implementing software applications for cross-platform digital wristwatch applications. In general it is an example of how such a domain-specific modelling environment (later referred to as DSM environment) can be implemented in a metaCASE tool such as MetaEdit+. In particular, we are interested in how well a DSM approach can adapt to cope with new platforms.

In this section we discuss DSM and the watch modelling language from the point of view of a modeller using it. Section 2 describes the architecture behind the watch example from the point of view of a metamodeller creating the DSM environment. Section 3 looks at the extension of the watch example to the MIDP world. MIDP is the Mobile Information Device Profile, a Java platform for building applications to run on small devices such as mobile phones or PDAs. As the MIDP platform did not even exist when the watch example was originally made, it provides a good test of how well a DSM solution can cope with major unforeseen changes in the platform.

First, a proviso: the watch example is not strictly a real industrial example, since the authors are not in the business of making digital watches. It was included with the authors' MetaEdit+ metaCASE tool as a fully-worked example, and thus aims to be as realistic as possible. With the advent of mobile phones supporting MIDP, it is now a true application with real users. Given the dearth of experience reports on DSM, hopefully even with the proviso this report will still prove useful.

1.1 Introduction to Domain-Specific Modelling

Why develop a DSM environment for modelling watches? Why not adopt some pre-existing general-purpose modelling language and a 'standard' CASE tool, then write code by hand? While it would be possible to apply such ‘standard’ technology here, there is much more to be gained by using a DSM. Let us consider the following reported benefits of the use of DSM in software development:

1. **Productivity increases by as much as a factor of 10.**
   Traditional software development has required several error-prone mappings from one set of concepts to another (Seppänen et al. 1996). First the domain concepts must be mapped to the design concepts and then further mapped to the programming language concepts. This is equivalent to solving the same problem over and over again. With DSM, the problem is solved only once by working with pure domain concepts. After that, there is no need for mapping as the final products are automatically generated from these models. Studies have shown that this kind of approach is 5-10 times faster than the usual current practices (Weiss and Lai 1999, MetaCase 1999, Kieburtz et al. 1996).
2. **Better flexibility and response to change.**
   Focusing on design rather than code results in a faster response to requests for changes. It is easier to make the changes at the domain concept level and then let the tool generate code for multiple platforms and product variants from a single set of models.
3. **Domain expertise shared with the whole development team.**
   The usual problem within development teams is the lack of platform knowledge among the developers. It takes a long time for a new developer to learn enough to become productive. Even the more advanced developers make mistakes in coding. In the approach presented here, an expert defines the domain concepts, rules and mapping to code. Developers then make models with the concepts guided by the rules, and code is automatically generated.

These are vital issues if development involves more than one of the following: domain-specific knowledge, product families (variants of similar products), medium to large development teams, critical time-to-market factors, and a strong need for quality. Other papers have described DSM in more detail (e.g. Kelly 2000, Tolvanen 2001); here we shall concentrate on an experience report about the watch example and its extension to the MIDP platform. First we will look from a modeller’s point of view.

1.2 The Watch Modelling Language

The watch modelling language itself consists of two diagram types. First there is a WatchFamily diagram that describes the models in the watch family. It also describes the displays and logical watch applications that have been used to create the models. Fig. 1 shows such a diagram, with a family of related wristwatch models in the **Models** group at the top of the diagram, and the logical watch application and display components are presented in the two groups at the bottom of the diagram. Each Model consists of one LogicalWatch and one Display.

Displays are kept deliberately simple: all their information is displayed in this graph. Each Display specifies its available icons, zones for displaying time digit-pairs, and buttons. Logical Watches are more complex and are further defined in sub-graphs. For

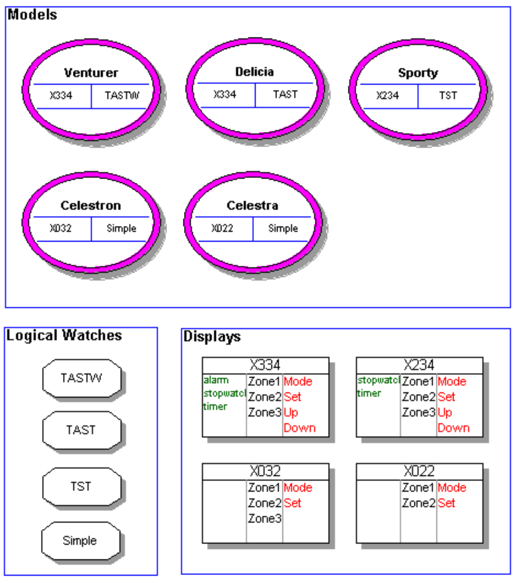


Fig. 1. The WatchFamily diagram

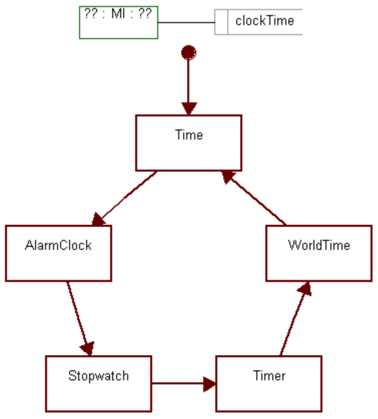


Fig. 2. The WatchApplication diagram

instance, the sub-graph defining the contents of the 'TASTW' Logical Watch is the 'TASTW' WatchApplication diagram as shown in Fig. 2.

This diagram shows the configuration of the logical watch application. It only contains the top-level logical configuration of sub-applications, basically showing which sub-applications have been included into this specific logical watch and in which order they are invoked. When the logical watch application is started (i.e. the watch is powered up), the basic Time sub-application will be invoked. If this sub-application is exited, an AlarmClock sub-application will be started. The cycle is completed when Time will be re-activated when exiting from the WorldTime sub-application. The name 'TASTW' comes from these sub-applications' initials in order.

Each sub-application is further defined in a sub-graph, which describes the implementation of that sub-graph. A simple example for the Time sub-application is shown in Fig. 3. Whilst the type of this graph is actually the same as that of TASTW, it is

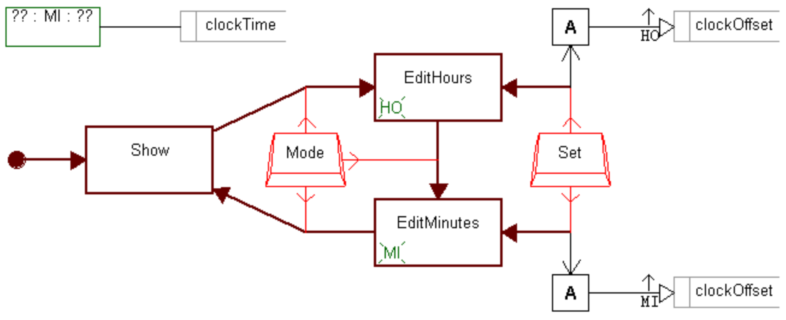


Fig. 3. The Time WatchApplication diagram

clearer here that this is not an ordinary state transition diagram. The basic states and transitions can be seen, but there are customized semantics and many domain-specific additions, to enable the use of such domain-specific concepts as buttons and alarms.

The Mode button triggers transitions between the three states. While in one of the Edit states, the Hours or Minutes zone on the display will flash, and the Set button triggers Actions that increment the Hour or Minute components of the clockOffset variable. The DisplayFn at the top left defines that the time displayed by the watch will simply be the current clockTime.

1.3 The Watch Modelling Language as a Development Platform

Breaking the watch apart in this way results in a high level of reusability. Since sub-applications, logical applications and displays can be defined separately from each other, and they communicate via pre-defined interfaces, they make natural components. This enables the developer to build new watch variants quickly by combining sub-applications into new logical watch applications and then combining these with displays (new or existing).

When developing using the Watch Modelling Language, there is no need to write any code. The graphical models form the complete specification of a watch and its behaviour, and the code generated from them is complete.

2 The Watch Architecture, Code Generators & Framework Code

Having looked at the Watch Modelling Language from the point of view of a modeller, we now turn to look at how it actually works. These details are hidden from the modeller, who can thus concentrate on building new watches and applications without knowing them or considering them.

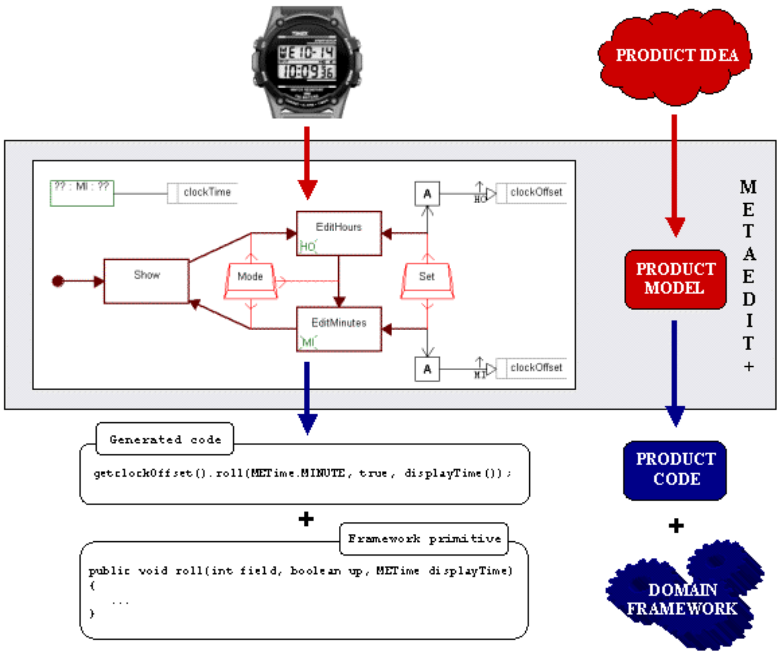
In real life the platform in this kind of case would be an electronic device controlled by a microchip. For an example to accompany MetaEdit+ this was of course not a viable option. As we wanted our test environment to run on any desktop, Java was chosen as the implementation language, and browsers as the runtime platform.

For the record, the whole project of designing and implementing the first working version of the Watch DSM language with one complete watch model took eight man-days for a team of two developers. Neither developer had prior experience of Java programming, or of building watch software, and there were of course no pre-existing in-house watch components. It took five days to develop the Java framework, two days for the modelling language, and one day for the code generator. These times include design, implementation, testing and basic documentation.

Since then, new watch models with new functionality have been implemented in fifteen minutes with this environment. As we estimated that it would have taken five to six days to develop the first watch model manually, and then one day for each additional watch model, it is fair to assume that the third watch model completed the development effort payback.

2.1 The Watch Architecture

The architecture for a DSM environment like our watch example generally consists of three parts: a modelling language, a code generator and a domain framework. To understand the role of each these within the architecture, we have to understand how the responsibilities are distributed among them. The basic principle of this distribution is illustrated in Fig. 4.



```
getclockOffset().roll(METime.MINUTE, true, displayTime());
```

```
public void roll(int field, boolean up, METime displayTime)
{
   ...
}
```

Fig. 4. The watch architecture

While designing and implementing the architecture for our watch example, we wanted to solve each problem in the right place and on the best possible level of abstraction. The modelling language was assigned to capture the behavioural logic and static aspects of watch models and applications, while the domain framework was created to provide a well-defined set of services for the code generator to interface to. Having already covered the modelling language, in the rest of this section we will examine the code generator and the domain framework classes the generated code interfaces with.

2.2 The Code Generator

The basic idea of the code generator within a DSM environment is simple: it runs through the models, extracts information from them and translates it into code for the target platform. With the models capturing all static and dynamic logical aspects and a framework providing the required low-level support, the code generator can produce completely functional and executable output code.

The MetaEdit+ report generator used to produce the code also provides a flexible tool to automate integration of the DSM environment with other tools. An example of this kind of integration within the watch example is the auto-build mechanism that enables us to automatically compile the generated code and execute it on a testing environment. This automation results in major savings of both the effort and time during the testing.

In the watch example, the auto-build proceeds with following steps:

1. **The scripts for compiling and executing the code are generated.** As the mechanism of automated compilation and execution varies between the platforms, the number and content of generated script files depend on the target platform.
2. **The code for framework components is output.** All framework code has been included into the watch models as attached Java files and is output from there as needed. This way we ensure that all required components are available at the time of compilation and have the control over the inclusion of platform specific components.
3. **The code for logical watches and watch applications is generated.** The state machines are implemented by creating a data structure that defines each state transition and display function. For each action the code generator creates a set of commands that are executed when the action is invoked.
4. **The generated code is compiled and executed as a test environment for the target platform.** Basically this step requires only the execution of the scripts created during the first step.

The structure of the code generation is show in Fig. 5 and 6. The code generators in MetaEdit+ are defined with a dedicated textual report definition language. Each report definition is associated with certain graph type and thus can operate on models made according to that specific graph type. These report definitions – that could be also referred to as sub-generators – can be arranged in a hierarchical fashion. The top level of the code generator architecture of the watch example (i.e. the sub-generators associated with WatchFamily graph type) is presented in Fig. 5 (a '*' in the name of a sub-generator denotes an individual version for each target platform).

The top-level generator is called 'Autobuild'. It handles the whole generation process by simply calling the sub-generators on the lower level. The sub-generators on the next level relate closely to those steps of the auto-build process presented earlier in this section. As '_JavaComponents' only outputs the pre-defined Java code for the framework components and '_compile and execute *' only executes scripts produced during the earlier steps of the generation process, we can concentrate on '_create make for *' and '_Models'.

The basic task of '_create make for *' sub-generators is to create the executable scripts that will take care of the compilation and execution of the generated code. As this procedure varies between platforms, there is an individual version of this sub-generator for each supported target platform. If there are any specific platform-related generation need like HTML for browser-based test environment in Fig. 5, they can be integrated with the '_create make for *' sub-generator.

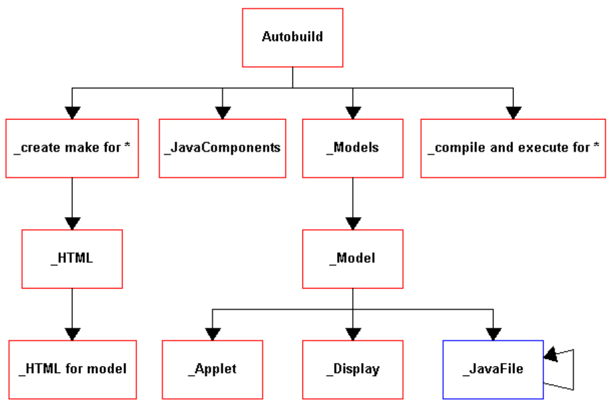


Fig. 5. The watch code generator architecture, part 1

The responsibility of the '_Models' and '_Model' sub-generators is to control the generation of code for the watch models, logical watches and watch applications. For each watch model, three pieces of code are generated: an applet as the physical implementation of the user interface, a display definition about the model specific user interface components, and the definition of the logical watch.

To understand how the code for a logical watch and a watch application is generated, we need to explore the code generator architecture further. The lower level of the architecture (i.e. the sub-generators associated with the WatchApplication graph type) is presented in Fig. 6.

The sub-generators '_JavaFile' (which is the same as in Fig. 5) and '_Java' take care of the most critical part of the generation process: the generation of the state machine implementations. To support the possibility to invoke a state machine from within another state machine in hierarchical fashion, a recursive structure was implemented in

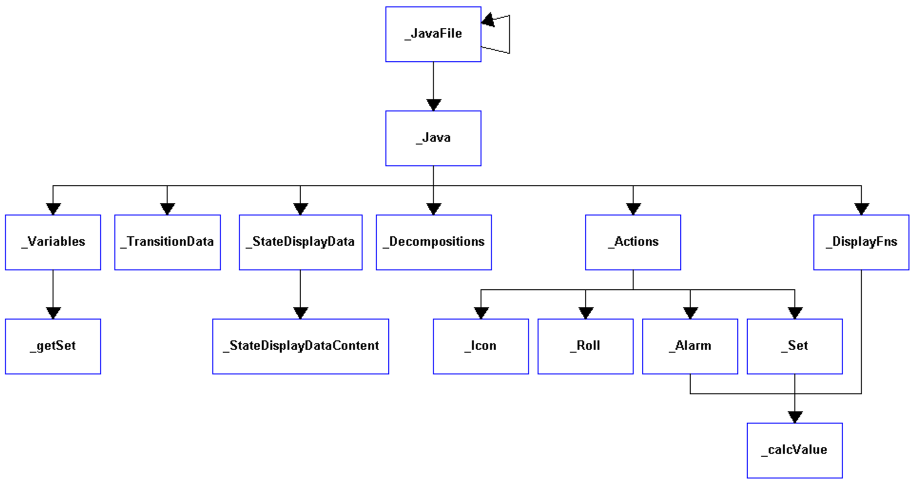


Fig. 6. The watch code generator architecture, part 2

the '_JavaFile' sub-generator. During the generation, when a reference to a lower-level state machine is encountered, the '_JavaFile' sub-generator will dive to that level and call itself from there.

Reading the third level of generators (below '_Java'), we can see the main parts of the code generation for each Watch Application. First, any variables defined in the model are defined, along with getter and setter functions. The next three generators fill in data structures that record the states, their display functions, transitions, actions and decompositions. The generators under '_Actions' generate functions containing procedural code to execute the various actions specified in the models by the types with the same names. '_DisplayFns' works similarly to generate the code needed to calculate the time to be displayed.

2.3 The Domain Framework

From the point of view of the DSM environment, the domain framework consists of everything below the code generator: the hardware, operating system, programming languages and software tools, libraries and any additional components or code on top of these. However, in order to understand the requirements set for the framework to meet the needs of a complete DSM environment, we have to separate the domain-specific parts from the general platform related parts of the framework. The platform is considered to include the hardware, operating system, Java programming language with GUI classes, and an environment to test our generated code (either browser or MIDP emulator). The architecture of the watch domain framework is presented in Fig. 7. Solid line arrows indicate the instantiation relationship while dotted line arrows indicate inclusion relationships between the elements.

The domain architecture of the watch example consists of three levels. On the lowest level we have Java classes that are needed to interface with the target platform. The middle level is the core of the framework, providing the basic building blocks for

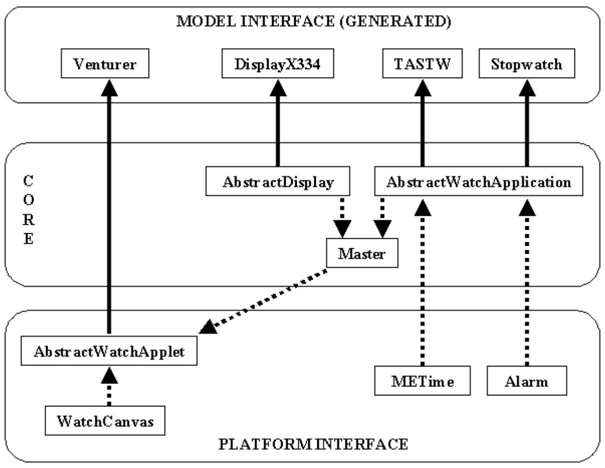


Fig. 7. The watch domain framework

watch models in the form of abstract superclass ‘templates’. The top level provides the interface of the framework with the models by defining the expected code generation output, which complies with the code and templates provided by lower levels.

There are two kinds of classes on the lowest level of our framework. METime and Alarm were implemented to raise the level of abstraction on the code level by hiding platform complexity. For example, the implementation of alarm services utilizes a fairly complex thread-based Java implementation. To hide this complexity, class Alarm implements a simple service interface for setting and stopping alarms, and all references from the code generator to alarms were defined using this interface. Similarly, METime makes up for the shortcomings of the date and time implementation of the Java version used. During the code generation, when we need to set an alarm or apply an arithmetic operation on a time unit, the code generator produces a simple dispatch call to the services provided by these two classes.

The other classes on the lowest level, AbstractWatchApplet and WatchCanvas, provide us with an important mechanism that insulates the watch architecture from platform-dependent user interface issues. For each supported target platform, there is an individual version of both of these classes and it is their responsibility to assure that there is only one kind of target template the code generator needs interface with.

On top of the platform interface and utilizing its services is the core of the framework. The core is responsible for implementing the counterparts for the logical structures presented by the models. The abstract definitions of watch applications, logical watches and displays can be found here (the classes AbstractWatchApplication and AbstractDisplay). When the code generator encounters one of these elements in the models, it creates a concrete subclass of the corresponding abstract class.

Unlike the platform interface or the core levels, the model interface level no longer includes any predefined classes. Instead, it is more like a set of rules or an API of what kind of generator output is expected when concrete versions of AbstractWatchApplet, AbstractWatchApplication or AbstractDisplay are created.

3 The Watch Example for MIDP

Mobile Information Device Profile (MIDP, 2000) is a set of Java APIs which provides a standard application runtime environment targeted at mobile information devices, such as cellular phones. The phone contains a Java virtual machine, and the user can download mini-applications as .jar files and run them on the phone. Application size is often restricted to 30kB, in particular for downloading applications wirelessly.

Such an environment provides an interesting application for the previously fictitious watch example. It also presents a challenge: is the solution we developed for digital watches still viable if we extend the domain to include watch applications on a MIDP phone? Changes could be necessary on four levels:

– The domain-specific modelling method and its metamodel
– The models
– The code generator
– The framework code

A moment's thought reveals that changes to the models (or certain changes to the metamodel which do not update models automatically) are the worst kind: many developers would potentially need to update many models. If the initial domain analysis has been good these can hopefully be avoided, or at least limited to backwards-compatible additions to support new functionality in the new platform.

Changes to the code generator or framework code will require only the metamodeller's time, similarly for changes to the metamodel which automatically update relevant parts of the models. In particular, code generator changes can allow us to support a new platform with a fraction of the time normally needed — even though there are changes needed throughout the whole body of code.

3.1 Changes to Support MIDP

Extending the Watch example to support MIDP required surprisingly few changes. In fact, more changes were made because of minor problems noticed in the initial code than because of MIDP.

3.1.1 Metamodels

No changes were required to the metamodels to support MIDP. To simultaneously support generation of both MIDP and the older Java, a new property was added to the top-level graph type. This allowed each framework classes to specify the target platforms for which it was intended.

3.1.2 Models

No changes were required to the models to support MIDP.

3.1.3 Generators

We had made three different implementations of a Java state machine whilst making the original watch, with ideas sketched out for another two. The implementation we went with required the reflection abilities of Java, which unfortunately are not present in MIDP. Hence we moved to a switch case based implementation, using initialised static final variables as labels.

This required an addition to the _Variables generator to generate the new static final variable for each Action and DisplayFn. Similarly, a minor change was made to the _TransitionData generator to generate the variable names rather than a string containing the same text.

The _Actions and _DisplayFn generators were similarly changed to place their body inside a case statement, rather than a similarly-named function.

A larger amount of work was required for the new '_create make for MIDP' generator. MIDP compiles its Java as for other platforms, but it also requires a preverify step, and a couple of configuration files naming and providing information about the MIDP suite (Watch family) and the MIDP applications (Watch models) it contains. Normally these configuration files would be written by hand, or filled in to a form, but in our case all the information needed can be obtained from the models.

3.1.4 Framework code

The original code was much in need of refactoring, having been the authors' first Java application, and not really intended to be maintained. First we refactored out the mass of user-interface, control and state machine behaviour from the applet into their own classes. From this, it was easier to see what had to be done.

The majority of classes were platform independent, requiring only basic Java functionality. The user interface and control APIs are different for MIDP, so a separate WatchCanvas class had to be made for MIDP. Being a second attempt at the same functionality, with more Java experience than before, it was soon noticed that the same solutions could be applied to the WatchCanvas class for applets too. This resulted in smoother updating in the applet, as well as keeping the applet and MIDP versions more visually similar.

MIDP does not have the Applet class, so our Applet was replaced with a Midlet, the MIDP equivalent. As our generated applet classes subclass from AbstractWatchApplet, our framework subclass of Applet, they work as subclasses of AbstractWatchApplet just fine, even when it is a subclass of Midlet. Thus, no changes were needed to the generation of the applet/midlet for each WatchApplication.

3.2 Results

With the above changes, the Watch modelling environment is now capable of making applications that run on a variety of MIDP and other Java platforms, adapting its display and user interface to the platform. For example, display size and fonts vary widely, as do buttons: some phones have up and down buttons physically available, other buttons are implemented as soft-keys or via menus. The modeller does not have to worry about these details: they are handled by the domain framework code.

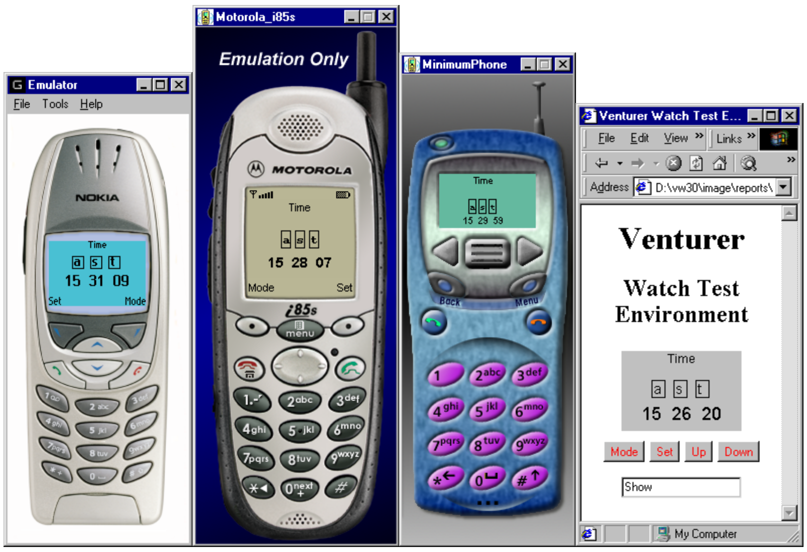


Fig. 8. Watch application in Nokia, Motorola and Sun emulators, and IE

4 Conclusion

Domain-specific modelling offers product family developers a high level of insulation from surrounding platform changes. The same Watch models have survived virtually untouched through changes from Java to Java2 to MIDP, with the main changes required being made by one person in only one place. Without DSM, the majority of developers would have to update most of their features for each platform change.

In particular, DSM offers excellent support for a family of products across a family of platforms. The current Watch models are capable of generating code for each of the three platforms, and on a variety of operating systems. Without DSM, there would quickly be no hope of maintaining one code base for all platforms.

These advantages are over and above those which come from making products by visual domain-specific modelling instead of writing textual code — a change which in itself normally increases productivity by 5 to 10 times. As always, however, DSM is only appropriate if there is a sufficient body of similar applications to be developed. Our experiences concur with those of Weiss et al. (1999): three watches would have been enough to offset the cost of building the DSM environment.

In supporting the MIDP platform, the majority of changes were in the new MIDP framework code and its build script. Minor changes were necessary to the code generation to work in the more restricted MIDP environment. Altogether, the changes took four man-days: 2 for the MIDP framework code, 0.5 for the MIDP build script, 1 for refactoring existing framework code, and 0.5 for adapting the code generation.

References

Kelly, S., Tolvanen, J-P, Visual domain-specific modelling: Benefits and experiences of using metaCASE tools, In: International workshop on Model Engineering, ECOOP 2000, Ed. J. Bezivin, J. Ernst (2000)

Kieburtz, R. et al., A Software Engineering Experiment in Software Component Generation, In: Proceedings of 18th International Conference on Software Engineering, Berlin, IEEE Computer Society Press (March 1996)

MetaCase, Nokia case study, http://www.metacase.com/ (1999)

MIDP, JSR-000037 Mobile Information Device Profile (MIDP), Final Release, http://jcp.org/jsr/detail/37.jsp (2000)

Seppänen, V., Kähkönen, A. -M., Oivo, M., Perunka, H., Isomursu, P., Pulli, P., *Strategic Needs and Future Trends of Embedded Software*. Technology Development Centre, Technology review 48/96, Sipoo, Finland (1996)

Tolvanen, J-P, Kelly, S., Modelling Languages for Product Families: A Method Engineering Approach, In: Proc. of OOPSLA Workshop on Domain-Specific Visual Modeling Languages, Jyväskylä University Press, (2001) 135–140

Weiss, D., Lai, C. T. R., *Software Product-line Engineering*, Addison Wesley Longman (1999)