### CHAPTER 9

## DIGITAL WRISTWATCH

Thisexamplediffersfromthe othersinthatthe intentionwasnottobuildacommercially viable family of products. Instead, the initial purpose was to provide a fully worked example of a Domain-Specific Modeling (DSM) language, generator, and models to demonstrate the principles of DSM, as part of our MetaEdit+ evaluation package. The actual process of creating the modeling language was, however, made as natural as possible, following the practices that had become established in real projects. The artificial background of the example is perhaps most clearly seen in its limited scale. Conversely, this small size is its strength as a pedagogical tool: small enough to be understood in a relatively short time but large enough to provide realistic insights into every aspect of DSM.

9.1 INTRODUCTION AND OBJECTIVES

The background for the digital wristwatch language is thus a fictitious manufacturer of digital watches, circa 1985—we shall call them Secio. Secio has noticed that producing the software for each watch by hand is becoming a significant bottleneck, as consumers demand functionality beyond simple setting and display of the time. It has also been realized that different consumers want different functionality and have different requirements for ease of use versus extensive functionality, physical compactness versus amount of information displayed, and so on.

Domain-Specific Modeling: Enabling Full Code Generation, Steven Kelly and Juha-Pekka Tolvanen Copyright # 2008 John Wiley & Sons, Inc.

191

Therefore, Secio has decided to build its range of watches for next year as a product family. There will be different watches for different consumer types and price points, but the watches will be able to share common parts of the software. Such common parts will include basic framework such as the ability to show numbers on an LCD display, as well as individual applications such as a stopwatch or a countdown timer. The basic framework components will be present in all watches, and either already exist or will be coded by hand. All the individual applications will not be present in all watches, yet it is hoped that later addition of an existing application to an existing watch could be a simple operation.

Fora varietyofreasons—some technical,some political—Seciohas decided to tryto create a DSM language for building the watch applications. The main objectives are to reduce the development time and complexity of building watch applications. In particular, the current watch software is one large piece of code with little separation of concerns. Being able to separate out different parts of the code would improve reuse possibilities, and also allow the developer to concentrate on one area at a time, thereby reducingerrors.TheseparationofbehaviorintoModel(operationsontimevalues),View (displayofthetimeandicons),andController(userinputonthewatchbuttons)willform oneweapontodivideandconquerthemassofcode.Themainweaponsthoughwillbethe higher level of abstraction and the close fit of the modeling language with the problem domain. Developers will be able to concentrate on the application behavior as perceived by the end user, rather than on the lower-level details of how to implement that behavior.

Behind the stage dressing of Secio’s objectives, the main objectivewas to provide a fully worked example of DSM. An important component in convincing people that DSM works is making the code generated from example models actually run. This presents several challenges not normally found in a real-world case of DSM. First, the most significant challenge is that the users must have a compiler installed for the language of the generated code. These days, compilers tend to come as part of a full IDE, requiring a large investment of time, disk space, and possibly money to set up. Second, the user must have a runtime environment compatible with the format produced by the compiler. For both compiler and environment, there may also be various settings such as paths that are specific to each PC, and these settings will normally need to be synchronized with the generated code. Finally, the users will have a variety of different backgrounds, and varying experience with different languages, IDEs, and programming styles.

The need to make an application that would compile and run on many different platforms pointed away from C, the natural language choice for an embedded system. WhilecorewatchbehaviorinCwouldhavebeenplatformindependent,theGUIwidgets and graphics library in C tend to be platform specific. Also, the majority of commonly used C compilers and IDEs that users were likely to have are commercial products. The installationanduseofthefreelyavailablecompilersaregenerallysufficientlydifficultto deter someone whose only motive would be to see an example application.

While other languages with freely available compilers and runtime environments existed, the mainstream choice—and hence most likely to already exist on users’ computers—was Java. The resulting Java applications would also be small enough to be easily passed to other users, and availability of at least a Java runtime environment on a PC is virtually guaranteed. While Java was not the current language of choice for embedded development, it was originally developed for such devices and would be familiar to a large number of users.

9.2 DEVELOPMENT PROCESS

The development of the watch modeling language was carried out by Steven Kelly and Risto Pohjonen of MetaCase over a couple of weeks. As this was an example rather than a real-world case, there was no outside customer: as both developers had owned digital watches in the 1980s, they felt qualified to play the role of domain expert. This distinguishes this from the other example cases, where the authors did not have sufficient domain knowledge at the start of the project to create the language on their own. This case thus brings the authors’ positions closer to those of readers thinking about their own application domains, and gives us a good opportunity to examine the thought processes of a domain expert. As will be seen, creating a DSM language is largely a question of determining what facts need to be recorded about each application in that domain, and where in the modeling language to store these facts.

The total time spent was approximately 10 man-days, including the modeling language, a full set of models, the generator, and the domain framework. Since this was the first Java project for either developer, the time also included learning the language, its libraries, and development environment. Over the subsequent years, a few more days have been spent on upgrading the framework and generator to work on other desktop platforms, cope with later Java versions, produce watches that run on mobile phones, and add support for model-level tracing of a running watch application. None of these changes has required changing the models, and the result has always been fully running applications, identical in behavior but with environment-related differences in appearance and sometimes in code.

9.3 MODELING LANGUAGE

The initial idea was to have a modeling language for building digital watches. In this section, we will follow the analysis of the domain and the development of the language in chronological order.

9.3.1 Reusing Watch Applications in a Family of Watches

It was soon evident that it would be good to break a watch down into its component applications: time display, a stopwatch, an alarm clock, and so on. Beyond providing a sensible modularization of the whole watch, this would also allow a watch application to be reused in different watches. Since these watches would have different physical designs—displays, buttons, and so on—there was a need for some decoupling of references to these physical elements in the models. For instance, if a model of a watch application wanted to specify that a certain operation was caused by a certain button, we had to answer questions about whether that button was available in the given physical watch, whether it would be named in the same way or have the same semantics, and what to do if no such button existed.

Thinking about this issue prompted the idea of explicitly modeling a whole group of watches as a family. This would be a separate level of modeling, probably with its own modeling language. Often in DSM such a level exists, but it is not always explicitly modeled: it is enough to simply have several products—watches in this case—each built from its own set of models. However, making reuse of models explicit generally makes it easier to maintain them and concretely shows the dependencies of the various components. For instance, a change in the Alarm application to require an extra button would affect which physical designs of watch the Alarm could be used in. If Alarm had simply been reused, this effect might not have been obvious. If, however, there were a top-level model showing each member of the watch product family, which physical watch body it used, and which watch applications it contained, the effects of that change would be easier to see.

If there was a need for a mapping between the buttons mentioned in a watch application model and those present in a physical watch, there was also the question of how to model this mapping. Would the mapping be included as part of the top-level family model, or would it be a separate kind of model between the top-level family model and the actual watch applications? Further, who would be responsible for building these mappings: the watch application designer, the family designer, the designer of a particular watch model, or somebody else? Similarly, would a separate mapping be required for each pair of a watch application model, for example a stopwatch, with a physical watch body, or could one mapping be reused over many watch applications or physical watch bodies?

The question of the mapping was thus difficult in both technical and organizational terms, and also hard to decide at this early stage. While we did not even have modeling languages for watch applications and watch families, it seemed unrealistic to expect to pick a good solution to a problem that would probably only become apparent once several families had been built. We thus decided to go with the simplest thing that could possibly work: there would be a limited number of named buttons, initially just Up, Down, Mode, and Set. Each physical watch body could contain any combination of these buttons, and similarly each watch application could refer to them directly. While less flexible than a different mapping for each watch, this had a good chance of working well for both watch modelers and users. Both groups would prefer a consistent semantics for the buttons: if in one watch application Up was used to increase a time value, and in another the same function was achieved using Set, learning to use the watch would be rather difficult!

Now we had a fair idea of the contents and division of labor of the two modeling languages. The family model would contain a number of watches, and each watch would be composed there from a number of watch applications and a physical watch body. A physical watch body would specify a number of buttons. These buttons would also play a key role in the definition of the watch applications: different buttons would cause different actions in different applications.

9.3.2 Watch Application Behavior

While the family model would have been easier to work on, we decided to tackle the watch applications next. If DSM were to mean anything, it would have to be able to specify the differing behaviors of the various applications sufficiently exactly that we could generate full code directly from the models. One possible tactic at this point would have been to hand code a couple of watch applications. This would have given us an idea of what kind of behavior they would actually have on a low level, as well as insights into what elements of that behavior might repeat over several different watch applications. However, as neither developer had programmed in Java at this point, it was thought that trying to go this way would be a bad idea. DSM is generally about abstracting from the best practice, and clearly there was no way our first attempts would fit into that category.

Instead, we decided to concentrate on the actual user-visible behavior of the watch applications, and trust that our general development experience and instincts would tell uswhenwehadasufficientlyexactspecificationtoenablefullcodegeneration.Thefirst question was thus about the buttons, the only way the end user can interact with the watch:When a user presses a button ina givenapplication,does thatbutton always have the same effect? In other words, could we program a watch as a composition of applications and an application as a composition of button behaviors? In some simple cases, this appeared to be true. In more complicated cases, for example setting the time, itmightbetrueifunderstoodsufficientlybroadly.Forinstance,pressingSetintheAlarm application would start setting the alarm, and pressing Set again would stop setting the alarm. While starting and stopping the set process were two different things, they could perhaps be understood as a toggling of the set process. Going further, though, within the set process it would be normal to press the Mode button to toggle between setting the hours and setting the minutes. However, outside of the set process the Mode button would be expected to exit the Alarm application and move us to the next application.

Although further stretching of semantics might have made it possible to model applications as simply one behavior per button, that behavior would have been more and more conditional on what had occurred before. It thus seemed best to admit that the action taken when a button was pressed depended on what state the application was in. A watch application could thus be modeled as some kind of state machine. Pressing buttons would cause transitions between states, and possibly also other actions.

At this point, it would normally have been a good idea to move from models that were either imagined or drawn simply on paper, to actually building a prototype modeling language. This would generally give a better basis for further decisions, while also making more concrete those things that seemed already decided—and perhaps revealing problems in them. Perhaps because both developers had a deep knowledge of DSM, MetaEdit+, and state machines, we actually continued on paper for a little while longer, looking at the possible kinds of actions.

The clearest action with a user-visible result was the control of the little LCD icons for applications: a Stopwatch icon for when the stopwatch was running, an Alarm icon for when the alarm was set to ring, and so on. We decided to make turning the icons on or off an explicit action, taken when following a transition between two states.

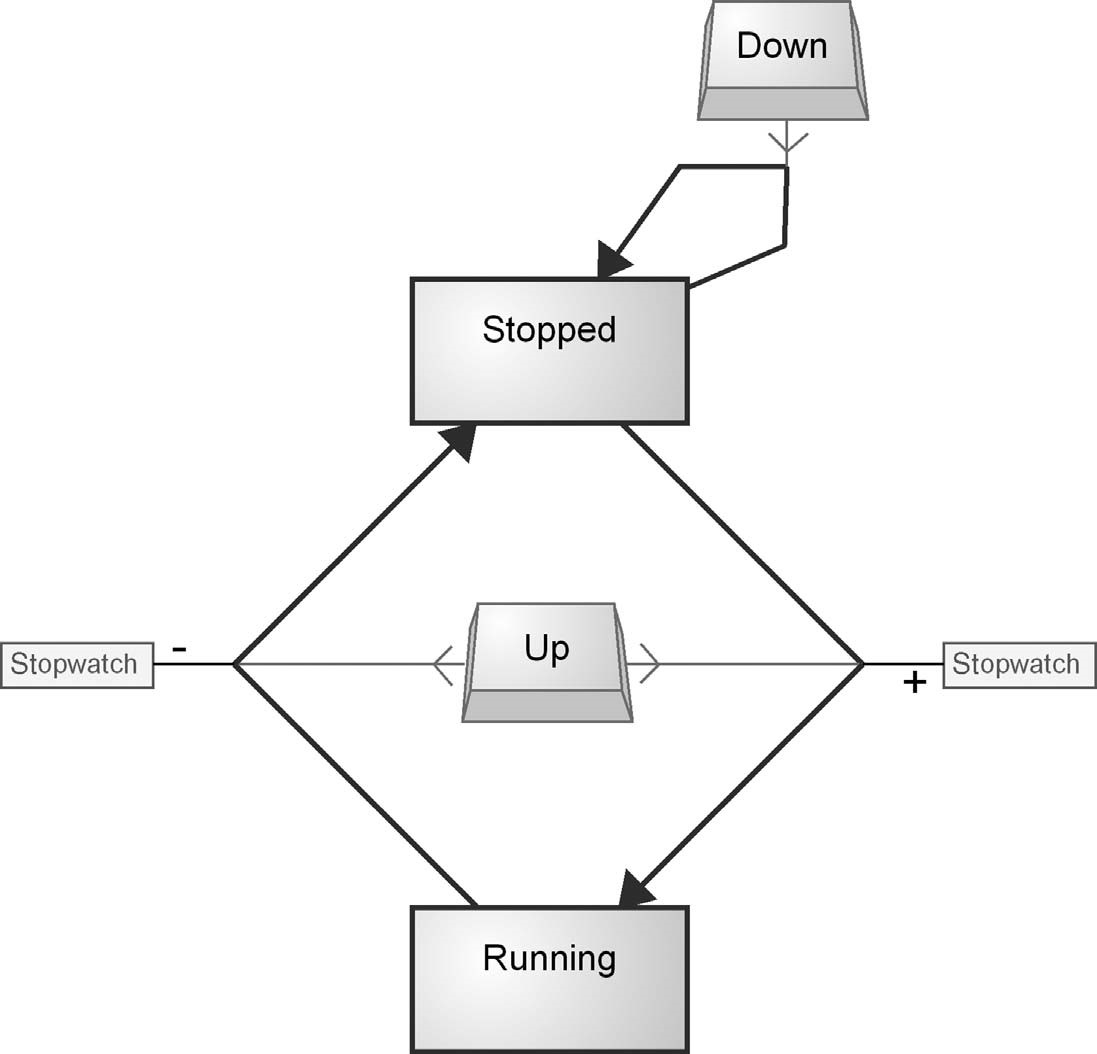


FIGURE 9.1 States, buttons, and icons (40 minutes)

Another possible approach would have been to make each state contain the information about the status of each icon, but we decided against this approach for two reasons. First, the on/off state of the icons was persistent when their application was exited and the next application started, and it was then unclear whether the original application was still in some sense live and in a particular state. Second, and more importantly, storing the icon status in each state would have meant either each state having to store the status of each of the icons, leading to maintainability problems if the number of icons increased, or each application only being able to control one icon, which while initially appearing to be true, might not necessarily be so in more complicated cases. It was also thought that the number of actions needed to make an icon behave as desired would probably be less than the total number of states in that application, making modeling easier if actions were used. We thus decided to represent the icons as objects in the modeling language, and to allow modelers to change their state by drawing a relationship to the icon to turn it on or off. The result so far, after 40 minutes of metamodeling, is seen in Fig. 9.1.

9.3.3 Watch Applications Manipulate Time

After the work so far, we could model a stopwatch that would know which state it was in (say, Stopped or Running), move between those states when the user pressed a button (say, Up), and that would also turn on or off the Stopwatch icon when moving between the states. All very well, but it might be useful if the stopwatch actually recorded something about the elapsed time, too!

If Buttons represented a Controller in the MVC triad that Secio envisaged, and Icons represented a View, then being able to record elapsed time was clearly part of the Model. Storing, recalling, and performing basic arithmetic on time variables were therefore important parts of watch application behavior that we needed to represent in some way in the modeling language. We thought it was fair to assume that any digital watch company worth its salt would already have existing functions to return the current time, along with functions to store and recall and perform basic arithmetic on time variables. This meant we could consider the time operations on a high level, rather than the implementation details of the bits and bytes of data structures to store time.

The first question to be answered was one about the boundaries of the domain: did we need to handle dates as well as time? Thinking through the various possible watch applications, it seemed that most had no connection to dates: an alarm could not be set for a given date, for instance. In fact, only the basic time display would show dates. The handling of dates would most likely be identical from watch to watch: a source of commonality, not of variability. In addition, the behavior of dates is complex, with weekdays, day numbers, months with different numbers of days, leap years, different orderings of date elements for different countries, and so on. All in all, it looked like building a modeling language for handling dates would not result in any productivity improvements for Secio. Rather than introduce complexity for no gain, we decided to leave dates out at this point and concentrate on time.

Thinking about the stopwatch application, it seemed there could be two ways to think about time variables and the underlying platform. In the first way, the platform would just respond with the current time when asked, and we could store that and then later ask again for the new time and calculate the difference. In the second way, the platform could offer a service to create a new timer, which it would then continuously update. While the second way would make modeling the stopwatch easier, it seemed unlikely in the resource constrained embedded environment of a digital wristwatch: updating multiple timers simultaneously would place unnecessary demands on processing power and battery life. We would, therefore, need a way of representing subtraction of time variables, and most likely addition too.

Thinking about other watch applications such as the alarm clock, it appeared there might be a need for another time operation. When editing the alarm time, pressing Up would increment (say) the minutes, which could be represented as adding one minute. However, when the minutes reached 59, the next press of Up would take them to 00, without incrementing the hours. In other words, this was not a true increment of adding one minute, but rather an increment to just one unit of the whole time, which would roll around at either end of the unit’s range. This could have been represented in the models with a comparison operator and an subtraction, for example: “if minutes equals 60 then subtract 60 from minutes.” However, this would be needed for every edit operation, for every unit of the time, and once for each end of the range. It would also require a modeling concept for comparisons, which we did not seem to need otherwise. We therefore decided to make a “Roll” operation a first-class concept in the

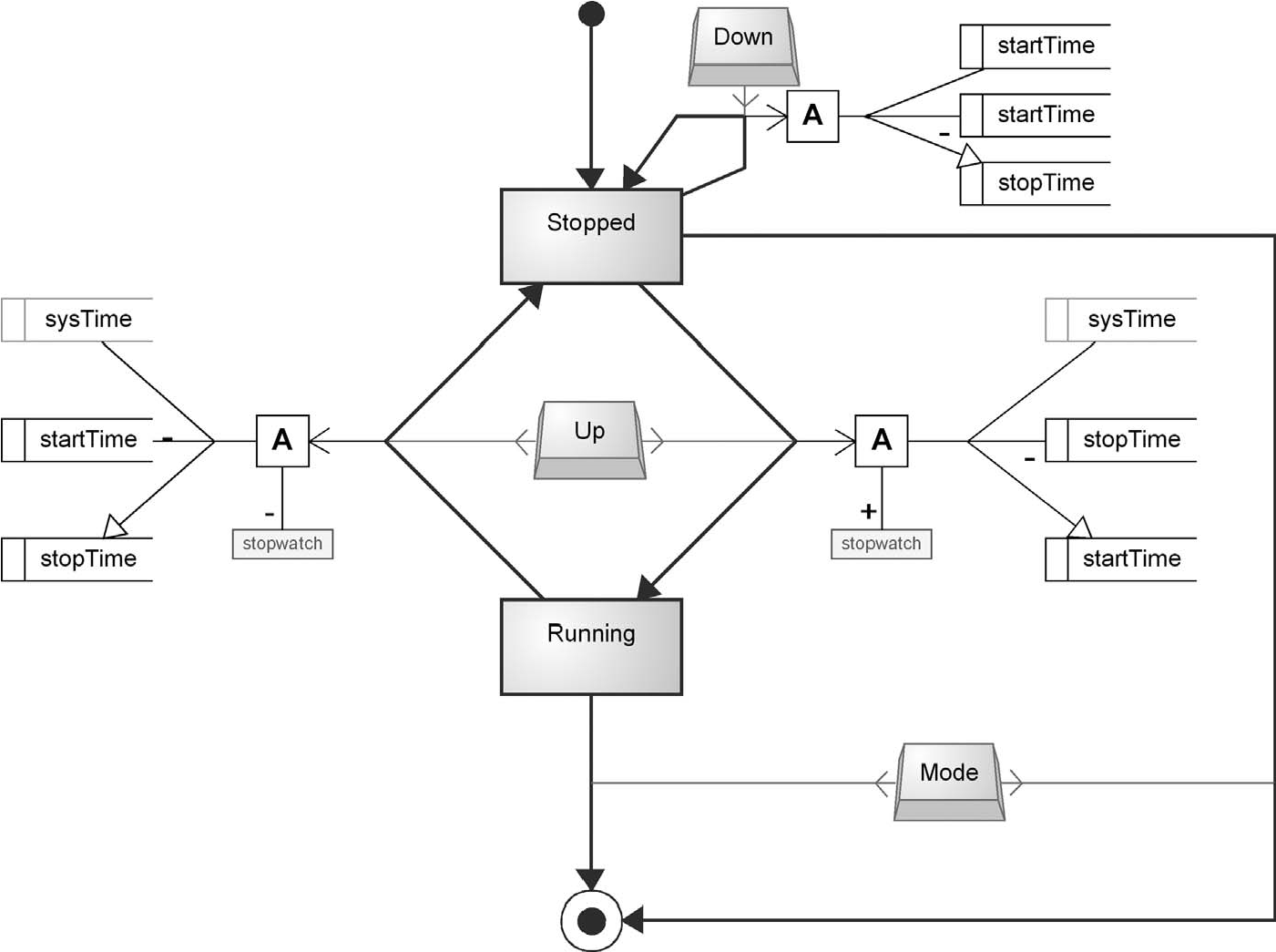


FIGURE 9.2 Time manipulation (70 minutes)

modeling language, incrementing or decrementing a particular unit in the whole time variable.

Now we knew the concepts we wanted for time variables and operations, how best to represent them in the modeling language? Programmers would be used to a textual representation, and we could indeed have had a textual DSL providing a natural syntax, restricted to just these operations. However, because there were so few operations needed, and each calculation would need only one or two operations, we decided to represent time variables as actual objects in the modeling language, and operations as relationships connecting to them. This would fit well with the earlier decision to have icons as objects, and actions on them represented as relationships drawn to them from state transitions. The result so far, after 70 minutes of metamodeling, is seen in Fig. 9.2.

9.3.4 Time Display

At this point, we had the majority of the concepts of the modeling language and had covered the Model and Controller parts of the MVC triad fairly thoroughly. We also had addressed some of the View part in the form of the Icons, but the most important View element, the display of time, was still largely untouched.

The display code of a real-time application is notorious for being difficult to get right. Had Secio been a real organization, its experienced developers would have been throwing up their hands at this point and saying “All you’ve done so far are the easy bits, and we never had problems with those anyway. Getting the display to update smoothly in real time is hard work, and you’ve not even touched on it. Besides, that’s the area where our developers make most of their mistakes: they just can’t seem to get the thread synchronization and semaphores right, no matter how many times we explain it.”

What then can we do for Secio? First, we can take a long step away from the implementation details toward the problem domain and see what actually needs to happen on the watch display from the end user’s point of view. Let’s take the most used application, Time. What is it displaying? The current time, of course! Now, the value of the current time is changing all the time, so are the updates for that something we need our models to specify? In a sense, maybe not: what is being displayed, on a high level of abstraction, is simply the current time. What then about the Stopwatch, what is it displaying? That seems to vary a little: initially, it just displays all zeroes; when it is running, it is displaying the elapsed time, which is updating constantly but always equal to the current time minus the original saved starting time. If we look at World Time, that displays the current system time adjusted by some fixed offset. A Countdown Timer displays the time the alarm will sound minus the current system time.

This is interesting: we seem to be able to express quite simply the basic idea of what each application should be displaying. This is actually not all that surprising: if the time value displayed required some immensely complicated algorithm to calculate, few people would be able to interpret its values. Of course, actually making the value display and update smoothly will be tricky—real-time software always is—but perhaps the models themselves can remain quite simple. Could we abstract out the complicated parts of real-time display into the domain framework? This would allow the model to simply specify what to display, while an expert developer’s handwritten code, written once but reused for all displays in all watch applications, would handle how to display it.

In our blessedly deep ignorance of the intricacies (and bugs!) of Java’s thread handling and synchronization, we decided to take this path. A watch application would specify a calculation to obtain the time to display, either just one such Display Function for the whole application (e.g., Time, Alarm) or perhaps different Display Functions for different states (e.g., Stopwatch states for being stopped and running, or Countdown Timer states for setting the timer and counting down). The modeler’s burden would end there, and not a thread synchronization or semaphore in sight. The expert developer would write a display update function that would run in a separate thread once every few milliseconds, ask the application to perform its Display Function calculation, and update the display with the result. Our example model at this stage, after 105 minutes of metamodeling, is shown in Fig. 9.3.

9.3.5 Odds and Ends

When building a modeling language, there are always things that get missed on the first pass. Thinking through a concrete example application and its model helps keep things on track, but any given example will rarely contain an instance of everything

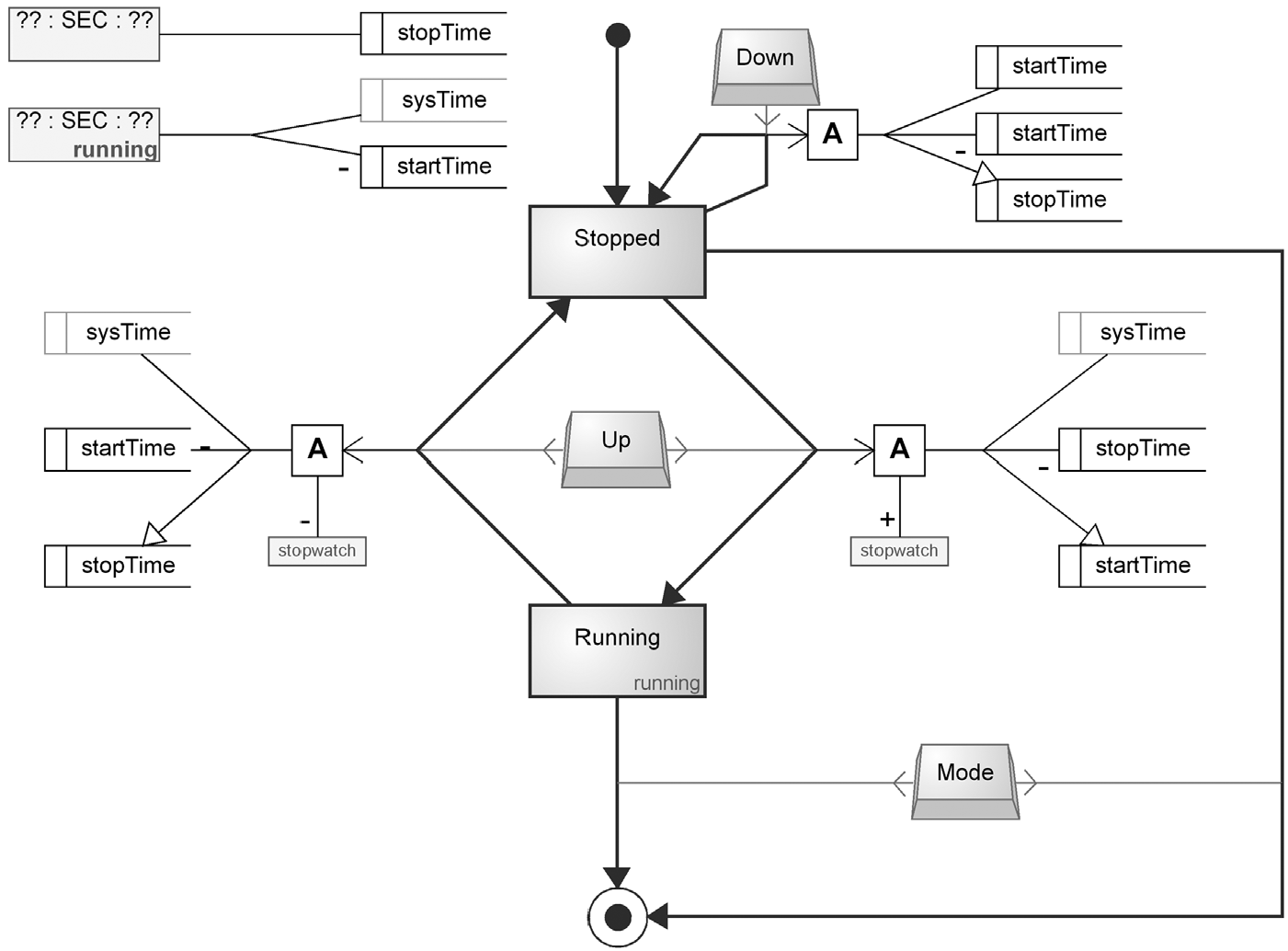


FIGURE 9.3 Display Functions (105 minutes)

needed in the modeling language. So too in this case: Stopwatch had been a good example for many areas, but it was missing features that were used in several other watch applications. We also missed one important part of Display Functions, which only became apparent in the Stopwatch when it was used in several different watch models.

The most obvious omission from the modeling language was the complete lack of the concept of an Alarm. It had been mentioned early on but never focused on for long enough to actually figure out how it should be modeled. Setting the alarm seemed easy enough: we could edit some time value for it, either the time at which the alarm would ring, or the amount of time until it would ring. An alarm ringing would be similar to a button press in that it could cause a change of state. More interestingly, this change of state could happen at any time, even if the watch was running a different application at that point. Clearly, we could not draw transitions from every state in the entire model in case the alarm rang there. Instead, we settled on having an alarm symbol, and a single special “alarm” transition from that to a state. Whenever the alarm rang, its application would take control and jump to that state. On exiting the alarm application, the watch would return to the application that was current when the alarm rang.

In deciding whether to set alarms at a time of day or as an amount of time until the alarm rang, we hit a problem. What would happen if the user changed the watch time after setting the alarm? If we stored alarms as a time of day, a standard Alarm Clock application would be fine, but a Countdown Timer would suddenly be wrong.

For example, if a Countdown Timer was set at 5:00 p.m. for 10 minutes, it would have stored its alarm as 5:10 p.m.; if the user moved the clock back an hour at 5:05 p.m. to 4:05 p.m. the countdown alarm would not ring for another 65 minutes. Conversely, if we stored alarms as an amount of time until ringing, the Countdown Timer would work fine, but a standard Alarm Clock would ring at the wrong time. We decided to solve this by allowing each Alarm object to specify whether it responded to changes in local time or not.

Thinking about setting alarms also reminded us that we needed some way to make a pair of digits on the time display blink, so the watch user would know which time unit he was editing. While we could have avoided adding a concept by saying that a time unit blinking was just a special case of an Icon, this would have meant that going from an “Edit Hours” state to an “Edit Minutes” state would have had to turn off “blink hours” and turn on “blink minutes.” We would also have to make sure that every path to “Edit Hours” would turn on the “blink hours” pseudo-Icon. A better approach seemed to be to recognize from the names of the states that blinking of a time unit was actually a feature of the state itself. We thus added a property “Blinking” to State, making it a list where the user could choose from Hours, Minutes, or Seconds. This also fitted well with choosing a Display Function in each state: the actual display thread would need to know both the function and any blinking time unit to be able to keep the display updated. Having the blinking property in the Display Function itself would not have helped, as it would mean having to create duplicate Display Functions differing only in which time unit was blinking.

Display Functions were also the source of the third addition to the modeling language. We realized that the mapping of the different time units of a time value to actual digit pairs on the watch display could be different in different physical watches, and also different between two watch applications in the same physical watch. For instance, in a lady’s watch there are often only two digit pairs. In the normal time display these are used for hours and minutes, but in a Stopwatch and possibly in a Countdown Timer, they should show minutes and seconds. However, editing a Countdown Timer generally only allows choosing hour and minute values, so while editing those should clearly be shown. Specifying a hard mapping in a watch application from time units to digit pairs would not work though, since it would mean the application would not work so well in a different watch model with a different number of digit pairs.

We tried several different schemes for specifying mappings, including specifying which time unit would be shown in the leftmost or rightmost digit pair. Neither of these gave satisfactory results when we wrote out on paper what would be displayed in various states, applications, and physical watches. In the end, we hit on the idea of a Display Function specifying a “Central time unit,” where “Central” was a mixture of “most important” and “should be displayed in the center digit pair.” A little heuristic—for what counts as “center” when there are an even number of digit pairs— rounded off the scheme, and we added an appropriate property to Display Function. Now we had a language that could specify all we could think of about watch applications.

9.3.6 Putting it all Together

Having a language that could specify watch applications brought us back to our starting point: Secio wanted to be able to compose a product family of Watches out of Watch Applications. One part of each Watch would thus clearly be a list of the applications it contained, for example, Time, Stopwatch, and Alarm. AWatch would not just be a jumble of applications though: the user would cycle through the applications in a certain order. At first, we thought this would be represented as a collection property, to which the modeler could add the desired Watch Applications. The idea of the cycle, however, gave us the idea of representing this graphically. Since we had already considered the possibility of having layered state machines for the Watch Application—a State in an application could decompose to a subapplication— we hit on the idea of using the same Watch Application modeling language to specify application composition.

This approach brought several benefits. First, it reduced the number of concepts the modeler would have to learn. Second, it gave the modeler more freedom in deciding how many levels the models should be divided up into: if a Watch had only one Application, for example a Stopwatch, the Watch could point directly to the Stopwatch, rather than having to specify an intermediate level with a single Stopwatch Application. Similarly, if a given application became too large for a single graph, or if parts of an application could be reused in other applications, they could be separated out into their own subapplications—on as many levels as seemed appropriate. Third, it allowed the possibility of more freedom at the application composition level: rather than restricting all Watches to always move between their applications in a fixed cycle, the choice of which application to go to next could be as complicated as the modeler desired. An example model is shown in Fig. 9.4.

While one part of a member of the product family of Watches would thus specify the behavior of the Watch, a second part was needed to specify the physical details of the Watch. Mostly these would be the domain of an industrial designer, but the behavioral part would also need some of that information. In particular, if we wanted to generate a Java applet as a Watch emulator, we would need to know three things: the buttons on the watch, the icons it had, and the number of digit pairs it could display. The first two could in theory have been found by trawling through the set of applications and including all referenced buttons and icons, but this was not how we wanted things to work. We wanted a Stopwatch application, say, that used Up, Down, and Mode, to be usable in a watch without a Mode button, for example one containing only the Stopwatch application. Similarly, rather than have an error if an application tried to turn on an icon that was not present in the physical watch, we would simply do nothing at that point. This made the watch applications more reusable, and allowed the application modeler to concentrate on the application itself, without having to know beforehand the exact details of the physical watches in which it would be used.

We decided to call a member of the product family a Watch Model—“model” being used in the sense of “a Corvette is a car model,” rather than “a graphical model of a car.” Each Watch Model had a Logical Watch, which pointed to the graph showing

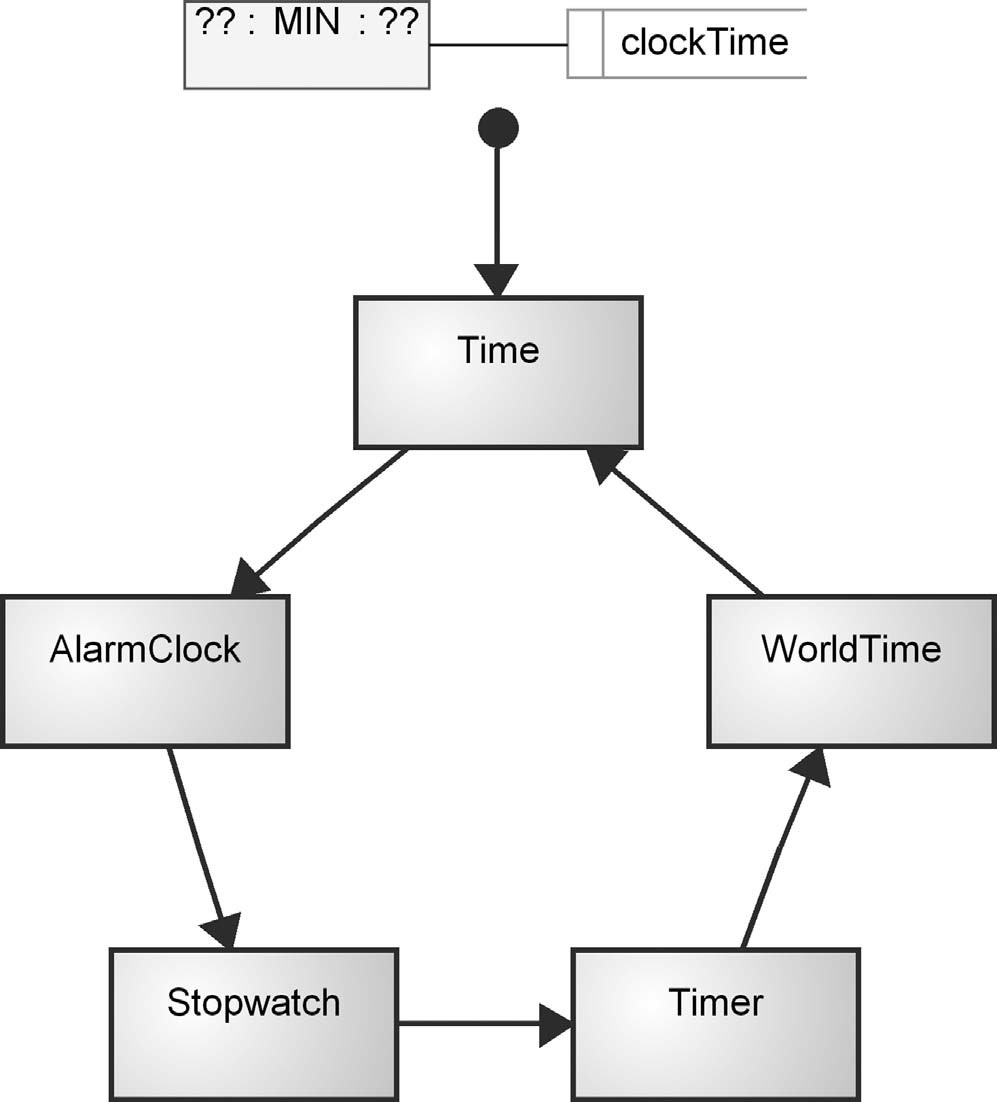


FIGURE 9.4 Logical watch as a cycle of applications

the cycle through the watch’s applications, and a Display, corresponding to the relevant parts of the physical watch body: icons, digit pair zones, and buttons. The Logical Watches and Displays would thus form components, and each might be used in more than one full Watch Model.

As these concepts were different from those used in the Watch Application modeling language, it was clear that we actually had a new modeling language here. We decided to make it graphically rather verbose, for pedagogical rather than practical reasons. Each Watch Model (rounded rectangles with a light green fill) showed the Display and Logical Watch it used, and the sets of possible Logical Watches and Displays were also shown in the same diagram (Fig. 9.5).

Another possibility here would have been to use a matrix representation rather than a graphical diagram. In models shown as matrices, the axes contain objects and each cell contains the relationship (if any) between the corresponding objects. The Logical Watches could thus be listed down the left side of the matrix and the Displays across the top. The cell at the intersection between a given Logical Watch and Display would then represent a possible Watch Model composed of those parts. The relationship’s name would be the name of the Watch Model and would be shown in the cell. Fig. 9.6 shows how this would provide a useful visual

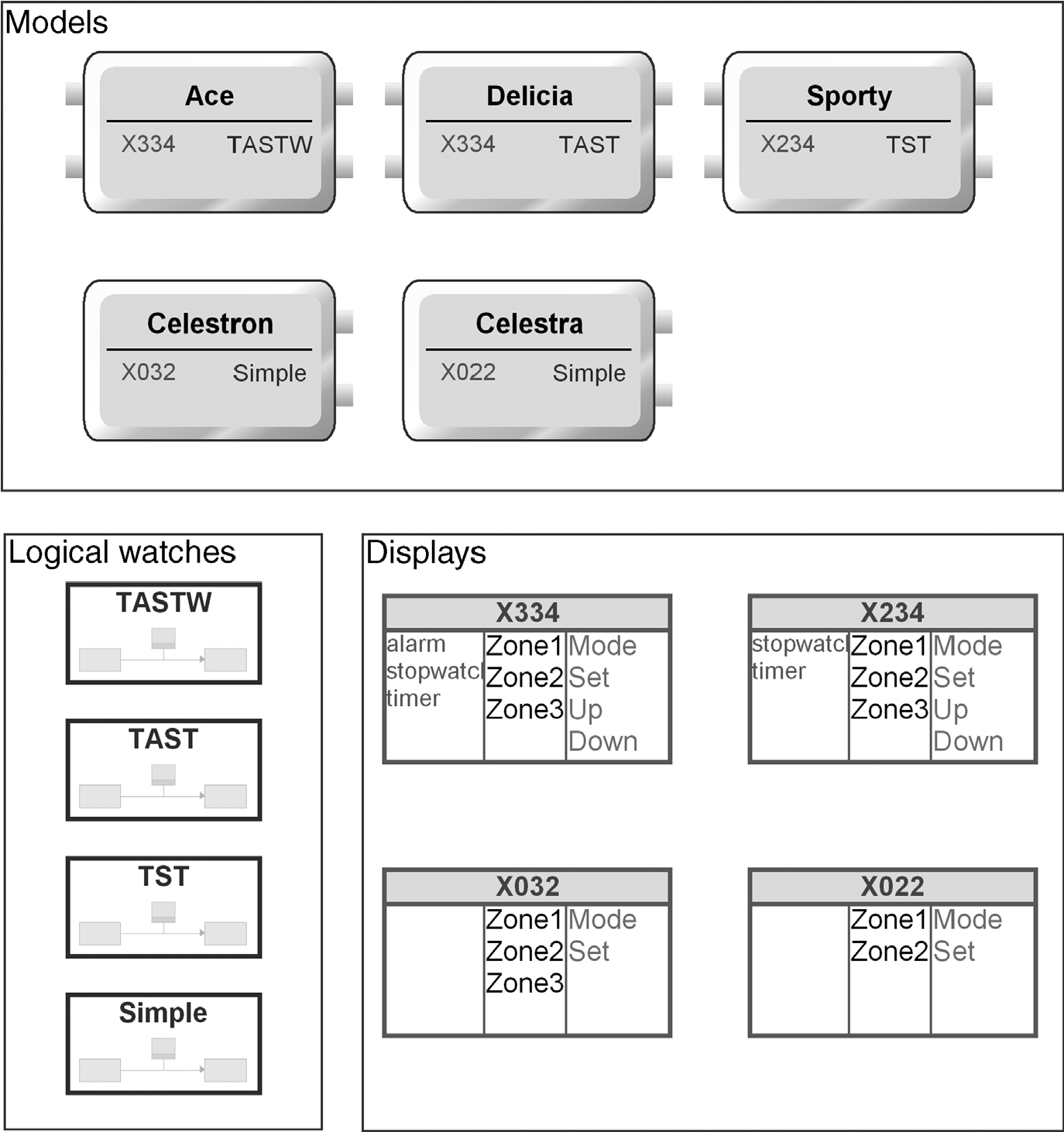


FIGURE 9.5 Watch Family diagram

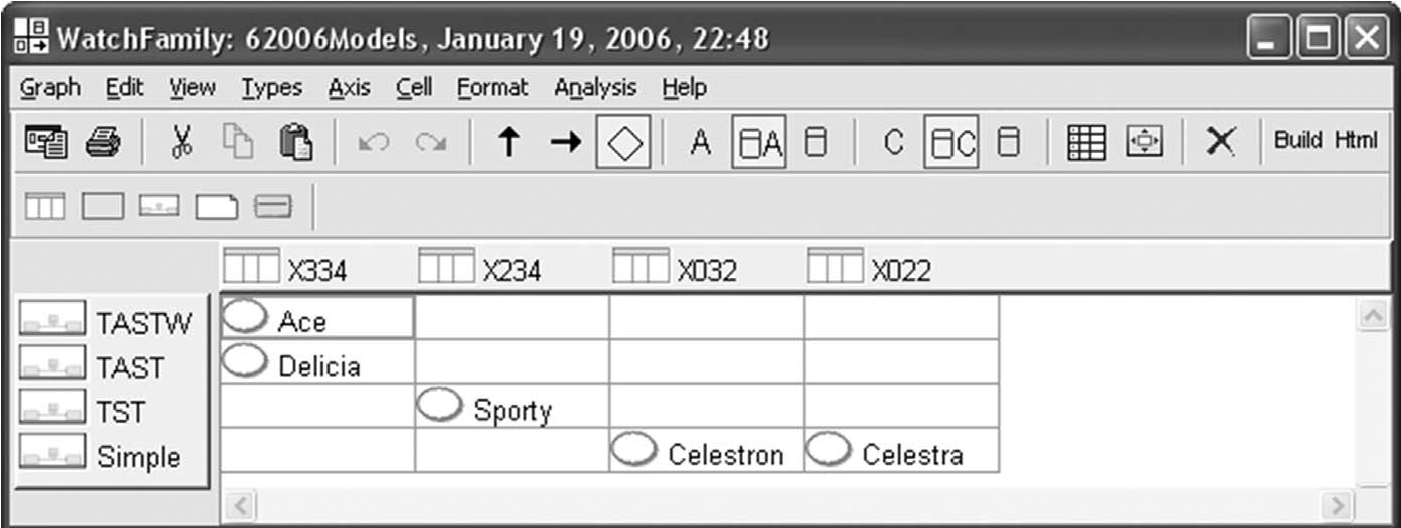


FIGURE 9.6 Watch Family as a matrix

overview of which potential combinations of Logical Watches and Displays had been used, how often each Logical Watch and each Display had been used, and which potential combinations remained as possible openings for new products. The matrix format would also collect the Watch Models, Logical Watches, and Displays together into their own areas, without the rather clumsy group symbol of the graphical diagram.

9.3.7 Rules of Watch Applications

The main need for rules and constraints in the Watch applications, as in most DSM languages, was in the relationships. The main relationship was the Transition between States, triggered by a Button and causing various kinds of actions. This was a clear case of an n-ary relationship: one relationship connecting several objects.

One question was how much we wanted to allow the aggregation of several transitions into one composite transition. For instance, it would have been possible to allow the same transition to come from many States: for example, a transition to the Stop state, triggered by the Mode button, would often be appropriate for several States. Similarly, we could have allowed the same transition to be triggered by multiple buttons (“Press any button to stop alarm ringing”), and a transition could clearly trigger multiple actions (e.g., setting an alarm and turning on the alarm icon).

However, as the transition would generally link four objects anyway—quite a large number—it was thought better to allow one From State, an optional Event from a Button, an optional Action, and one To State. A small Action object would serve as a placeholder for multiple actions, which would be connected to it by relationships. The Event role from the Button object would be optional: some state transitions could happen without any button press.

The transition from the Start State to the first State would be required not to have a Button: an application is never actually in the Start State, which simply marks the entrance point. Additionally, there was a constraint that a Start State could have only one transition leaving it, and similarly there should only be one Start State in each Watch Application. Interestingly, the Stop State behaves differently: there can be many Stop States, each with possibly many transitions arriving at it. While an application is never actually in a Stop State, there is no ambiguity resulting from this situation, in contrast with the case of Start States.

The other main kinds of relationship whose rules required thought were the time operations. We decided to implement these as Plus, Minus, and Set roles. The contents of the variables pointed to by the Plus and Minus roles are added and subtracted appropriately, and the result of the operation is stored in the variable pointed to by the Set role. There could thus be one Set role, and zero to many Plus or Minus roles. As this left the unsatisfactory possibility of a relationship with one Set role and no other roles, we decided that there should also be a Get role, which would point to the first variable term in the operation. Thus, for something like “stopTime = sysTime startTime” stopTime would be in a Set role, sysTime in a Get role, and startTime in a Minus role.

As it turned out, this was a good decision for code generation: the first term in a calculation is not generally preceded by a plus or minus sign. It was, however, a slightly poor decision in terms of modeling all possible calculations: for example, “stopTime = startTime.” Interestingly, there has not yet been a need for such a calculation. Still, perhaps a better solution would be to replace the Get, Plus, and Minus roles with a single Get role with a property specifying whether it was Minus or Plus.

9.3.8 Notation for Watch Applications

To model an application in sufficient detail to allow full code generation generally requires several different kinds of information. In general-purpose modeling languages each kind of information would be separated into its own language. A clear problem with this approach is that it is hard to get a complete view of the application, as it is split over several diagrams. Such an approach also requires a fair amount of duplication, as objects (or references to them) must be reused in several different diagrams, in order to provide the full information about the object from all viewpoints. This leads to another problem: the difficulty of keeping all the diagrams synchronized.

To avoid these problems, a domain-specific modeling language often includes several different viewpoints within one language. Because the language constructs needed for each viewpoint are only a domain-specific subset of those needed for that viewpoint in a generic modeling language, the resulting language remains of a manageablesize.Itdoeshowevercontainavarietyofdifferentkindsofinformation,and one way to use notation is to help separate the different kinds of information visually.

The Watch Application modeling language contains three main types of information, corresponding to the three parts of MVC that Secio had decided to use. The Model part represents the underlying data on which the application acts, in this case time variables. These parts of the model are shown in black or gray, with gray being used for read-only pseudovariables, such as sysTime, by analogy with graying out a field in a user interface to show it is read-only. The View part represents the output of the application to the user, that is, the visible parts of the application that change as it runs. These parts of the model are shown in green: the Icons and Display Functions. The Controller part represents the input to the application by the user, in this case the Buttons. The Buttons, and the lines from them to the Transitions, are shown in red in the models. Since the behavior of the buttons depends on the State the application is in, we decided to make the States and Transitions a dark red: part Controller, part Model. As each State showed the name of the Display Function it used, which was part of the View information, that name was shown in green.

The symbols themselves were drawn as vector graphics, aiming more for simplicity than beauty. As a State has no clear visual presence in a physical watch, its symbol was just a rectangle containing its name. If its Display Function was not the default for that graph, that was shown in green in a smaller font at the bottom right. If one of the time units was blinking in that State, the abbreviation of the time unit was

shown with four short rays pointing outward. For the Start and Stop States, we chose the familiar representations of a filled circle and a filled circle within a hollow circle. Buttons clearly had a physical counterpart, but also a fairly standard pictogram of the edges of a stylized three-dimensional button. As the physical counterparts on actual watches tend to be very small, we used the pictogram. For the Alarm we drew a yellow bell shape, using a smaller red version of it as a symbol for the special Alarm Transition relationship. Time variables lacked a clear visible counterpart, but were a familiar concept from several generic modeling languages; in the end, we chose the Data Store symbol from Data Flow Diagrams.

An important part of any modeling language is the flow of control or information, which is normally indicated by arrowheads on the role lines. We thus placed arrows on the lines from a Button to the Transition it triggered, from the Transition to the Action it executed, and from the Transition to the State it ended up in. As the last of these was the most important element of flow in the whole modeling language, it was shown most visibly, with a dark red filled arrowhead. The final piece of flow was in the time operations, where the value of the calculation is assigned to a time Variable: this was shown with a hollow black arrowhead.

As always, these decisions were a balancing act, trying to make the meaning of the various parts of the model clear without straying into a graphical melee where every part tries to scream its own importance. If the direction of information flow along a role linewas obvious from other context, no arrowhead was used. This was the case for Actions turning Icons on or off, the Get, Plus, and Minus roles in time calculations, and the role connecting a Display Function to its calculation. The direction for the Button and Action roles of the Transition relationship could also be implied, but small open arrows were added to make clear the part played by each of the roles in this n-ary relationship.

9.4 MODELS

The three models we have looked at earlier in the chapter reflect the three levels found in most of the watch applications. The top level is a single diagram in the Watch Family language (e.g., 2006 Models shown in Fig. 9.5). Each Logical Watch there is decomposed into a simple Watch Application diagram (e.g., TASTW shown in Fig. 9.4) at the middle level showing how it is composed from various applications, represented as a cycle of states. Each state there is decomposed into its own Watch Application diagram (e.g., Stopwatch shown in Fig. 9.3) containing the behavior of that application.

The AlarmClock application in Fig. 9.7 shows some aspects not present in the Stopwatch. When the application starts, we move straight to the Show state, where (as indeed in all states in this application) the Display Function simply shows the alarmTime variable. Pressing the Set button takes us to the EditHours state, where the hours are flashing on the display. Pressing the Up or Down buttons there rolls the hours value of alarmTime up or down. Note that the thick dark red role from EditHours

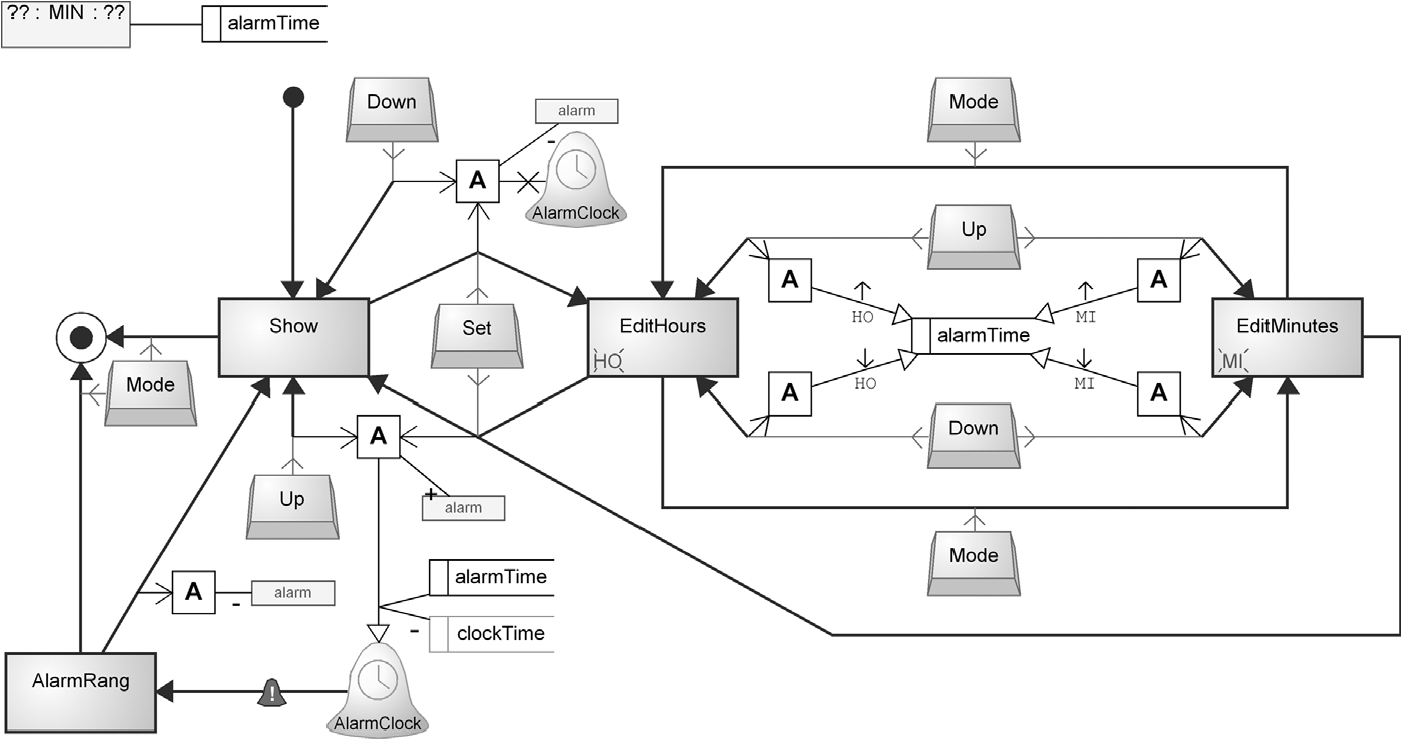


FIGURE 9.7 AlarmClock application

returns back to EditHours along the same path, ending in an arrowhead, making the role leaving EditHours invisible. Another way to draw this would be as a circular relationship, but as these are so common, it was thought that the modeler would prefer this shorthand: all four roles of the transition are present; the From role is just not particularly visible. Pressing Mode in EditHours takes us to the similar EditMinutes, and vice versa.

Pressing Set in either Edit state takes us back to the Show state, carrying out the Action immediately below and to the right of Show on the way. This turns on the alarm icon and sets the AlarmClock alarm to ring after a while, calculated as alarmTime minus clockTime. The alarm is set to be sensitive to changes in local time (indicated by the clock face in its symbol). In the Show state we can press Down to unset the alarm and turn off the icon. Assuming we leave the alarm to ring, when it rings it follows the transition with the small red bell symbol from the Alarm symbol to the AlarmRang state. From there, without any buttons being pressed, we move directly to the Show state, turning off the alarm icon on the way. From the Show state, pressing Mode will exit the application.

9.4.1 Use Scenarios

An important feature of the Watch modeling language is its support for reuse of models and model elements. This is present at all levels, and across different watch families, logical watches, and applications. Reuse here means that two or more parts of a model refer to the same model element. Since this is a reference rather than a copy, updates to the reused element will instantly and automatically be in effect in all parts referring to that element. Reusing rather than copying thus reduces the amount of work needed for maintenance, reduces the risk of corrections only being made to one copy, and makes the amount of data in the models smaller and thus easier to understand.

At the Family level, the same Logical Watch can be reused in several Watch Models: for example, the Simple application for viewing and setting the time is used in both Celestron and Celestra watches. Similarly, the same Display can be reused in several Watch Models: for example, X334 is used in both Ace and Delicia. The Buttons and Icons used in Displays are each defined only once, and reused in each Display that has them.

At the middle level, each reference to a lower-level Watch Application is clearly a case of reuse. Since the name of a State in the middle-level diagram is the same as the name of the Watch Application it decomposes to, a State for a given Watch Application would be essentially identical between different Logical Watches. We thus reuse the whole Stopwatch State from the TASTW Logical Watch in the TASTand TST Logical Watches. Any improvements to the Stopwatch will thus instantly be a part of all Watch Models that use these Logical Watches.

At the bottom level, the Buttons and Icons are reused from their definitions at the Family level. The global pseudovariables such as sysTime are also defined only once and reused in many Watch Applications. This is in contrast with the true Variables and the Alarms, which are defined local to a given Watch Application. They may be reused within that one application, for instance first in a calculation that sets the value, and later in a calculation that reads the value.

Looking at the Watch Applications, both Timer and Stopwatch contain a variable called stopTime, but these refer to different things: a Timer’s stopTime is the time of day when its alarm will ring, whereas a Stopwatch’s stopTime is the amount of time that had elapsed when the stopwatch was stopped. Two Variables may thus share a name, but because of the scoping they are two different objects in the models, and two different variables in the running watch applications. The scoping is implicit in the modeling language, and made explicit in the code generation.

The same Display Function will normally be referred to by many States. In the simplest case, all States in a Watch Application will use its single Display Function. To make this as easy as possible for the modeler, such a Display Function can be given a blank name, and then each State that does not define a Display Function will use this default Display Function.

In the Watch Application diagrams we also take advantage of another kind of reuse: reuse by relationship. The clearest case is the reuse of Actions: an Action can be reused by more than one Transition, simply by having each Transition link to it with its Action role line. Another case, so obvious that it might be overlooked, is the reuse of States, which are reused within a single application in the sense that they may be reached by more than one path.

Since in some cases trying to link disparate objects together can lead to a visual spaghetti of crisscrossing lines, it is also possible to make representational duplicates of objects in a diagram. For instance, in the Time application in Fig. 9.8, it is possible to return from the EditMinutes, EditHours, and EditSeconds states to the basic Show state, via an ExitEdit state that makes the edited time persistent. Rather

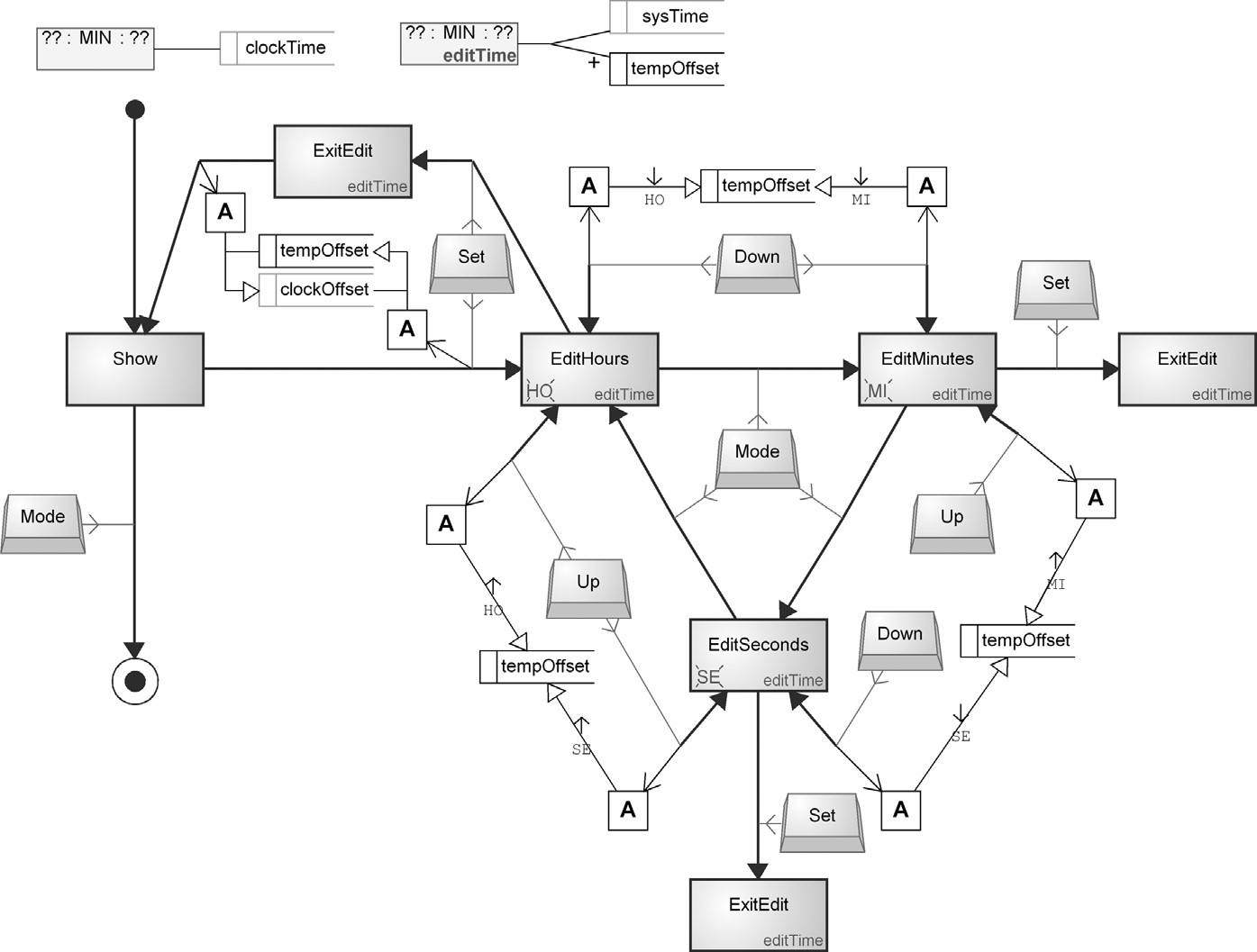


FIGURE 9.8 Reuse in the Time application

than try to draw three relationships all arcing back to the start, a duplicate of the ExitEdit state is placed conveniently close to each of the Edit states, allowing a single short transition for each. No transitions leave these duplicates, but the model knows they are all the same as the main ExitEdit state, so the transitions that leave there are applicable in the duplicates too. Conceptually we can think that we jump straight from a duplicate to the main ExitEdit state, but in fact there is no difference, and no one representation of the ExitEdit state is pre-eminent. The ExitEdit state simply has the sum of all of the transitions that enter or leave any of its representations in this graph. Since having real reusable objects like this may be too radical for more conservative users, it would also be possible to have a new object type “State Reference,” and use that for the other representations of ExitEdit. Of course, if the users are that conservative, you should just let them code the whole thing in assembly language and forbid the use of subroutines—or simply give them a nice analog watch as a retirement gift...

The Time application is also an example of a possible type of reuse that has not been implemented in this modeling language. There is clearly something similar in the States, Buttons, Transitions, and Actions for EditHours, EditMinutes, and EditSeconds. Each Edit State selects one time unit of the same time Variable, and the Up and Down Buttons roll this time unit up and down. A similar pattern is seen in other applications where a time variable is being set by the user, that is, Alarm and Timer. There, however, only the hours and minutes are being set, and there are differences between the three cases as to the Display Function used. Because of these differences, it was not possible to build a subapplication that each of these three applications could use.

We did, however, consider a new modeling language concept, TimeEditor, which would have stood for a set of related states like this. ATimeEditor instance would have specified the variable to be edited, the first and last time units to be edited, and the Display Function to be used. This information would have been sufficient to cover the variability between the three cases, and we could have built a domain framework function to handle the behavior, and made the generator create a simple call to that function.

With only three cases, however, and none of these being impossibly complicated with the existing modeling language, we decided to leave the extension until later. Rather than creating a new concept for this particular kind of repetition in the models, it might prove to be better to identify several kinds of repeated patterns and come up with a more general way to specify parameters to a subapplication. In the long run, this approach would probably have allowed more expressive power with a smaller set of concepts.

9.4.2 Watch Application Metamodel

For completeness, Fig. 9.9 shows the full metamodel for Watch Applications.

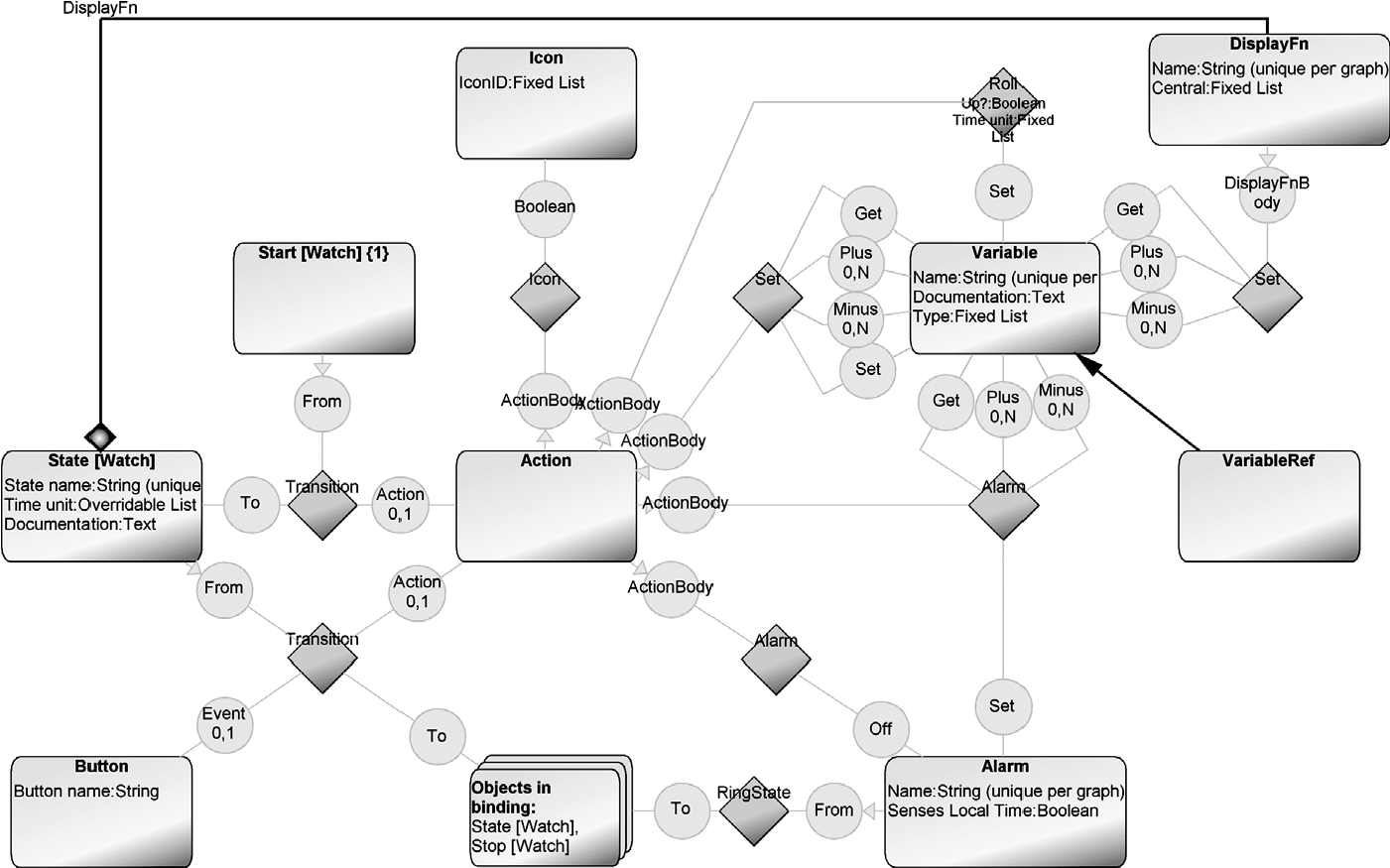


FIGURE 9.9 Watch application metamodel

9.5 CODE GENERATION FOR WATCH MODELS

Generators were built for watch models to both Java and C. We will look at each in turn, focusing first on Java.

9.5.1 Java Generator for Watch Models

In the watch example, the generator goes beyond simply producing Java code corresponding to the models: It also takes care of creating and running build scripts, and ensuring that the domain framework is available. Generation proceeds with the following steps:

1. The batch scripts for compiling and executing the code are generated. As the mechanisms of compilation and execution vary between platforms, the number and content of the generated script files depend on the target platform.
2. The code for the domain framework components is generated. This is an unusual step, since normally any domain framework code files would already be present on the developer’s local disk or referenced from a network share. For the purposes of the Watch example as part of an evaluation package, however, we had to provide the framework code files and make sure they were installed in an appropriate place. The easiest way to do this was simply to include them as textual elements along with the models. This way we ensured that all required components were available at the time of compilation, and we also had control over the inclusion of platform-specific components.
3. The code for logical watches and watch applications is generated. The state machinesareimplementedbycreatingadatastructurethatdefineseachstate,transition, and display function, along with their contents. For each action, the code generatorcreatesasetofcommandsthatareexecutedwhentheactionisinvoked.
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 was the code generator implemented then? Each generator in MetaEdit+ is associated with a certain modeling language and can thus operate on models made according to that specific language. To avoid having one huge generator, generators can be broken down into subgenerators and call each other, forming a hierarchy or network. The top level of the Watch DSM solution is the Watch Family modeling language, and this also forms the top level of the generator. The generators at this top level are shown in Fig. 9.10 (a “\*” in the name of a subgenerator denotes an individual version for each target platform). Arrows denote calls to subgenerators, and the order of execution is depth first and left to right.

At the top is the main generator, “Autobuild.” The role of “Autobuild” here is similar to that of the “main” function in many programming languages: it initiates the

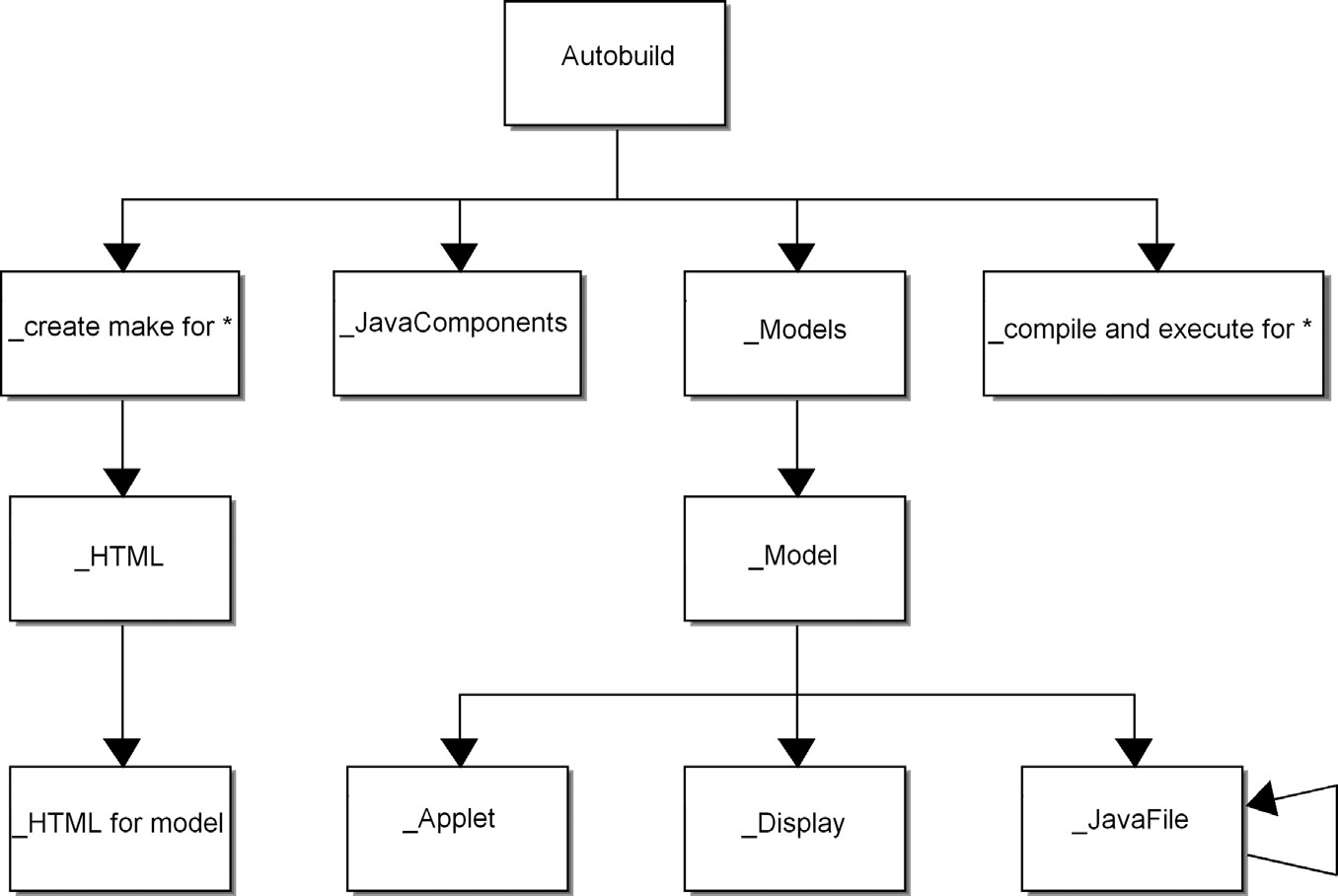


FIGURE 9.10 The watch code generator architecture, part 1

whole generation process but does not itself contain anything other than calls to the subgenerators on the next level down. The subgenerators on the next level relate closely to the steps of the Autobuild process presented earlier in this section. As “\_JavaComponents” just outputs the predefined Java code for the framework components, and “\_compile and execute \*” just executes scripts produced during the earlier steps of the generation process, we can concentrate on the more complex subgenerators, “\_create make for \*” and “\_Models.”

The basic task of the “\_create make for \*” subgenerators 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 subgenerator for each supported target platform. If there are any specific platformrelated generation requirements, for example creating the HTML files for the browserbased testenvironmentinFig.9.10,theycanbeintegratedintothe“\_createmakefor\*” subgenerator.

The responsibility of the “\_Models” and “\_Model” subgenerators is to handle 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 to create the specified user interface components, and the definition of the logical watch.

An example of the code generated for an applet (produced by the “\_Applet” subgenerator) is shown in Listing 9.1. The generated code defines the applet as an extension of the AbstractWatchApplet framework class and adds the initialization for the new class. Values from the model are shown in bold.

Listing 9.1 Generated code for an applet.

public class **Ace** extends AbstractWatchApplet { public **Ace**() { master=new Master(); master.init(this, new Display**X334**(), new **TASTW**(master));

}

}

The simple generator that produced this is shown in Listing 9.2, with the generator code in bold.

Listing 9.2 Code generator for the applet (generator code in bold).

**'**public class **' :Model name; '** extends AbstractWatchApplet { public **' :Model name; '**() { master=new Master(); master.init(this,

new Display**' :Display:Display name; '**(), new **' :LogicalWatch:Application name; '**(master))**;**

}

}**'**

The display definition can be generated in the same vein by the subgenerator “\_Display,” as shown in Listing 9.3. Again, a new concrete display class is created, inheriting from AbstractDisplay, and the required user interface components are defined within the class constructor method.

Listing 9.3 Generated code for a display.

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");

}

}

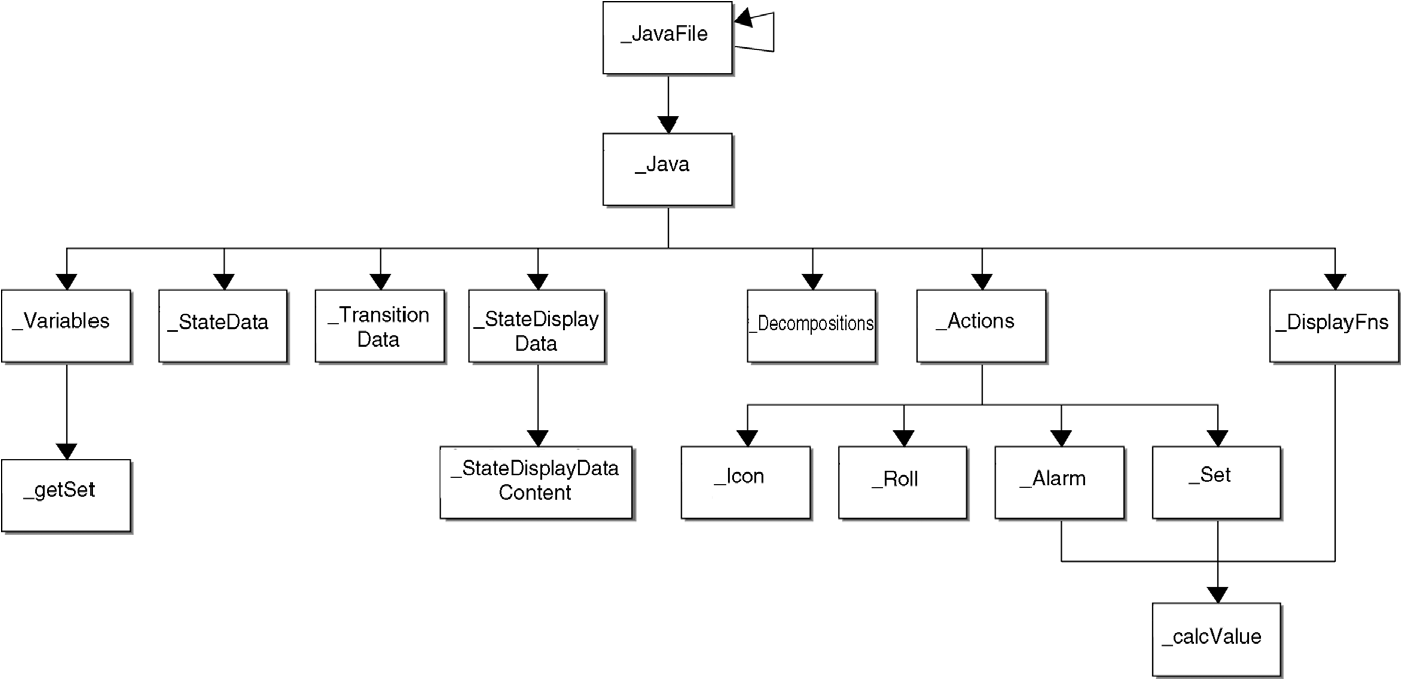


FIGURE 9.11 The watch code generator architecture, part 2.

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 subgenerators associated with the Watch Application modeling language) is shown in Fig. 9.11.

The subgenerators “\_JavaFile” (which is the same as in Fig. 9.10) 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 a hierarchical fashion, a recursive structure was implemented in the “\_JavaFile” subgenerator. During the generation, when a reference to a lower-level state machine is encountered, the “\_JavaFile” subgenerator will dive to that level and call itself from there.

The task of the “\_Java” subgenerator 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 9.4, which shows the implementation of the Stopwatch application.

For a comprehensive understanding of the “\_Java” subgenerator output, let us study the generated code line by line. As before, a new concrete Watch Application class is derived from the AbstractWatch Application class (line 1). From here on, the responsibility for the generated code is distributed among the

“\_Variables,” “\_StateData,” “\_TransitionData,” “\_StateDisplayData,” “\_Actions,” and “\_DisplayFns” subgenerators.

The “\_Variables” and “\_getSet” subgenerators 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” subgenerator produces lines 32–34, followed by the state (lines 35–38) and state transition definitions (lines 40–45) generated by the “\_StateData” and “\_TransitionData” subgenerators. The “\_StateDisplayData” and “\_StateDisplayDataContent” subgenerators then provide the display function Listing 9.4 The generated code for Stopwatch application.

1. public class Stopwatch extends AbstractWatchApplication
2. {
3. static final int a22\_3324 = 1;
4. static final int a22\_3621 = 2;
5. static final int a22\_4857 = 3;
6. static final int d22\_4302 = 4;
7. static final int d22\_5403 = 5;

8

1. public METime startTime = new METime();
2. public METime stopTime = new METime();

11

1. public METime getstartTime()
2. {
3. return startTime;
4. }

16

1. public void setstartTime(METime t1)
2. {
3. startTime = t1;
4. }

21

1. public METime getstopTime()
2. {
3. return stopTime;
4. }

26

1. public void setstopTime(METime t1)
2. {
3. stopTime = t1;
4. }

31

1. public Stopwatch(Master master)
2. {
3. {
4. super(master, "22\_1039");
5. addStateOop("Start [Watch]", "22\_4743");
6. addStateOop("Running", "22\_2650");
7. addStateOop("Stopped", "22\_5338"); 38 addStateOop("Stop [Watch]", "22\_4800");

39

1. addTransition ("Stopped", "Down", a22\_3324, "Stopped");
2. addTransition ("Running", "Up", a22\_4857, "Stopped"); 42 addTransition ("Stopped", "Up", a22\_3621, "Running");
3. addTransition ("Stopped", "Mode", 0, "Stop [Watch]");
4. addTransition ("Running", "Mode", 0, "Stop [Watch]"); 45 addTransition ("Start [Watch]", "", 0, "Stopped");

46

1. addStateDisplay("Running", -1, METime.SECOND, d22\_5403);
2. addStateDisplay("Stopped", -1, METime.SECOND, d22\_4302);
3. }

50

* 1. public Object perform(int methodId)
  2. { 53 switch (methodId)
  3. {
  4. case a22\_3324:
  5. setstopTime(getstartTime().meMinus(getstartTime()));
  6. return null;
  7. case a22\_3621:
  8. setstartTime(getsysTime().meMinus(getstopTime()));
  9. iconOn("stopwatch");
  10. return null; 62 case a22\_4857:

1. setstopTime(getsysTime().meMinus(getstartTime()));
2. iconOff("stopwatch");
3. return null; 66 case d22\_4302:

67 return getstopTime(); 68 case d22\_5403:

1. return getsysTime().meMinus(getstartTime());
2. } 71 return null;
3. }
4. }

definitions (lines 47 and 48), while the basic method definition and opening of the switch statement in lines 51–54 again come from the “\_Java” subgenerator.

The generation of the code for the actions triggered during the state transitions (lines 55–65) is a good example of how to be creative in integrating 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” subgenerator first provides the master structure for each action definition: a case statement using the unique ID of the Action. It then follows the relationships for each part of the action and executes the subgenerator bearing the same name as the current action relationship (either “\_Icon,” “\_Roll,” “\_Alarm,” or “\_Set”). The subgenerator produces an appropriate line of code for this part of the action, for example turning an icon on in line 60. This implementation not only reduces the generator complexity but also provides a flexible way to extend the watch modeling language later if new kinds of actions are needed.

Finally, the “\_DisplayFns” and “\_calcValue” subgenerators produce the calculations required by the display functions (lines 66–69). The “\_calcValue” subgenerator—which is also used by the “\_Alarm” and “\_Set” subgenerators— provides the basic template for all arithmetic operations within the code generator.

The generation of the simple middle-level Watch Applications such as TASTW proceeds in the same way. As the middle-level models are simpler than the applications they contain—for example, they have no actions—the resulting code is also simpler. In order to support the references to the lower-level state machines (i.e., to the Watch Applications of which the middle-level model is composed), the definitions of these decomposition structures must be generated. This is taken care of by the “\_Decompositions” subgenerator.

9.5.2 C Generator for Watch Models

Since the choice of Java and the highly specific state machine framework of the Java generator were not a familiar approach to most embedded developers, we later built a C generator. The generator, shown in Listing 9.5, is significantly shorter than that for Java and follows a completely different approach. The application’s States and Buttons are turned into enums, and the state machine is generated as two-level nested switch–case statements. The outer level has a case for each state the application is in, and the inner level for that state specifies the actions and transition for each button that may be pressed in that state.

Listing 9.5 The C generator for Watch Applications.

subreport '\_C\_Enums' run 'int state = Start; int button = None; /\* pseudo-button for buttonless transitions \*/

'

subreport '\_C\_RunWatch' run;

'void handleEvent() { int oldState = state; switch (state) {

'

foreach .(State | Start) { ' case ' id ':' newline

' switch (button)' newline

' {' newline

do ~From>Transition; { ' case ' if ~Event; then do ~Event.Button { id } else 'None' endif ':' newline do ~Action.Action { do ~ActionBody>()

{ ' ' subreport '\_C\_' type run

} }

do ~To.(State | Stop)

{ if not oid = oid;2 then

' state = ' id ';' newline endif

}

' break;' newline

}

' default:' newline

' break;' newline

' }' newline

} ' default:

break; }

button = None; /\* follow buttonless transitions \*/ if (oldState != state) handleEvent(); }'

This generator produces the familiar nested switch–cases used in much embedded software. The results for Stopwatch are shown in Listing 9.6.

Listing 9.6 The generated C code for the Stopwatch application.

typedef enum { Start, Running, Stopped, Stop } States; typedef enum { None, Down, Mode, Up } Buttons; int state = Start;

int button = None; /\* pseudo-button for buttonless transitions \*/ void runWatch() {

while (state != Stop)

{ handleEvent(); button = getButton(); /\* waits for and returns next button press \*/

}

}

void handleEvent()

{ int oldState = state; switch (state)

{ case Start:

switch (button)

{ case None: state = Stopped; break; default: break; } case Running:

switch (button)

{

case Up:

stopTime = sysTime - startTime; icon (0, stopwatch); state = Stopped; break;

case Mode: state = Stop; break; default:

break;

} case Stopped:

switch (button) { case Mode:

state = Stop; break;

case Down: stopTime = startTime - startTime; break;

case Up:

startTime = sysTime - stopTime;

icon (1, stopwatch);

state = Running; break; default:

break;

} default:

break;

} button = None; /\* follow transitions that do not require buttons \*/

if (oldState != state) handleEvent(); }

9.6 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 set of requirements for the framework to meet the needs of a complete DSM environment, we have to separate the domain-specific parts of the framework from the general platformrelated parts.

### THE DOMAIN 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 in which the watch example was originally written did not contain any useful service to handle timed events such as alarms. Thus, we implemented such a service ourselves as part of our domain framework. The more recent versions of Java, however, 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 here), Java programming languagewith AWT classes, and environment to test our generated code (e.g., a web browser with Java runtime). 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 in this way—is shown in Fig. 9.12 (solid line arrows indicate a specialization 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 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 provides the interface of the framework with the models by defining the expected code generation output, which complies with the code and the templates provided by the other levels.

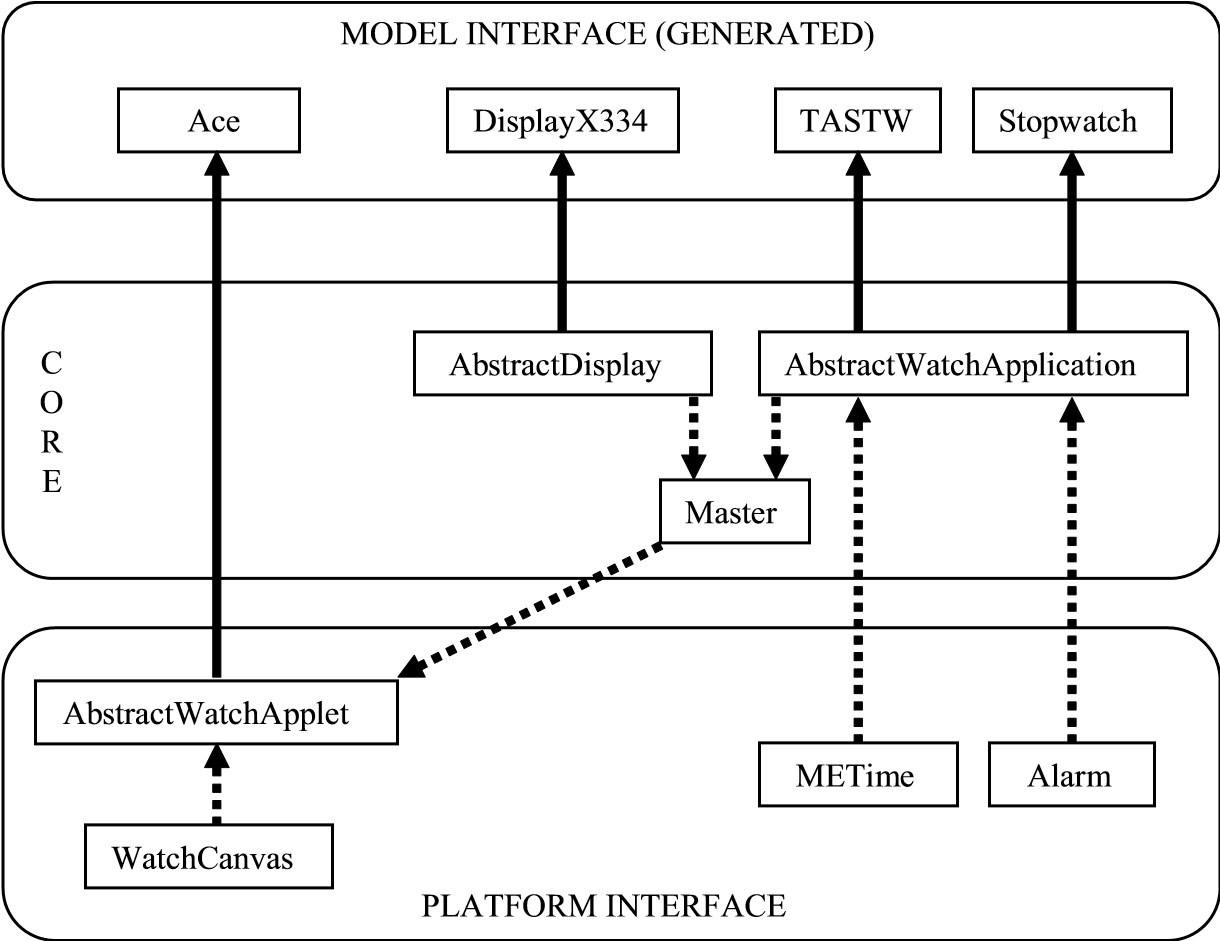


FIGURE 9.12 The watch domain framework

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 are defined using this interface. Similarly, METime makes up for the shortcomings of the implementation of date and time functions in 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 ensure that there is only one kind of target template the code generator needs to interface with. Initially, there was only one target platform: Java applets in a web browser.

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 Abstract WatchApplication 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.

9.7 MAIN RESULTS

Comparing the Watch models to the tangle of code that is normally found when similar embedded systems are hand coded, it is clear that the DSM solution helped build systems better here than is normal in the industry. However, looking at the generated applications, it is equally clear that the code produced is shorter and in some ways simpler than would normally be written by hand. How much then is the improvement due to DSM actually only due to the greater attention spent on developing a good framework?

To gain some insight into this, we devised an experiment. Subjects extended the Stopwatch application to add lap time functionality, first by DSM with code generation, and again manually by editing the original Stopwatch Java code in the same architecture. While the sample size was too small to be statistically significant, the results in Table 9.1 may still be interesting.

The necessary changes to add a lap time function were roughly eight lines of Java code or eight objects in the model. For a senior developer, the productivity for

MAIN RESULTS TABLE 9.1 Productivity Improvements of DSM Modeling versus Manual Coding

Modeling

Coding

Modeling:Coding

Productivity Ratio

Time (s)

Functions/hr

Time (s)

Functions/hr

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Senior developer | 38 | 94.7 | 200 | 18.0 | 5.2 : 1 |
| Junior developer | 160 | 22.5 | 639 | 5.6 | 4.0 : 1 |
| Total |  | 117.2 |  | 23.6 | 5.0 : 1 |

modeling was over five times that for coding. In the case of a junior developer, the difference was only four times, but for the combination of both developers the productivity for modeling was five times that for coding.

As is often found, the senior developer was three to four times more productive than the junior developer, and this difference was evident whether modeling or coding. However, the productivity of a junior developer modeling was greater than that of a senior developer coding—a tantalizing prospect for project managers. Imagine all of your developers being 25% more productive than your top developer is currently!

To return to our original question, it is now clear that the main benefit of DSM is in the difference between modeling and coding. While building a good framework to support DSM is useful, hand coding on top of even a good framework is still four to five times slower than DSM.

9.7.1 Extending the DSM Solution to New Platforms

Initially, therewas only one target platform for the Watch applications: Java applets in a web browser. With the advent of Java applications on mobile phones another possible platform appeared. While it is still Java, the new platform, MIDP, differed radically in its user interface possibilities. Text fields to show time digit pairs were replaced by a bitmap display, and mouse and keyboard input was replaced by simple soft keys and cursor keys. In addition, the main Applet class was replaced by a Midlet class, and MIDP Java did not support reflection, which we had used in the code generated for Actions and Display Functions. Finally, the build process for compiling and packaging a MIDP application required a number of extra steps and several new files.

In spite of these differences, extending the Watch DSM solution to support this new platform was possible mainly by simply building versions of the AbstractWatchApplet and WatchCanvas classes for the phone MIDP framework. Only minor changes were necessary to the code generation, to work in the more restricted MIDP environment. No changes were necessary to the modeling language or models. 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.

Later, the generators and framework were further extended to take advantage of the MetaEdit+ API to provide visual tracing of model execution. As the watch application ran, it would make WebServices calls back to MetaEdit+ when control passed to a new state, and MetaEdit+ would highlight that state. Again, these changes only affected the framework and generators.

The DSM solution thus insulated Secio from changes in the implementation environment, offering good 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 probably be no hope of maintaining one code base for all platforms.

9.8 SUMMARY

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 5 days to develop the Java framework, 2 days for the modeling language, and 1 day for the code generator. Another day or two was then spent adding support for multiple Watch Families and creating the full set of example models. These times include design, implementation, testing, and basic documentation.

Calculating the return on investment for these times is not possible without more information about Secio: what their current code framework was like, how many watch models they wanted to produce, and so on. An estimate for the latter can be found from the fact that for a real digital watch manufacturer, Casio, their current portfolio for the United Kingdom alone contains over 220 different watches (not including variants differing only in color or materials).

### PART IV

## CREATING DSM SOLUTIONS

In this part, we teach you how to create the various parts of a DSM solution (Chapters 10–12), discuss the processes and tools for creating and using the DSM solution (Chapters 13–15), and wrap up with a summary and conclusions in Chapter 16. The examples from Part III are often used in explaining the principles discussed here, so you would be advised to at least skim those examples first.

Creating thevarious parts of a DSM solution may be the responsibility of one person, or then the task may be split over two or more people. Chapters 11 and 12 on the generator and domain framework, and to some extent Chapters 14 and 15 on tools and usage of DSM, will make most sense to experienced programmers. Chapters 10 and 13 on DSM language creation and processes may interest those in more of an architect or project management role.