Version 4.0

The Watch Example

MetaCase Document No. WE-4.0

Copyright © 2005 by MetaCase Oy. All rights reserved

First Printing, 3rd Edition, January 2005.

MetaCase
Ylistönmäentie 31
FIN–40500 Jyväskylä
Finland

Tel: +358 14 4451 400
Fax: +358 14 4451 405
E-mail: info@metacase.com
WWW: http://www.metacase.com

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including but not limited to photocopying, without express written permission from MetaCase.

You may order additional copies of this manual by contacting MetaCase or your sales representative.

The following trademarks are referred to in this manual:
CORBA and XMI are registered trademarks and UML and Unified Modeling Language are trademarks of the Object Management Group.
HP and HP-UX are trademarks of Hewlett-Packard Corporation.
Linux is a registered trademark of Linus Torvalds.
MetaEdit+ is a registered trademark of MetaCase.
Microsoft and Windows are registered trademarks of Microsoft Corporation.
Motif is a trademark of the Open Software Foundation.
Pentium is a trademark of Intel Corporation.
Solaris and Sun SPARC are registered trademarks and Java is a trademark of Sun Microsystems.
UNIX is a registered trademark of X/OPEN.

Preface

The goal of this example is to demonstrate the features and implementation of a domain-specific modeling environment to the reader. The example will cover the issues of designing and implementing the modeling language, building tool support for it (including 100% code generation), working with it and finally extending it. Please note that certain parts of the example may require working hands-on to ensure the best understanding of the subject matter.

For exploring the Watch example thoroughly, the following things are required:

- Java 2 Platform, Standard Edition (J2SE) version 1.4.2 or later for compiling the Java code produced by the example code generator. Optional features presented in chapters 4 (support for MIDP platform) and 5 (the use of MetaEdit+ API) require the installation of additional components – for more information about them, see their respective chapters.

  J2SE for Windows can be downloaded from java.sun.com and it should be installed in `c:\java\j2sdk`. On Linux we recommend the use of the free IBM JDK version 1.1.8 that is available at http://www.ibm.com/java (to download, go to the 'Tools and Products' section, choose item called 'IBM Developer Kit and Runtime Environment, Linux, Version 1.1.8' and proceed with 'Register and download'). This distribution installs its binaries in `/usr/jdk/java118/bin` directory by default and thus this is the Java compiler home path expected by the code generator.

  It is possible to have J2SE installed somewhere else than the directories recommended here, but the code generator uses these directories by default so you would have to make some changes. For instance, you could compile and execute the code manually, set appropriate symbolic links, or change the path shown with bolded text in the WatchFamily '_create make for Windows/Linux' reports.

- A web browser with Java support.

- MetaEdit+ Method Workbench for trying out the extensions presented in chapter 2.5. The hands-on examples in chapter 5 require either MetaEdit+ Method Workbench or MetaEdit+ CASE version with API support. The rest of the example can be explored with any version of MetaEdit+.

For further information about MetaEdit+, please refer to the 'MetaEdit+: User's Guide', 'MetaEdit+ Method Workbench: User's Guide' or our web pages at http://www.metacase.com.

1 The Watch Example

The Watch example presents a modeling language and its tool support specifically tailored for implementing software applications for digital wristwatches. In general it is an example of how such a domain-specific modeling environment (later referred to as DSM environment) can be implemented in MetaEdit+. In this case we are discussing a language designed for embedded and real-time systems, but the basic principles of this example can be applied to other domains as well.

In this chapter we discuss the ideas behind the Watch example and its implementation. Chapter 2 then explains how to access the example and how to play around with the Watch modeling language. Chapter 3 provides an autopsy of the architecture behind the Watch example while chapters 4 and 5 are dedicated to the advanced topics of extending the example to the MIDP world and implementing visual debugger with MetaEdit+ API.

Please note that walking through the watch example requires a basic knowledge of how to use MetaEdit+. A good starting point to get this knowledge is the Family Tree example in the 'Evaluation Tutorial'.

1.1 THE BASIC IDEA OF THE WATCH EXAMPLE

Why develop a DSM environment for modeling watches instead of adopting some pre-existing general-purpose modeling language and a 'standard' CASE tool? 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. 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 on the best possible level of abstraction 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.

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 domain knowledge among the developers. It takes a long time for a new developer to learn enough to become productive. Even the more advanced developers need to consult with domain experts frequently. In the approach presented in this tutorial, the 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 very important issues if your 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.

Thus the basic idea here is to show you, through the watch example, how you too can reap these benefits by developing a DSM environment for software development in your own domain. This is done in two stages. To begin with we will explain how such an environment is developed in the first place. Secondly we will show how it is used to develop applications for its specific domain.

1.2 THE DEVELOPMENT OF THE WATCH EXAMPLE

The development of the watch example closely resembles the implementation of any DSM environment. In this kind of venture, there are always three related entities: the problem domain, the target platform and the DSM environment as a bridge between them. As the domain and target platform are the parts that already exist, the task in making a DSM environment is to find their key entities and then define mappings between them.

For this example, we needed a domain that we knew well enough ourselves and that could be easily understood by developers from different domains. The domain chosen was digital watch, in particular a family of related wristwatch models. The DSM environment should allow development of watches by modeling them and then automatically generate fully functional watch applications directly from the models.

As the domain in this case is still quite simple, finding the key domain concepts was fairly easy. A digital watch can be divided into a physical display unit, and a logical watch specifying the behavior. The display unit defines the widget and button configuration for the watch. The display widgets are either icons that indicate whether or not a service (like alarm) is activated, or two-digit display zones that can show basic time units like seconds, minutes or hours. The logical watch application defines the required functionality as a set of sub-applications like alarms and stopwatch.

Breaking the watch apart in this way results in very good reusability. As 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). This idea of watch configuration is illustrated in Figure 1-1.

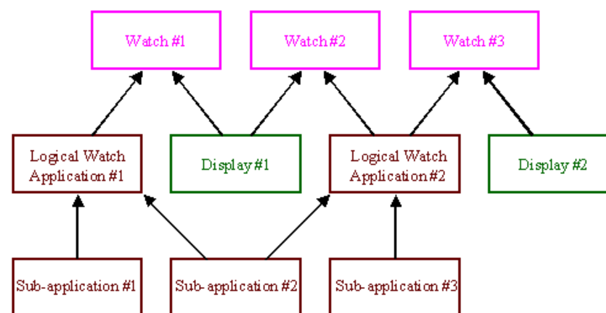

Figure 1-1. The watch configuration

In real life the platform in this kind of case would be an electronic device controlled by a microchip. For an example that anyone could run this is of course not a viable option. Instead, we aimed for a test environment in which we can execute the generated code and test the functionality of a watch, as it would appear in the final product. Such a test environment would in any case be a common element in software development of this kind.

We decided to base the modeling language for the watch applications on state machines, which are a traditional way to model embedded software. We did, however, significantly extend the state machines semantically to achieve better expressive power for our purposes. With these extensions, the finite state machines describe the whole application logic, making it possible to generate 100% of the code for the watch implementation.

The watch modeling 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. The logical watch applications and sub-applications have been described with WatchApplication diagrams that define the state machine implementation for each application. WatchApplication diagrams are actually extended state transition diagrams with customized semantics and domain-specific additions, which enable the use of such domain-specific concepts as buttons and alarms. An example of a WatchApplication diagram is shown in Figure 1-2.

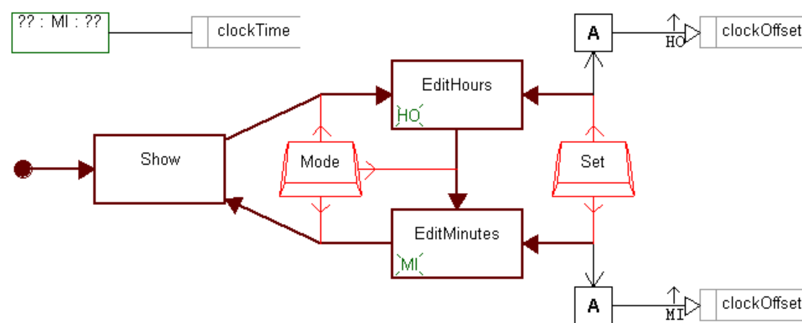

Figure 1-2. An example of a WatchApplication diagram.

As we wanted our test environment to run on any desktop, Java was chosen as the implementation language. A domain-specific framework of Java classes was established in addition to the standard Java runtime. This framework is the same for all watches made in this way, and provides the user interface and the abstract super-classes for displays and state machines. The code generator was then implemented to take advantage of these component, for instance by creating subclasses of them.

For the record, the whole project of designing and implementing the first working version of the Watch modeling 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 modeling language, and one day for the code generator. These times include design, implementation, testing and basic documentation. Since then, new watch models 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 Working with the Watch Example

In this chapter we discuss how to access the watch example and how to work with it, first by playing around with existing watch models, then by creating new models and functionality ourselves and finally by extending the watch modeling language itself.

2.1 ACCESSING THE WATCH EXAMPLE

To access the watch example, start MetaEdit+, login as usual into the demo repository and choose the 'Watch' project from the lists when MetaEdit+ prompts for the projects to open and for the default project. When MetaEdit+ has completed the login procedure, the watch example can be accessed via the usual MetaEdit+ tools like the Graph Browser and the Diagram Editor.

2.2 PLAYING AROUND WITH THE WATCH EXAMPLE

To start the tour of the watch example, open the Graph Browser from the MetaEdit+ main launcher. All models related to the watch example are now shown on the **Graphs** list. Choose the '2004Models' WatchFamily diagram from the list and open it in the Diagram Editor either by double-clicking or by selecting open from the pop-up menu (as shown in Figure 2-1).

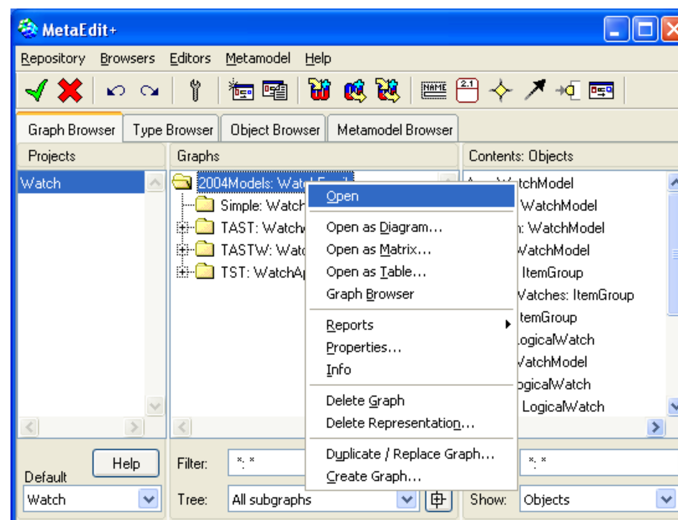

Figure 2-1. Opening a WatchFamily diagram.

The '2004Models' WatchFamily diagram (Figure 2-2) presents a family of related wristwatch models. In the diagram, the actual models are shown in the **Models** group at the top of the diagram, while the logical watch application and display components are presented in the two

groups at the bottom of the diagram. To access the properties of a model element, double-click it or select it and choose **Properties…** from its pop-up menu.

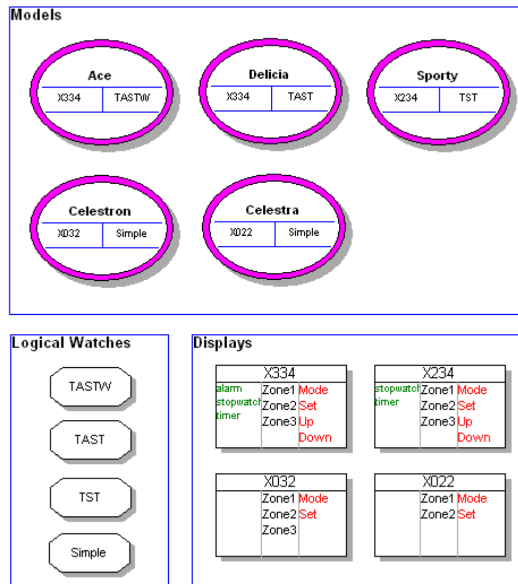

Figure 2-2. The '2004Models' WatchFamily diagram.

To further explore the watch example, let us take a look at how the logical watch applications have been constructed. Choose the one called 'TASTW' from the **Logical Watches** group, select **Decomposition…** from the pop-up menu and **Open** from the following dialog (or, simply double-click 'TASTW' while holding down the Ctrl-key). This will open the 'TASTW' WatchApplication diagram as shown in Figure 2-3 (you could also have opened this diagram from the Graph Browser).

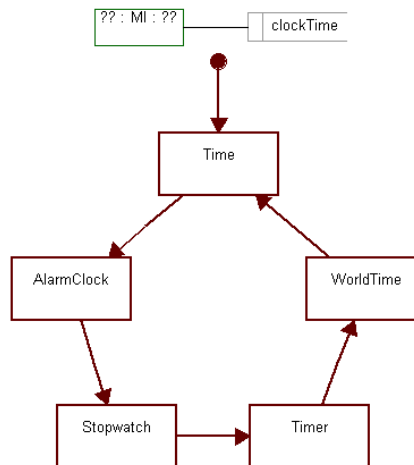

Figure 2-3. The 'TASTW' WatchApplication diagram.

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, a Stopwatch sub-application will

be started. The cycle is completed when the Time will be re-activated when exiting from WorldTime sub-application. The name 'TASTW' comes from these sub-applications' initials in order. To see how the sub-applications have been defined, choose one and open its decomposition graph by selecting **Decompositions…** from its pop-up menu and then **Open** from the following dialog (or, by double-clicking the sub-application element while holding down the Ctrl-key).

Now we are ready to generate the code of the test environment for our watch models. Close all WatchApplication diagrams and go back to the WatchFamily diagram. Before we run the generation, we have to define the target platform for which the code will be generated. Select **Graph | Properties…** in the WatchFamily Diagram Editor and choose the OS you are running on (Windows or Linux) as the **Generation target platform** in the dialog that opens (choices for MIDP and API are covered in chapters 4 and 5). Also make sure that you have the correct version of the J2SE installed in the directory described at the beginning of this document.

To invoke the code generation, select **Graph | Reports | Run…** (or press **Run Report** button in Diagram Editor toolbar) and choose 'Autobuild' from the list that opens. This will execute a report that generates the Java code for all watch models and will compile and start the test environment in a web browser (as shown in Figure 2-4).

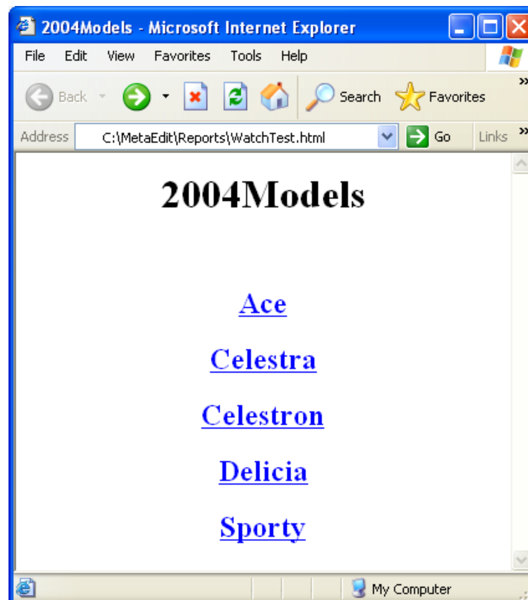

Figure 2-4. The watch test environment.

To test a watch model, choose it from the web page to open its test environment. In the watch test environment (as shown in Figure 2-5) you can test the watch as it would appear in real life by pressing the buttons and observing the behavior of zones and icons on the display. The current application and its state are always shown lower down on the page for debugging purposes.

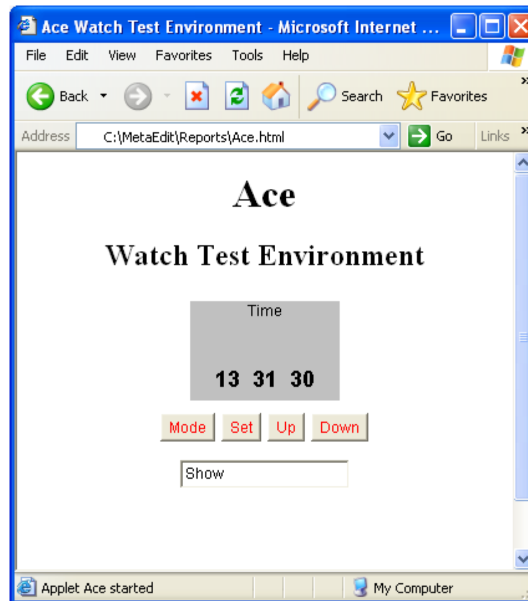

Figure 2-5. The test environment for a watch model.

It is also possible to generate a test environment for just some selected watch models. To do this, open the graph properties dialog for '2004Models' WatchFamily diagram and add the desired watch models into the **Selected models** list by selecting **Add Existing…** from the pop-up menu of the list and choosing the appropriate watch models from the dialog that opens. When the models have been added to the **Selected models** list, run the 'Autobuild' report as explained previously.

Along with the Java code for the Watch test environment it is also possible to generate the technical documentation directly from the Watch models. To try this feature out, run the report called 'Watch family documentation'. The report output will be shown in the web browser with clickable pictures and hypertext links.

2.3 Creating a New Watch Model

The next step in working with the Watch example is to try to develop a new watch model. The easiest way to do this is just simply combine an existing display with an existing logical watch application. However, let us try something more complex here: a stopwatch-only model with four zones and two buttons. This requires the building of a real variant with a new display, as there is currently no display with four zones and two buttons.

First, open the WatchFamily diagram if not opened before and create a new Display object. Enter the name for the display ('X042' in our example) and add four zones, (you can reuse 'Zone1', 'Zone2' and 'Zone3' from the existing models by selecting **Add Existing…** from the popup menu of **UnitZones** list, but you have to create 'Zone4' as it is not available), and two buttons, (you can reuse 'Up' and 'Down' buttons from previous models). The property dialog for the display should now look like Figure 2-6. Choose **OK** and close the dialog.

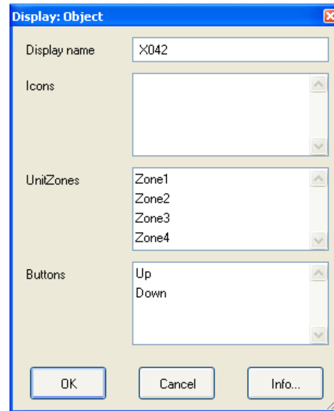

Figure 2-6. Property dialog for the new Display object.

The next component we need is the logical watch application for our stopwatch. We can reuse the existing stopwatch sub-application, but we need to repackage it before we can use it as a logical watch application. In WatchFamily diagram, create a new LogicalWatch object and attach the existing 'Stopwatch' WatchApplication graph as its property. Choose the new LogicalWatch object and select **Decompositions…** from its pop-up menu. Accept WatchApplication as the type of decomposition graph and choose 'Stopwatch' from the following list. The logical watch application has been now defined. Note that now we are using Stopwatch directly from the LogicalWatch, rather than having a top-level WatchApplication state diagram in between.

To finalize our new watch model, we need to assemble our new display and logical watch components as the model. Create a new WatchModel object in WatchFamily diagram. Enter its name ('JustStopwatch' in our example) and attach our newly created display and logical watch application as its respective properties. The WatchFamily diagram should now look like Figure 2-7.

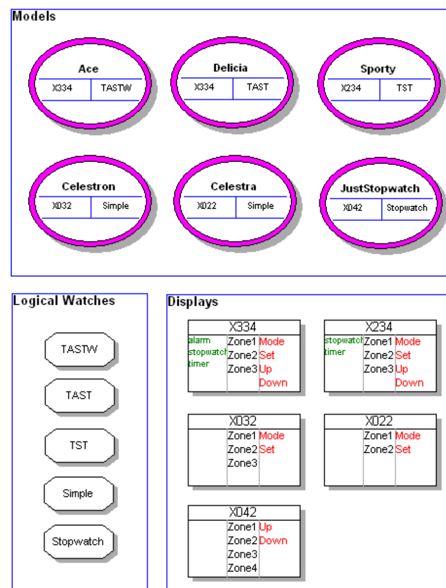

Figure 2-7. The WatchFamily diagram with new objects.

2.4 ADDING FUNCTIONALITY TO A WATCH MODEL

As the final exercise in our Watch example we will build new functionality into an existing sub-application by modifying its state machine definitions. In the Graph Browser, open the 'Stopwatch' WatchApplication diagram. The diagram will appear as shown in Figure 2-8.

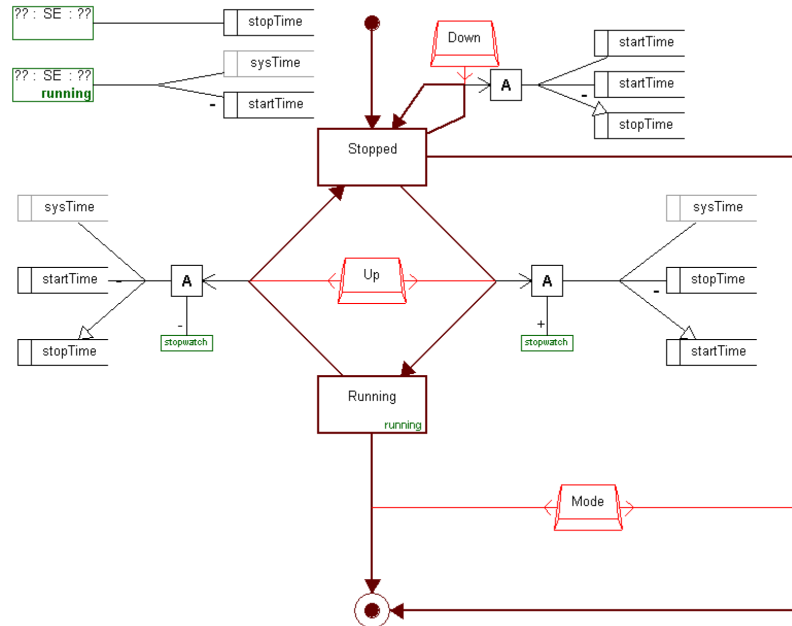

Figure 2-8. The 'Stopwatch' WatchApplication diagram.

As the name says, 'Stopwatch' is a sub-application that enables the user to time the length of various events. When this application is activated, it immediately enters the 'Stopped' state showing the zeroed time counter on the display. From here there are three ways to proceed. Pressing 'Mode' will deactivate the application and pass the control back to the top-level state machine. Pressing 'Down' will reset the counter. Pressing 'Up' will start the counter by setting the start time and entering the 'Running' state. Pressing 'Up' again while in 'Running' state will stop the counter by calculating the stop time and returning back to the 'Stopped' state. It is also possible to terminate the application by pressing 'Mode' while in 'Running' state.

However, there is still something missing from our 'Stopwatch' sub-application: it does not have a lap-time function. Basically, to add such a functionality we need describe that when 'Down' is pressed while the 'Running' state is active, the lap-time is calculated (as it is not available as a pre-defined entity) and a new 'LapTime' state will be activated, showing the lap-time on display. When 'Down' is pressed again, the control is dispatched back to the 'Running' state.

Before proceeding with any new functionality, let us explore the basic mechanism of how displayed time units are controlled. Each application state refers to a display function that specifies how to calculate the time shown while the state is active. Display functions are presented with DisplayFn objects (two green boxes at the top of Figure 2-8) and each of them can be shared by many states. The green text at the bottom right corner of the state symbol indicates which display function it refers to. There are two display functions in our Stopwatch application, one called 'Running' and one without a name. Leaving the name blank is the way

to define a default display function – all the states without an explicitly named display function will use this nameless display function by default. The display function definition also sets the key time unit that will be shown as the middle one on the display. The actual time unit arithmetic is based on variables and variable references that are services built into our Java platform. For example, the display function 'Running' returns the current counter time by subtracting the value of 'startTime' variable from the value of the 'sysTime' variable reference that carries the current system time.

As for the first task of creating the lap-time functionality, create a new State object and enter 'LapTime' as its name. As the default display function suits for our purposes here and as we do need a blinking display, you can leave other property fields blank. Proceed then by defining a Transition relationship from the 'Running' state to 'LapTime' and then another from 'LapTime' back to 'Running'.

The next thing we need to do is to associate the 'Down' button as a triggering event for both of these transitions. We do not need to define a new button as we can reuse an existing button definition. Select the existing 'Down' button, copy it with Ctrl+C, and paste it with Ctrl+V. The pasted button will follow the cursor: move it down to near the 'LapTime' state (Figure 2-9) and click to place it there. Associate this button with both transitions between 'LapTime' and 'Running' states by choosing each relationship, selecting **Add a New Role…** from their popup menus and connecting the new roles to the 'Down' Button. The diagram should now look similar to Figure 2-9.

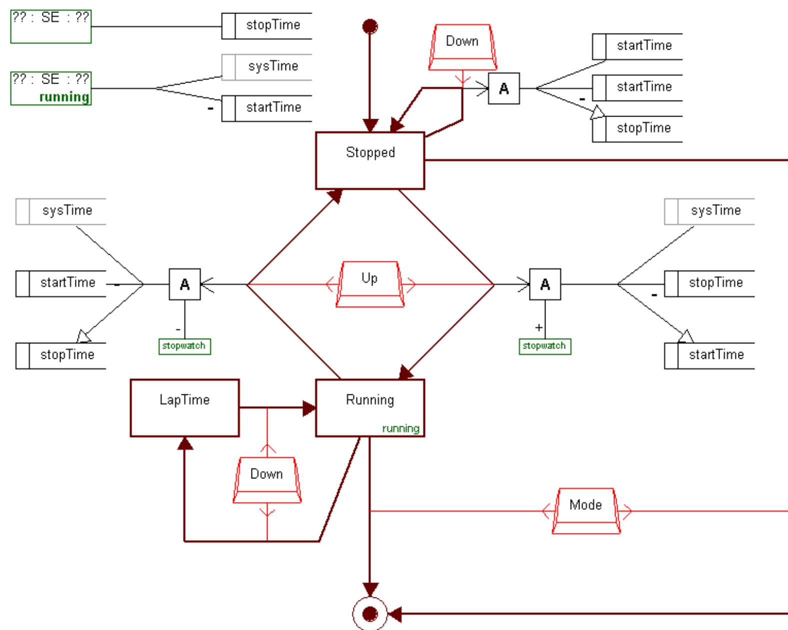

Figure 2-9. The 'Stopwatch' diagram with definitions for a new state.

To complete the implementation of the new functionality, we need to define the actions that are required for calculating the lap-time during the transition from the 'Running' state to the 'LapTime' state. First create a new Action object, and add a new role to it from the Transition relationship going from 'Running' to 'LapTime'. Then reuse the sysTime VariableRef object and startTime and stopTime Variable objects from this same graph as shown in Figure 2-10.

Figure 2-10. The extended version of 'Stopwatch' sub-application.

To define the calculation relationship between Action, VariableRef and Variable objects, hold shift down and select first the Action object, then 'sysTime', then 'startTime' and finally 'stopTime' and then select **Connect** from the popup menu. From the list of possible relationship combinations, choose 'Set (ActionBody Action) (Get sysTime) (Minus startTime) (Set stopTime)' and accept the following dialog as it is. The definition of lap-time functionality has been now completed. The lap-time will be shown correctly as our calculation stores the lap-time value in the 'stopTime' variable that is attached to the default display function used by the 'LapTime' state. You can now try it out by generating the code and running the test environment.

2.5 EXTENDING THE WATCH MODELING LANGUAGE

What about the future, then? It is obvious that in real life a modeling language like this continues to evolve, both in and of itself, and to keep pace with changes in the domain or platform. Thus it is important to be able to develop the DSML further according to the new requirements that may arise. The way we developed the Watch modeling language enables us to add new features to the language relatively easily by extending the modeling language definitions, code generators and pre-defined Java classes. As an example on how to extend the Watch modeling language, let us add a new Constant object type to be used in WatchApplication diagrams and code generation support for it.

To add a new object type, open Object Tool from the MetaEdit+ launcher and define a new object called 'Constant' with properties 'Hours', 'Minutes' and 'Seconds' (make sure to choose number as a data type for each of these properties). With these definitions, the Object Tool should look like in Figure 2-11.

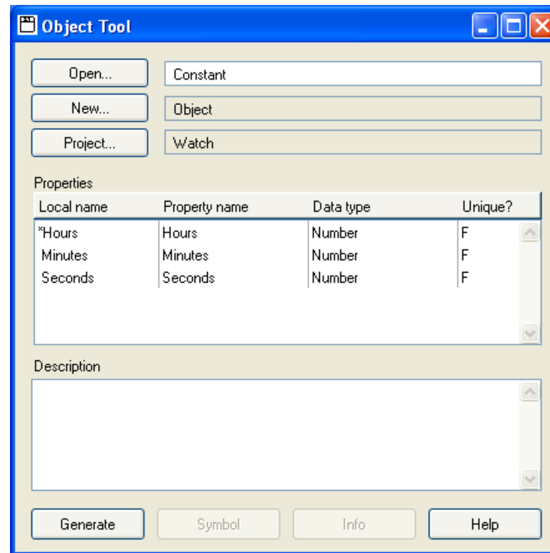

Figure 2-11. The Object Tool with 'Constant' definitions.

Confirm the creation of the ‘Constant’ type by pressing **Generate** button. To complete the definition of new object type, we need to still create a symbol for it. Launch the Symbol Editor by pressing the **Symbol** button on the Object Tool. Create a symbol similar to the one shown on Figure 2-12 (you can also copy an existing symbol like the one used for ‘Variable’ and modify it for this purpose). Remember to save the symbol when leaving the Symbol Editor.

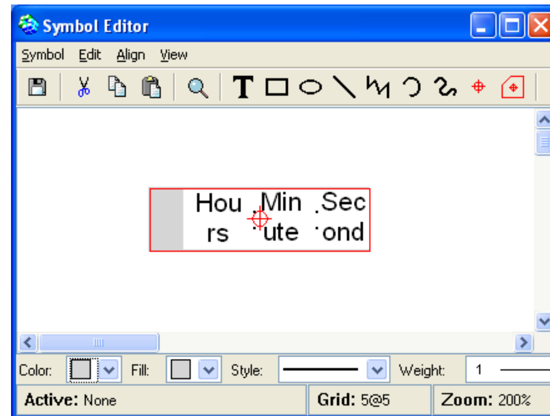

Figure 2-12. The symbol for 'Constant' object type.

The next thing to do is to integrate this new type into our existing WatchApplication diagram type. Open the Graph Tool from MetaEdit+ launcher and retrieve the definitions for WatchApplication graph type. Press the **Types** button and add ‘Constant’ to the **Objects** list in the Graph types definer dialog that opens. Close the Graph types definer and press the **Bindings** button in the Graph Tool to open the Graph bindings definer. In the Graph bindings definer, connect the ‘Constant’ object type with ‘Get’, ‘Minus’ and ‘Plus’ roles in the bindings of the ‘Alarm’ and ‘Set’ relationships (an example of this kind of binding is shown in Figure 2-13). Now you can create and use Constant objects in your models.

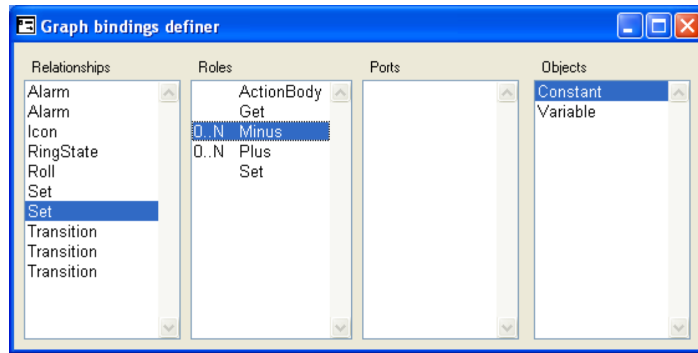

Figure 2-13. A binding with ‘Constant’ connected to ‘Minus’ role.

The final touch to complete the addition of this new type into our modeling language is to get the code generator to support it. Open the Report Browser for the WatchApplication graph type and modify the ‘_calcValue’ report as follows:

```
Report '_calcValue'
do ~Get.()
{  if type = 'Constant'
   then '(new METime('; :Hours; ', ';
     :Minutes; ', '; :Seconds; '))';
   else 'get'; id; '()';
   endif;
}
do ~(Minus|Plus)
{  '.me'; type;
   do .()
   {  if type = 'Constant'
      then '(new METime('; :Hours; ', ';
        :Minutes; ', '; :Seconds; '))';
      else '(get'; id; '())';
      endif;
   }
};
endreport
```

Accept the changes by saving the report. The new ‘Constant’ type has been now integrated as a fully operational part of the Watch modeling language. To try out how it works, consider the fragment of the ‘Stopwatch’ WatchApplication diagram shown in Figure 2-14:

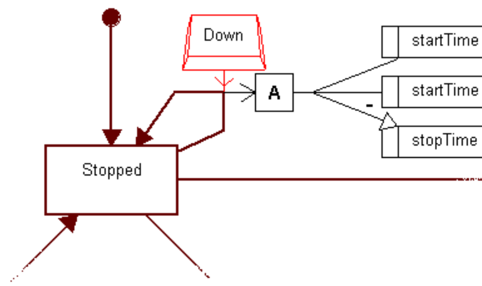

Figure 2-14. The original ‘Stopwatch’ diagram.

The implementation of resetting the stopwatch is obviously somewhat weak: to get the zeroed counter one has to subtract the value of the ‘startTime’ variable from itself and store it a value

for ‘stopTime’ variable. With the new ‘Constant’ type we can define this more naturally by just assigning a constant value of zero to the ‘stopTime’ variable, as illustrated in Figure 2-15:

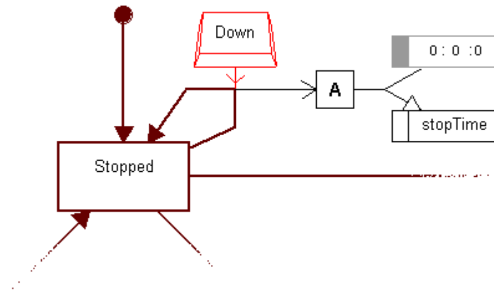

Figure 2-15. The new version of ‘Stopwatch’ diagram.

The issue of future extensions should be recognized and taken care of during the original design and implementation of a domain-specific language. Ease of extension was not our primary goal while implementing the Watch modeling language, but there are still certain ‘hooks’ for this purpose. The main area we felt extensions would be necessary were the different operations possible in actions. For this reason we decided to make each operation type have its own relationship type. This allowed us to easily make bindings and constraints specifying the rules of what arguments the operation can or must have. Similarly, the code generation for that kind of operation is easy to add as a new report named after the relationship type. Together, these make it very easy to extend the modeling language with new operations.

More complex extensions usually require further additions and modifications within the modeling language, code generator and in the framework classes. However, as each of these have been implemented in a modular fashion, making the required modifications should be relatively easy. Most importantly, only one person need make the required changes, and all of the modeling language users will benefit from them.

3 Exploring the Watch Modeling Language Further

We have been now walking through the basic tasks of working with the Watch modeling language. You are however welcomed to explore the Watch example further. To make this easier, we will go through a more detailed description on the software architecture behind the Watch example.

3.1 The Watch Architecture

The architecture for a DSM environment like our watch example generally consists of three parts: a modeling 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 Figure 3-1.

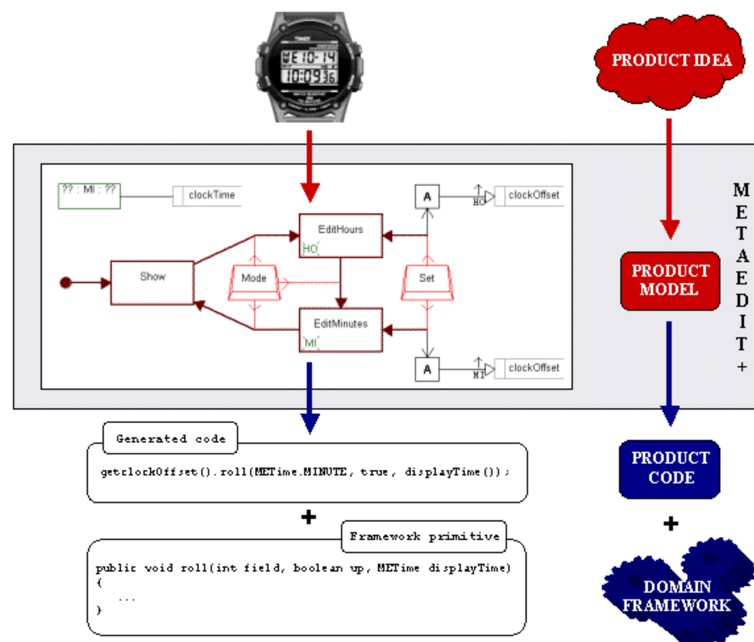

Figure 3-1. The watch architecture

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. In order to proceed along with this approach, we had to device a rough division of responsibilities first. As we assumed the code generator to be the most complex part of our environment, we decided to keep it as simple and straightforward as possible. This decision then laid the

foundation for the roles of the modeling language and the domain framework. The modeling language was assigned to capture the behavioral 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.

As we can see, the basic ideas behind the watch architecture are reasonably simple. In the next few chapters we will provide a more detailed description of what kind of solutions were required on each level and how they have been implemented.

3.2 THE MODELING LANGUAGE

The single most important asset of a DSM environment is the domain-specific modeling language. To substantiate this claim, let us consider the payback promised by the DSM approach. The labor of the domain experts who create the DSM environment is the original investment on the DSM. The return of investment, then, is a result of the increased productivity of the developers working with the DSM environment. While the code generator and the domain framework are important parts of the DSM environment, they still remain reasonably invisible for the developer where as the role of the modeling language is emphasized as the main instrument of the DSM. Thus, the better modeling language we have the more we benefit from the DSM.

The first rule of thumb for creating a domain-specific modeling language is to keep the language as independent as possible from the target code. We know that it often appears originally easier to try to build the modeling language as an extension on top of the existing code base or platform, but this kind of visualization of the code world very seldom lead to a significant rise in the level of abstraction and it definitely does not free the developer to think according to the domain instead of the code. Thus, to achieve the best possible level of abstraction for our modeling language, we need to base it on the domain itself.

First, ask yourself what do you want to do with the domain-specific modeling language. When you have set this mission statement for yourself, you can proceed by asking how can you do what you want and what is needed to do that. This analysis leads to the discovering of the first domain concepts that are to be incorporated into the modeling language. For example, our mission statement for the watch modeling language set the following goal: we want to be able to model those static and behavioral elements that constitute a digital wristwatch. Based on this statement, the further analysis soon identified a watch model, display, logical watch and a watch application as the elementary concepts of the watch architecture.

At this early stage of the modeling language definition, one should focus on identification and definition of domain concepts. There are several ways to do this and it is important to understand that usually none of them alone can provide a complete coverage. Good results typically require concurrent use of a number of various strategies. In any case, the key success factor in finding the domain concepts is the domain knowledge possessed by the domain experts.

A good way to identify the domain concepts is to study the aggregation structures within the domain and product. The aforementioned elementary watch concepts were discovered this way. Another method is to try to find the commonality and variability among the products. For example, it is easy to see that the different watch models still share the common concept of button that would suggest button as a probable domain concept for the modeling language. Similarly, we can see that the watch models differ from each other by the number of buttons

and features they include, which says something about the configuration space requirement for the watch models.

It is also worthwhile to examine the existing specifications to see if there are any repeating patterns that could be refined further as a domain concept. And don't forget the common wisdom or folklore as a source for ideas – our decision to use the state machine as the computational foundation for watch applications and logical watches was based on the fact that it is a widely used solution in the area of embedded systems in general.

The next step after the identification of domain concepts is to assemble them as a modeling language. The language is defined with MetaEdit+ as a metamodel that applies the domain concepts as the semantic structures for the language. The detailed description of the metamodelling process is out of the scope of this tutorial, so for more comprehensive information about the metamodelling please refer to the Family Tree example in the Evaluation tutorial and 'MetaEdit Method Workbench: User's Guide'.

Two important aspects should be considered during the creation of the modeling language: layering and reuse. Modeling language development efforts typically start with a flat model structure that has all concepts arranged on the same level without any serious attempts to reuse them. However, as the complexity of the model increases while the number of elements increases, the flat models very seldom are suitable for presenting hierarchical and modularized software products. Thus, we need to be able to present our models in layered fashion.

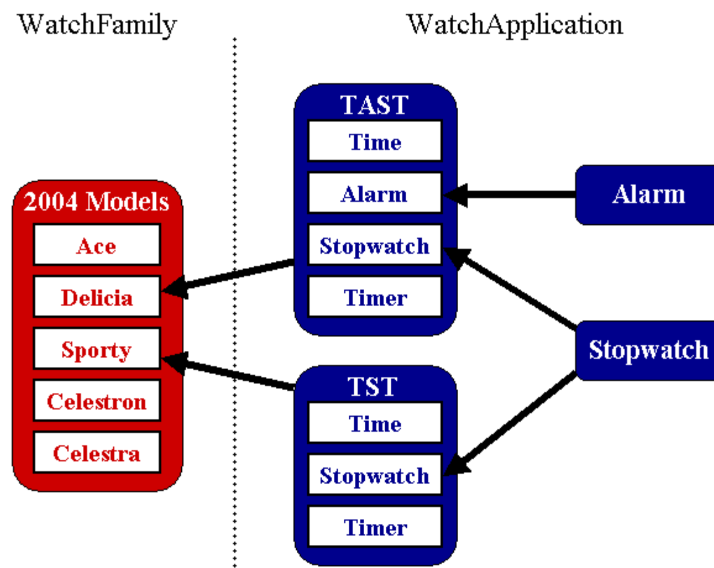

Figure 3-2. Watch model hierarchy

A partial illustration of watch model hierarchy is presented in Figure 3-2. The watch model hierarchy consists of three layers. For the top-layer models we use a proprietary WatchFamily diagram type to present the static high-level configuration of each watch model. On the two subsequent layers we employ the WatchApplication diagram type to present the dynamic variability within the logical watches and watch applications. As any additional diagram type increases the complexity of the modeling language, it is recommended to design diagram types recursive manner so that they can be applied on more than just one layer. On the other hand, if the semantics of two consequent layers are different, two different diagram types are the recommended solution.

Another aspect that influences the layer structure is reuse. The idea of reuse-based layering is to treat each model as a reusable component for the models on the higher level. In this kind of solution, the reusable element has a canonical definition that is stored somewhere and referenced from where needed. In the watch example, the whole layering is actually based on this principle. Each watch application is a true model component and can be referenced from any logical watch. Similarly, each logical watch can be attached to any concrete watch model.

Reuse can also happen regardless or in addition to the layering. Typically we want also reuse the concepts within or between the models. The concept of a display function is an example of reuse within a model. Defined once in a WatchApplication diagram, each display function can be referenced by any state within the same diagram. Reuse between the models is needed when the same concept appears on several models. Examples of this kind of concept within the watch example are the buttons. While it is possible to create a new button concept for each state transition, it is much better idea to reuse an existing one. It is not only faster to create button this way, but it also makes it easier to propagate the possible future changes (like changing the name of the button). This kind of reuse differs from those described above in the sense that there is no canonical definition for the reusable element. It can be considered as a floating concept that exists as long as it is instantiated somewhere.

The last step during the initial development of a domain-specific modeling language is the finalization of the language. Typically this means the extension of the language with concepts that are required for code generation, component interfaces, consistency checking and documentation. In watch example, the inclusion of the watch domain framework code as included components in WatchFamily diagram is an example of this kind of late addition. With this solution we wanted to ensure that the framework code is always available for the code generation. Otherwise, and probably a bit surprisingly, no other code or documentation generation related additions or modifications to the original modeling language were needed.

3.3 THE CODE GENERATOR

The basic idea of the code generator within a DSM environment is simple: it crawls through the models, extracts information from them and translates it as the 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 generated.** All framework code has been included into the watch models and generated 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.

How the code generator has been implemented, then? As we have seen before, the code generators in MetaEdit+ are defined with a dedicated 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 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 Figure 3-3 (a '*' in the name of a sub-generator denotes an individual version for each target platform).

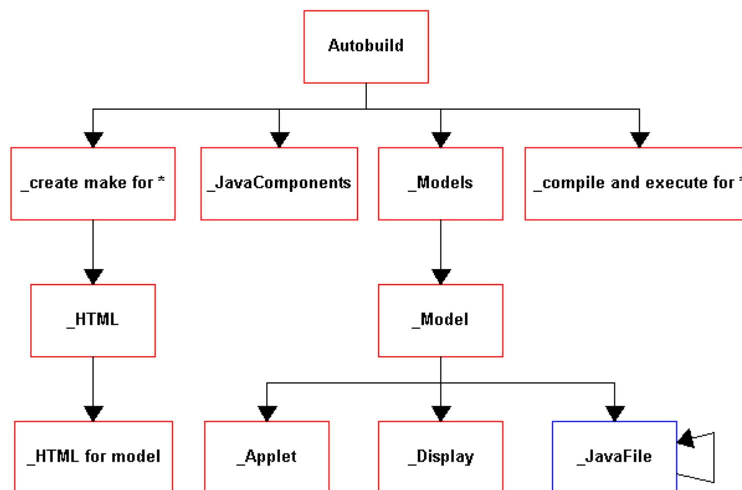

Figure 3-3. The watch code generator architecture, part 1

On the top, there is a master generator called 'Autobuild'. The role of the 'Autobuild' is similar to those of the main programs in most programming languages: it initiates the whole generation process but basically does not contain anything else but the calls to 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 chapter. 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 more on the more complex sub-generator, '_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 Figure 3-3, they can be integrated with the '_create make for *' sub-generator.

The responsibility of the '_Models' and '_Model' sub-generators is to master 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.

An example of the code generated for an applet (produced by the '_Applet' sub-generator) is shown in Listing 1. The generated code defines the applet as an extension of the AbstractWatchApplet and adds the initialization for the new class.

```
public class Venturer extends
AbstractWatchApplet
{
  public Venturer()
  {
    master=new Master();
    master.init(this, new DisplayX334(),
      new TASTW(master));
  }
}
```

Listing 1. Generated code for an applet

The display definition can be generated in the same vein by the sub-generator '_Display', as shown in Listing 2. Again, a new concrete display class is inherited from the AbstractDisplay and the required user interface components are defined within the class constructor method.

```
public class DisplayX334 extends
AbstractDisplay
{
  public DisplayX334()
  {
    icons.addElement(new Icon("alarm"));
    icons.addElement(new Icon("stopwatch"));
    icons.addElement(new Icon("timer"));

    times.addElement(new Zone("Zone1"));
    times.addElement(new Zone("Zone2"));
    times.addElement(new Zone("Zone3"));

    buttons.addElement("Mode");
    buttons.addElement("Set");
    buttons.addElement("Up");
    buttons.addElement("Down");
  }
}
```

Listing 2. Generated code for a display

However, 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 WatchApplication graph type) is presented in Figure 3-4.

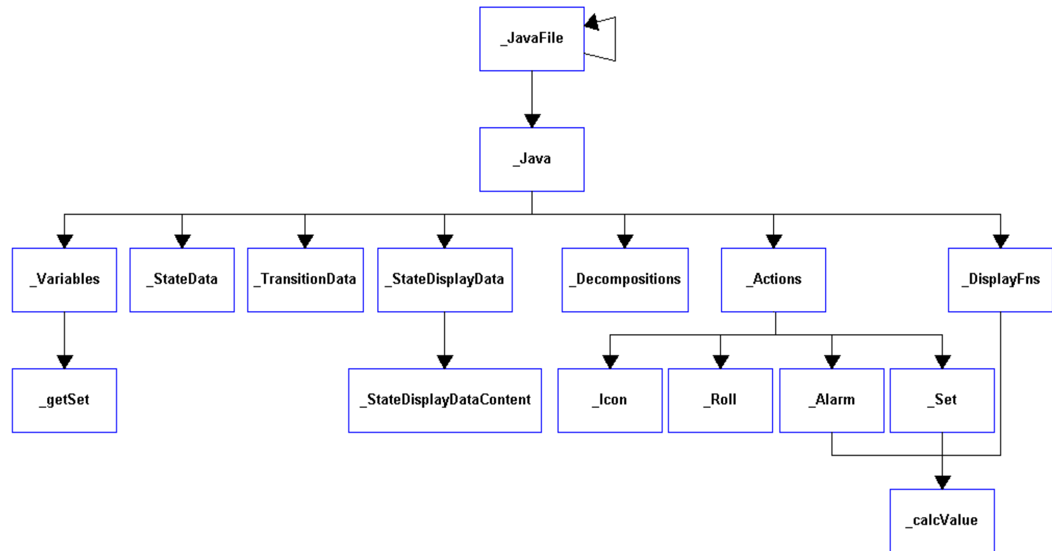

Figure 3-4. The watch code generator architecture, part 2

The sub-generators '_JavaFile' (which is same as in Figure 3-3) 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 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.

The task of the '_Java' sub-generator is to generate the final Java implementation for the logical watch and watch application state machines. An example of such code can be found in Listing 3 that presents the implementation of the Stopwatch application.

```
 1  public class Stopwatch extends AbstractWatchApplication
 2  {
 3    static final int a22_3324  = +1; //+1+1+1+1
 4    static final int a22_3621  = +1+1; //+1+1+1
 5    static final int a22_4857  = +1+1+1; //+1+1
 6    static final int d22_4302  = +1+1+1+1; //+1
 7    static final int d22_5403  = +1+1+1+1+1; //
 8
 9    public METime startTime = new METime();
10    public METime stopTime = new METime();
11
12    public METime getstartTime()
13    {
14      return startTime;
15    }
16
17    public void setstartTime(METime t1)
18    {
19      startTime = t1;
20    }
21
22    public METime getstopTime()
23    {
24      return stopTime;
25    }
26
27    public void setstopTime(METime t1)
28    {
29      stopTime = t1;
30    }
31
32    public Stopwatch(Master master)
33    {
34      super(master, "22_1039");
35      addStateOop("Start [Watch]", "22_4743");
36      addStateOop("Running", "22_2650");
37      addStateOop("Stopped", "22_5338");
38      addStateOop("Stop [Watch]", "22_4800");
39
40      addTransition ("Stopped", "Down", a22_3324, "Stopped");
41      addTransition ("Running", "Up", a22_4857, "Stopped");
42      addTransition ("Stopped", "Up", a22_3621, "Running");
43      addTransition ("Stopped", "Mode", 0, "Stop [Watch]");
44      addTransition ("Running", "Mode", 0, "Stop [Watch]");
45      addTransition ("Start [Watch]", "", 0, "Stopped");
46
47      addStateDisplay("Running", -1, METime.SECOND, d22_5403);
48      addStateDisplay("Stopped", -1, METime.SECOND, d22_4302);
49    }
50
51    public Object perform(int methodId)
52    {
53      switch (methodId)
54      {
55        case a22_3324:
56          setstopTime(getstartTime().meMinus(getstartTime()));
57          return null;
58        case a22_3621:
59          setstartTime(getsysTime().meMinus(getstopTime()));
60          iconOn("stopwatch");
61          return null;
62        case a22_4857:
63          setstopTime(getsysTime().meMinus(getstartTime()));
64          iconOff("stopwatch");
```

```
65          return null;
66        case d22_4302:
67          return getstopTime();
68        case d22_5403:
69          return getsysTime().meMinus(getstartTime());
70      }
71      return null;
72    }
73  }
```

Listing 3. The generated code for Stopwatch application

For the comprehensive understanding of the '_Java' sub-generator output, let us study the generated code line by line. As previously, a new concrete watch application class is derived from the AbstractWatchApplication class (line 1). From here on, the responsibility for the generated code is distributed among the '_Variables', '_StateData', '_TransitionData', '_StateDisplayData', '_Actions' and '_DisplayFns' sub-generators.

The '_Variables' and '_getSet' sub-generators are responsible for declaring the identifiers for actions and display functions to be used later within the switch-case structure (lines 3 – 7). They also define the variables used (lines 9 – 10) and the implementations of their accessing methods (lines 12 – 30). A short return back to the '_Java' sub-generator produces the lines 32 – 34, followed by the state (lines 35 - 38) and state transition definitions (lines 40 – 45) generated by the '_StateData' and '_TransitionData' sub-generators. The '_StateDisplayData' and '_StateDisplayDataContent' sub-generators then provide the display function definitions (lines 47 – 48) while the basic method definition and opening of the switch statement in the lines 51 – 54 again come from the '_Java' sub-generator.

The generation of the code for the actions triggered during the state transitions (lines 55 – 65) is a good example of how to creatively integrate the code generator and the modeling language. On the modeling language level, each action is modeled with a relationship type of its own. When the code for them is generated, the '_Actions' sub-generator first provides the master structure for each action definition and then executes the sub-generator bearing the same name as the current action relationship (either '_Icon', '_Roll', '_Alarm' or '_Set'). This implementation not only reduces the generator complexity but provides also a flexible way to extend the watch modeling language later if new kinds of actions are needed.

Finally, the '_DisplayFns' and '_calcValue' sub-generators produce the calculations required by the display functions (lines 66 – 69). The '_calcValue' – which is used also by the '_Alarm' and '_Set' sub-generators – provides the basic template for all arithmetic operations within the code generator.

The generation of a logical watch proceeds the same way. As there are typically no actions related to the state transitions within a logical watch, they are left off from the generated code as well. More over, in order to support the references to the lower-level state machines (i.e. to the watch applications the logical watch is made of), the definitions of these decomposition structures must be generated. This is taken care by the '_Decompositions' sub-generator.

As we can see, there is no magic within the code generator, just a carefully designed modularization of the solution and integration with the modeling language and domain framework. In general, the rule is to try to keep generation as simple as possible: if anything appears difficult, consider raising it up into the modeling language, or pushing it down into the domain framework code.

Having covered the modeling language and the code generator, we will next discuss issues related to the domain framework code.

3.4 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.

In many cases the demarcation between the platform and the domain-specific part of the framework remains unclear. For example, the version of Java that the watch example was originally written in did not carry any useful service to handle timed events like alarms. Thus, we implemented such a service ourselves as a part of our domain framework. The more recent versions of Java, however, now do provide a similar mechanism, meaning that it could be part of the platform if the watch implementation only needed to support more recent versions of Java.

Without any deep theoretical discussion about what is the border between framework and platform, we shall use the following definitions here: The platform is considered to include the hardware, operating system (Windows or Linux), Java programming language with AWT classes and environment to test our generated code (either browser or MIDP emulator). The domain framework consists of any additional components or code that is required to support code generation on top of this platform. The architecture of the watch domain framework – as defined this way – is presented in Figure 3-5 (solid line arrows indicate the instantiation relationship while dotted line arrows indicate inclusion relationships between the elements).

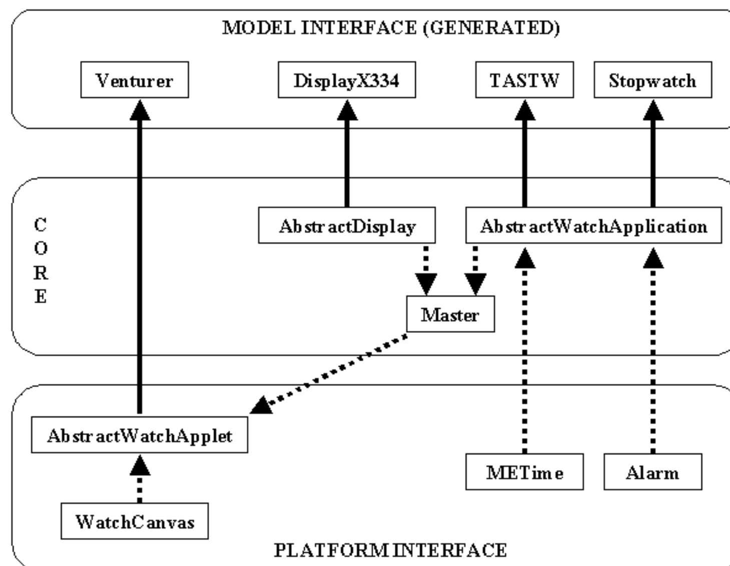

Figure 3-5. The watch domain framework

The domain architecture of the watch example consists of three levels. On the lowest level we have those 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 watch models in the form of abstract superclass 'templates'. The top level then 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 the other 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.

Typically a framework is mainly composed of in-house components either already available from previous projects, or specifically made for this purpose.

3.5 THE WATCH ARCHITECTURE REVISITED

As the watch DSM environment was created purely as an example, it is clear that in its current form it does not form a perfect vehicle for a true real-life development effort. The main problem comes from having to support different operating systems for the build scripts. This necessitates a property to select the build platform. Unfortunately, the property selection is saved along with other properties, and thus would not work well in a multi-user environment. As the property is changed by each user, on different platforms, that property becomes locked and other users cannot change it — and hence can only perform a build if they happen to be on the same platform.

A better solution, but one which would have been overkill for this example as a single-user tutorial, would be to have another kind of graph to specify build properties. The graph could consist of just properties, for instance a name ‘Linux watch build’, the Generation target platform selection ‘Linux’, and a property pointing to the previous top-level graph, 2004Models. There could be a number of such graphs for 2004Models, one for Linux, one for Windows etc.

Another possibility, allowing still better multi-user interaction, would be to make the report-running graph yet more generic by adding a property containing the name of the report to run, and making the property data type of the graph reference be just Graph. This would allow each user to make their own project with such ‘build command’ graphs for a variety of uses.

Of course, the real situation in the organization should be the deciding factor in designing such solutions. A little creative thought about using reports, graphs and projects can often pay good dividends in terms of smooth co-operative development.

4 The Watch Example for MIDP

Mobile Information Device Profile (MIDP) 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 over the air (i.e. as wireless data).

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 modeling 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 metamodeler'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.

First though we will set up and run the MIDP example, then return in Section 4.3 to look at what changes had been made to the original applet-based Watch to support MIDP.

4.1 MIDP DEVELOPMENT ENVIRONMENT

MIDP applications can be developed on all major platforms, but for the sake of simplicity we only provide instructions for Windows. If possible, install all components into the directories suggested here, then you will not need to edit the paths in the MetaEdit+ reports that generate build scripts.

The required components for trying out Watch applications in MIDP are:

- Java 2 Platform, Standard Edition (J2SE) version1.4.2 or later. This was already discussed in Preface and should be installed in `c:\java\j2sdk`.
- Java 2 Platform, Micro Edition (J2ME) Wireless Toolkit, version 2.0 or higher, that can be downloaded from java.sun.com and should be installed in `c:\java\j2mewtk`.

J2ME Wireless toolkit comes with a MIDP device emulator that is suitable for our purposes, but if you want to use better looking emulators with more bells and whistles, we advice you to take a look at and download MIDP emulators for various Nokia mobile phones from www.forum.nokia.com (requires registration). The following Nokia MIDP SDKs with emulators have been integrated with Watch example (the recommended installation directory is given within parenthesis):

- Nokia 3510i MIDP SDK 1.0 (`c:\java\nokia\devices\Nokia_3510i_MIDP_SDK_v1_0`)
- Nokia 7210 MIDP SDK 1.0 (`c:\java\nokia\devices\Nokia_7210_MIDP_SDK_v1_0`)

If you have installed all components into their recommended directories, you can skip straight to Section 4.2. Otherwise you need to change the paths that are marked in bold in the '_create make for MIDP' report in the WatchFamily graph type.

If you are about to use Nokia emulators, you want to toggle the use of advanced alarm behavior with sound and vibration. The '2004Models' graph properties contain a list of Included Components. Double-click the second AbstractWatchApplet entry in the list: it should open and show AbstractWatchApplet for MIDP. Two include lines are commented out near the start; uncomment them. At the end, uncomment a block of ten lines of code. OK both dialogs to accept the changes.

4.2 Building and Running MIDP Watch

Once you have installed all the above components, open the Watch example project in MetaEdit+. Edit the properties of the '2004Models' graph, setting the **Generation target platform** to MIDP and choosing the emulator from the **MIDP Emulator** list.

You can then run the Autobuild report for that graph. This should generate the Java code and MIDP configuration files, and a batch file which it will then run. The batch file will compile the code, verify it for MIDP, package it into a .jar file, fill out extra parts of the configuration files (e.g. the size of the .jar file), and make a .jad file. This is the application descriptor file, which is a simple textual description of the .jar file and application. The batch file will then start the emulator with the parameters from that .jad file.

If all goes well, you will be presented with a phone showing a list of the Watch models. Use up and down keys on the phone to highlight one, and then press Select to start it. Once in a watch model, you can cycle through the sub-applications there with the Mode button, and interact with them using Mode, Set (normally shown as on-screen soft-keys), Up and Down (mapped to the up and down keys or 2 and 8 on the phone).

4.3 A Little History

The original Watch example was designed to run in an applet in a browser on Windows, and was soon extended to support Linux. This thus covered the two most common MetaEdit+ platforms. Whilst MetaEdit+ does not yet run on MIDP(!), the original idea of a watch application appeared suitable to the platform. This would also provide us with an example of a real embedded application, along with the accompanying limits on memory, speed, power etc.

Thus we began looking at MIDP, and extending the Watch example to support it. In the meantime, and completely coincidentally, Nokia released its first generation of MIDP phones, which included as preloaded applications a Stopwatch and a World Time display.

The main hurdle to supporting MIDP was simply MIDP itself: the subject was still new, developing quickly, and the documents and software to support it were Spartan and hard to use. All the more reason to have a simpler solution for developers: our fictitious watch developers could carry on designing watches as before, without needing to know anything about MIDP.

4.4 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.

One such problem was an area that we decided not to support initially, because of the lack of use: how should an alarm react, when the clock time is changed. We had build metamodel support for this (alarms have a property to say whether they are dependent on local time: true for an alarm clock, false for a countdown timer), but had left out the implementation in the framework classes as being not worth the effort. “After all, who is likely to lug a PC around to use our watch application’s alarm?” we thought. How short-sighted we were!

4.4.1 Metamodels

No changes were required to the metamodels to support MIDP. In the course of improving the support for alarms in the context of a user changing the time on the watch, it was noticed that the existing metamodel allowed Roll roles to VariableRef objects: i.e. changing the value of a function result or variable from somewhere else. This was corrected with a constraint that Roll could only apply to a Variable (local to this graph), not a VariableRef.

An addition was made to the top-level graph for the included components (framework classes), to allow different framework classes for different ‘Generation target platforms’.

4.4.2 Models

No changes were required to the models to support MIDP. To support the improved alarm handling, we found we had to buffer the clockOffset VariableRef in the Time application, editing the buffered copy and then setting it back into clockOffset when the user exits the edit states. This improves the behavior too: otherwise all alarms would be updated on each Roll up or down of a digit, causing alarms to ring during editing.

4.4.3 Reports

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 initialized static final variables as labels.

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

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

These changes were all largely cosmetic, and only the generation of the initialization values for the static final variables required a little ingenuity, as Java limits these to being integer expressions which do not refer to any other variable – not even another static final.

A larger amount of work was required for the new '_create make for MIDP' report. MIDP compiles its Java as for other platforms, but it also requires a pre-verify 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 principle all the information needed can be obtained from the code (or in this case, the models).

The overly-tight constraints on these configuration files' formats make generating them with only the reporting language and DOS batch commands something of a challenge. Having to support several Windows versions, each with different DOS commands and behavior, hardly made the task any easier. In the end it was accomplished with a couple of helper batch files, which generate sequential numbers and report the size of a specified file.

4.4.4 Framework code

The original code was much in need of refactoring, having been the authors' first Java application, and not really intended to be looked at. First we refactored out the mass of user-interface, control and state machine behavior from the applet into the classes of their own. 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. As a result, some behavior is still duplicated between the two platforms' versions of that class, but not enough to merit refactoring it out into its own class.

MIDP does not have the Applet class, so our Applet was replaced with a Midlet, the MIDP equivalent. There appears to be no reason why an Applet should not have been used for MIDP: the same functions are present, just with different names. 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 necessary to the report that generates the applet/midlet for each WatchApplication.

4.5 DOMAIN-SPECIFIC MODELING FOR PLATFORM INDEPENDENCE

The results of this exercise are clear: domain-specific modeling has provided a near-perfect vehicle for insulating applications 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 to one target. Compare that with the normal scenario, where the majority of developers would have to update the majority of their features for each platform change.

In particular, think about what it means in terms of supporting a family of products across a family of platforms. In the normal scenario mentioned above, there would quickly be no hope of maintaining one code base for all platforms. The current Watch models, however, are capable of generating code for each of the three platforms, and on a variety of operating systems.

These advantages are over and above those which come from making products by visual domain-specific modeling instead of writing textual code — a change which normally increases productivity by 5 to 10 times.

5 Building a Visual Debugger for the Watch Example with the API

One of the most common questions regarding the DSM approach is how to track down bugs in the generated code. Conventional debugging strategies may be handy when we are building the framework and code generator for DSM, but for the developer who is working with the DSM environment, the game of debugging is indeed a fair bit different than with traditional IDEs. Let us first consider these differences:

- **No manual changes to generated code.** The benefits of domain-specific modeling are mostly due to the 100% code generation, and working only on the design level. Bug corrections are thus not made to the final code, but to the models instead.
- **Errors originate from design, not from code.** As code is generated automatically and always with the same rules, there are no typical code level errors like typos or syntactical mistakes. This means that for normal developers, errors originate from the design level (the metamodeler will correct errors in the code generators).

Both these factors lead to the conclusion that with DSM, the debugging must be done on design models. This is without a doubt a remarkable benefit: traditionally the developer had to separately update design models after making the code corrections, now it is possible to make corrections in one place only.

When debugging on the model level, the important question is how to find the faulty part of the model? Our original test environment for watch applications provided a very limited solution for this problem by providing information about the current application state at the run-time. A nice option, but it would be much better if we could somehow animate the execution of our state models at the same time as we are running them within the test environment.

This chapter explains how this kind of visual trace or debugging aid can be built for your DSM environment by employing the API facilities of MetaEdit+. We will first have a brief look at the MetaEdit+ API in general and what additional components are required for its use. Then we will learn how to make calls back to MetaEdit+ from our generated code and to play around with the visual debugger.

5.1 Getting Started with the MetaEdit+ API

The API in MetaEdit+ consists of a set of commands that enable the user to access and change conceptual elements (Graphs, Objects, Relationships, Roles, Properties and Ports) within MetaEdit+. It is not possible, however, to access or change representational or metamodel elements (for example, you cannot change diagram layouts with API). There are also some additional features only usable with the API, like animating diagram elements for simulation or tracing purposes (we will exploit this feature for our visual debugger for the Watch example).

The API interface is implemented as a SOAP Web Service server in MetaEdit+. Thus, an application interfacing with MetaEdit+ API must implement a SOAP client that takes care of establishing the connection and making calls to MetaEdit+. As SOAP is a widely supported and open standard, this makes MetaEdit+ functions accessible with almost any programming language, platform or environment.

Before we can proceed with our visual debugger example, we must install the SOAP components for out Java environment. The following two components are needed (both of them can be downloaded from xml.apache.org):

- The Apache Xerces2 XML parser for Java, version 2.5.0 (recommended to be installed in `c:\java\xerces-2_5_0`).
- The Apache Axis SOAP library, version 1.1 (recommended to be installed in `c:\java\axis-1_1`).

Again, the recommended installation directories are the ones used in generators by default in Windows platforms. If you want to use different directory structure or platform, you have to modify the '_create make for API' report to suit for your purposes.

To ensure that the SOAP code will run in your web browser, you need to copy `.jar` files from both Axis and Xerces2 distributions to your browser's Java Runtime Environment `lib\ext` directory (e.g. `c:\Program Files\Java\j2re1.4.2_02\lib\ext` in Windows). The `.jar` files for Axis are:

- `axis-ant.jar`
- `axis.jar`
- `commons-discovery.jar`
- `commons-logging.jar`
- `jaxrpc.jar`
- `logj-1.2.8.jar`
- `saaj.jar`
- `wsdl4j.jar`

And for Xerces2:

- `xercesImpl.jar`
- `xmlParserAPIs.jar`

After copying these files, you still have to grant permissions for the Java Runtime Environment to run these files. To do this, add the following definition to your `java.policy` file (e.g. `c:\Program Files\Java\j2re1.4.2_02\lib\security\java.policy` in Windows platform):

```
grant codeBase "file:///C:/MetaEdit/reports/*" {
permission java.security.AllPermission;
};
```

Please remember to change the path on the first line for MetaEdit+ report output directory, if needed.

You have now completed setting up the required SOAP components. There is, however, one more thing we have to do before we can try out the visual debugger: we must start the API server in MetaEdit+. To do this, start the API Tool from the toolbar in the MetaEdit+ Main Launcher or by selecting **Repository | API tool** from the Launcher menu.

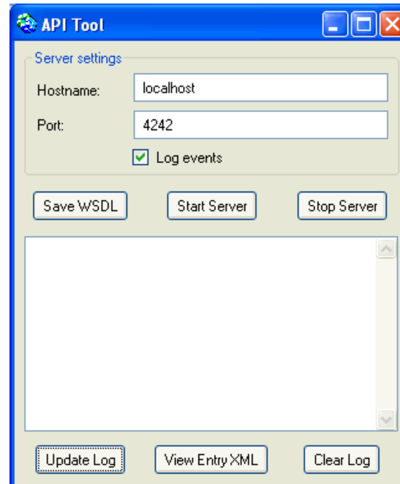

Figure 5-1. The API tool.

In the API Tool (as shown in Figure 5-1), first press the **Save WSDL** button. This will open a file dialog asking where the `MetaEditAPI.wsdl` file should be saved. Choose the MetaEdit+ home directory (e.g. `c:\MetaEdit` in our example). The WSDL (Web Service Description Language) file contains the definitions for API commands and as they are always the same for the current version of MetaEdit+, you need to save this file only the first time you use the API. When the WSDL is generated, start the API server by pressing the **Start Server** button. You can now minimize the API Tool and proceed by trying out the visual debugger.

5.2 VISUAL DEBUGGER FOR THE WATCH EXAMPLE

The Watch example distribution already contains an implementation of a visual debugger. To try it out, set the **Generation target platform** to 'API' in the property dialog for the '2004Models' graph, make sure that API server is started and all browser windows are closed, and run the 'Autobuild' report. The test environment will start the web browser again, and you can choose which watch model you want to test. When you activate the watch model and play around with it, the test environment will make a SOAP call to MetaEdit+, opening the diagram for the state machine that is being executed and highlighting the active state in the diagram (as in Figure 5-2). Thus, you can now run watch applications and follow their execution through the models.

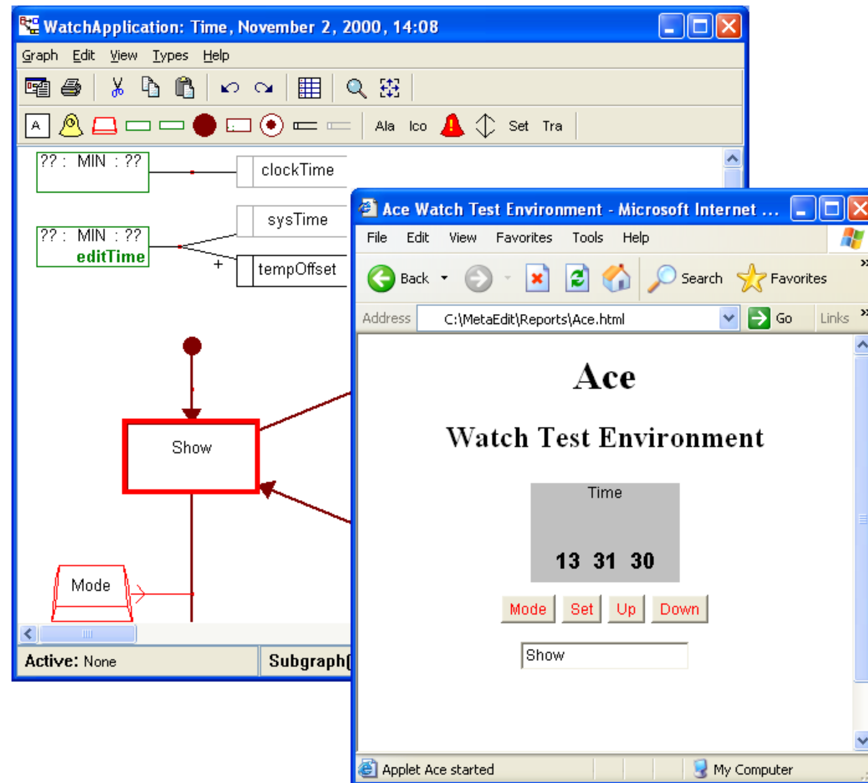

Figure 5-2. Visual debugging of a watch application.

How does it work, then? As the general behavior of animation (i.e. highlighting the target state after a state transition) remains the same for all elements, we found it most feasible to implement it as a framework feature. Therefore, API-specific versions of Master and AbstractWatchApplication framework Java components were created. It took one day from one developer to make these extensions. The ultimate benefit of this solution is of course that there were no changes in models and only a minimum amount of changes in code generators (i.e. we had to create a new API-specific high-level '_create make for API' and '_compile and execute for API' reports).

The basic implementation idea of our visual debugger is very simple: at the end of each state transition, a SOAP message requesting highlighting of the specified model element is sent to the SOAP server running in MetaEdit+. MetaEdit+ processes the message, highlights the requested element and sends an acknowledging SOAP message back to the client application (this return message may also contain return values from API operations, but in this example only a NULL value is returned). For more information about the actual implementation of the visual debugger, we advice you to study the Java code in Master and AbstractWatchApplication framework classes.

6 Conclusion

In this example we have shown how a DSM environment can be implemented and used in MetaEdit+. While the example domain, digital wristwatches, is about embedded real-time software development, the principles explained here work in the vast majority of other domains.

The key success factor in a development process like this is the domain knowledge. Our experience suggests that when sufficient domain and platform expertise is available, designing and implementing a DSML is a reasonably fast process. This approach can be adopted in large organizations with multiple product families and globally distributed development sites, and equally well in small organizations with fewer products and smaller development teams.

Why not in your organization as well?