Chapter 13 - Processes and their coordination, additional UI topics

Overview

Up to now, we were developing applications that ran until completion, run for a while, switch to another application, run the other application, return to the original application, and so on. However, there are many applications that should run while other applications run, automatically taking turns with them, using their share of CPU time when they need it and returning control to other applications when they don't need it any more. Some examples of this are network communication (the network communication program must be ready to run at any time and give up control when it does not need the CPU any more), printing while another program is running, and getting data from the web while the user is doing something else. Simpler examples include a computer clock that displays time without interfering with other programs and losing seconds, a stopwatch running in parallel with other programs, and alarms that run in the background, marking time and producing screen notification at a time previously specified by the user.

There are even applications that require that several of their own components run 'simultaneously'. As an example, it is not unreasonable to design a simulation of an airport with multiple airplanes landing, taking-off and circling the airport in such a way that each airplane behaves as if it had its own computer supporting it. And the Smalltalk environments itself has several 'applications' running at the same time - if we consider, for example, the independent process of garbage collection as an 'application'.

All of these examples use the principle that a CPU does not have to be dedicated to a single program or a single thread within one program, but can switch between any number of programs or threads of execution in an organized way, giving the appearance that they all run independently of one another and in parallel. In computer science terminology, these independent threads of execution are called processes and this chapter gives an introduction of their implementation in VisualWorks Smalltalk.

Please note that this chapter is not a complete treatment of processes but rather an introduction that illustrates the concept on several intuitive problems.. We also use our examples to introduce some new aspects of the construction and control of custom UI components.

13.1 A stopwatch and the concept of a Process

Some very natural programming problems require splitting the use of the CPU among several threads of tasks. The following example illustrates one such a situation.

<u>Example: Analog and digital stopwatch</u>

Problem: Design a stopwatch with analog and a digital components as in Figure 13.1. Clicking *Start* starts the stopwatch, updating both the analog and the digital part in one-second increments. Clicking *Stop* freezes the display, and clicking *Reset* resets the display to 0 seconds. The stopwatch must have minimal effect on the operation of the rest of the environment in that it should be possible to run other applications while the stopwatch is running, and running another application should not stall the stopwatch.

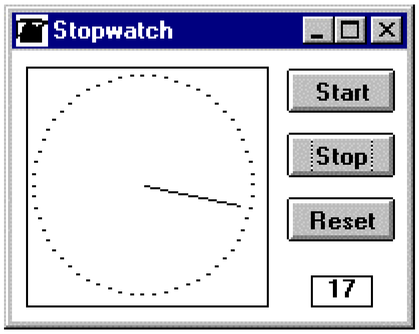


Figure 13.1. Stopwatch example.

Solution: The principle is straightforward: When the user clicks *Start*, the program begins an infinite loop that increments time every second and updates the display. *Stop* and *Reset* have obvious effects. The solution will require only an application model class (StopWatch) and a view class for the display of the analog watch (WatchView). Since the user cannot directly interact with the analog display, the subview does not need an active controller.

Although the principle is simple, the new problem is that the program must act independently of other programs.

To start, we painted the canvas and assigned the following properties: time is the Aspect of the input field showing digital time, watch is the Aspect of the view holder, and start and reset are the names of the action methods. We then proceeded to design and implement the two classes as described next.

Class WatchView

As a view class, WatchView requires a displayOn: method to display the analog face and handle damage repair. Updating will use dependency and method update:.

Method displayOn: will draw the seconds marks around the face of the clock, and the hand. The hand (but not the marks) must also be redisplayed when time changes and when the user clicks *Reset*. Displaying the hand in a new position requires erasing the current hand, and drawing a new one in the new position. Erasing can be done by redrawing the hand in the color of the background. We will implement this mechanism by four methods: two low-level methods erase: and draw: that draw and erase the hand, and two high-level methods move: and reset: that 'move' the hand to a specified second or to the starting position respectively. All four will take integer time in seconds as its arguments, and move: and reset: will combine erasing the hand in its current position and redrawing it in the new position. Method move: will be used during regular stopwatch operation, reset: for resetting.

Let's analyze the details of drawing. The marks around the face are drawn in one-second intervals. Since there are 60 seconds in a minute and one minute corresponds to a full circle on the stopwatch, mark locations are separated by $2\pi/60 = \pi/30$ radian increments. The point at which the mark is drawn is best obtained by using polar coordinates - the radius (constant) and the angle (60 increments of $\pi/30$ each). The angle of the hand is calculated similarly - by multiplying integer seconds by $\pi/30$.

We must now consider how to implement this operation efficiently because we want the display to take as little time as possible, partly to eliminate flashing, and partly to take away as little CPU time as possible from other running programs. Calculating π divided by 30 every time we want to display a mark or the hand is obviously unnecessary since we can pre-calculate this constant and cache it in an instance variable. Another constant can also be pre-calculated: The starting point of the stopwatch (0 seconds) is at the top of the circle, in other words, at - 90 degrees with respect to the x axis. 90 degrees corresponds to one quarter of a circle, in other words $2\pi/4 = \pi/2$. We will calculate this value during initialization and save it in another instance variable. We also need a variable to hold the coordinates of the center, the length of the hand, and the radius of the circle of the marks around the face. These variables must all be initialized before we can execute the displayOn: method.

Considering that the center of the subview can only be calculated when the subview has been created (either before or after the window opens), we will trigger this initialization from the postBuildWith: method in the StopWatch application model; it would not work if triggered from initialize which is executed before the subview is built. We will call this initialization method setParameters and its definition in the subview class WatchView is as follows:

```
setParameters
"Precalculate essential parameters of stopwatch geometry."
| bounds |
        bounds := self bounds.             "Obtain square containing the subview."
        center := bounds center.
        radius1 := bounds width - 20/2.    "End point of hand."
        radius2 := radius1 + 6.            "Circle on which marks are drawn around the face."
        const2 := Float pi / 2.            "90 degrees in radians – corresponds to 15 seconds."
        const1 := const2 / 15              "Separation of seconds in radians."
```

Method displayOn: which redraws the whole face of the digital clock with the marks and the hand is as follows:

```
displayOn: aGraphicsContext
"Display markings and hand."
| theta |
	"Draw marks around the face."
	theta := 0. "Angle at which the next mark will be drawn."
	1 to: 60 do: [:sec| | endPoint |
				theta := theta + const1. "Increment by π/30."
				endPoint:= center + (Point r: radius1 theta: theta).
				aGraphicsContext displayDotOfDiameter: 3   “Seconds mark.”
								at: endPoint].
	self drawHand: model time value. "Draw hand, getting seconds from the application model."
```

Note that we used addition to calculate theta because it is faster than multiplication The low-level drawing method draw: displays the hand by drawing a straight line from the center of the face to the endpoint at an angle given by the time:

```
draw: integerSeconds
"Draw hand in position corresponding to seconds in integerSeconds and update input field."
| endPoint |
endPoint:= center + (Point r: radius1 theta: integerSeconds * const1 - const2). "Angle corrected by 90°."
self graphicsContext displayLineFrom: center to: endPoint.
model time value: integerSeconds    "Display digital value in input field."
```

Method erase: is very similar and the only differences are that it changes the paint of the graphics context to that of the background (to make the old hand invisible) and that it does not rewrite the input field. We found how to get the background color of a widget in the Cookbook.

```
erase: anInteger
"Erase hand in position corresponding to seconds in anInteger."
| backgroundColor endPoint |
	backgroundColor := (model builder componentAt: #watch) lookPreferences backgroundColor.
	endPoint:= center + (Point r: radius1 theta: anInteger * const1 - const2).
	(gc := self graphicsContext)
		paint: backgroundColor;
		displayLineFrom: center to: endPoint
```

Note that if we were sure that we always wanted stopwatch background to have the same color, for example white, we could assign it directly as ColorValue white. The more general approach that we have used will work even if we change the background color property of the widget.

As we already said, the high level drawing and resetting is done by move: and reset:. The move: method first erases the old hand and then displays the new hand in the new position:

```
move: anInteger
"’Move’ hand to position defined by anInteger."
	self erase: anIteger - 1;
	     draw: anInteger
```

The reset: method erases the hand at the old position (argument) and redraws it at 0 seconds as follows:

```
reset: anInteger
"Reset hand from position defined by anInteger to start position."
	self eraseHand: anInteger;
	     drawHand: 0
```

The only remaining WatchView method is update:. This method is executed when the model sends itself changed: and this happens when the user clicks *Reset*, and inside the infinite loop when time increments. Each of these two behaviors is different and we must thus use changed: with a symbolic argument to trigger the proper response. We must also include an argument representing the time from which the stopwatch is being reset (in the case of reset:) or the new time in regular operation (method move:). If we use #reset: or #move: as the symbolic argument, the definition can be as simple as follows:

```
update: aSymbol with: integerSeconds
"Update the hand in a way appropriate for the current context represented by aSymbol."
	self perform: aSymbol with: integerSeconds
```

This will require that the model must send changed:with: with arguments #reset: or #move:, and time.

Class StopWatch

During the start up of the application, we must initialize the *View* variable to a new instance of WatchView, set its model, and initialize the input field aspect variable.

```
initialize
	watch := WatchView new.
	watch model: self.
	time := 0 asValue		"The type property of the input field is number."
```

We have already mentioned that when the builder is finished building the window, we must ask the subview to initialize its parameters in postBuildWith:

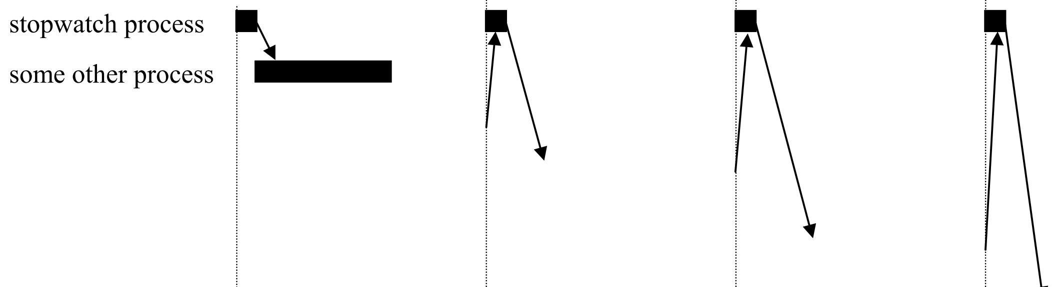
```
postBuildWith: aBuilder
	watch setParameters
```

where setParameters is the WatchView method defined above. Except for accessing methods, it now only remains to write action button methods. The *Start* button opens an infinite loop in which it creates a one-second delay, updates time, and sends itself the changed:with: message with appropriate parameters. This is easy except for two things: we must be able to stop the loop by clicking the *Stop* button, and it must be possible to run another program when the stopwatch is active but not running. In the second case, the stopwatch program must wrestle control back when it needs to update.

In essence, what we need is to package the iteration that triggers the watch motion as a separate program, store access to this separate program in an instance variable, and use this instance variable to 'kill' the program from the *Stop* action method. Separately packaged and independently running programs are called *processes* (more on this shortly), and we will thus use the name process for the name of the instance variable referring to this separately running block. The whole definition of start is as follows:

```
start
"User clicked Start. Start a new separately running unending process and create a reference to it."
	process := [(Delay forSeconds: 1) wait.
			time value: time value + 1.
			self changed: #move: with: time value] repeat] fork
```

As you can see, VisualWorks' style of creating a process is to send the fork message to a block whose contents is the program corresponding to the process. This schedules the block for evaluation (more on this later) and makes it accessible as a process. Our particular block (process) runs only long enough to execute the Delay statement which stops the process for one second - allowing any other waiting independent program (process) to start running (Figure 13.2). (This description of the operation of processes is simplified and we will give a more accurate description later.)



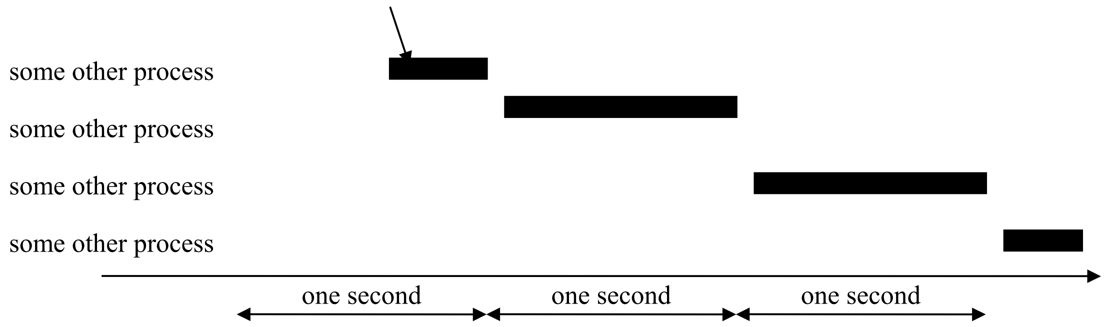


Figure 13.2. Simplified behavior of processes: Stopwatch runs until stopped by Delay. At this point another process can start execution and the stopwatch resumes when the one second delay expires.

The *Stop* button stops the infinite iteration process started by the *Start* button by sending it the terminate message:

```
stop
"Terminate the process implementing continuous stopwatch operation."
	process terminate
```

By the way, stop does not send changed:with: because it does not have any effect on the display - it just freezes the iteration. Action method reset:, on the other hand, sends changed:with: with #reset: and the current value of time because it must erase and redisplay the hand. The time argument is used to determine where the hand to be erased is:

```
reset
"Use dependency to reset stopwatch display to 0."
	self changed: #reset: with: time valu
```

where the 0 argument of with: is irrelevant because the message eventually executes reset in WatchView which resets to 0 anyway.

The program is now complete and ready to run! It works almost perfectly and allows us to execute short messages while the iteration is delayed. There is, however, one problem: If we start executing a longer message while the stopwatch runs, the display will freeze and continue incrementing only when the other program stops. To check this out, start the stopwatch, type

```
[3 factorial] repeat
```

into a Workspace, and execute it. The stopwatch will complete its current display, the factorial loop will jump in, and the stopwatch will never again get a chance to update. This behavior violates the part of the specification that states that execution of another program should not affect the stopwatch. Let's explore what is happening and how we can correct it.
When we start the stopwatch, the block

```
[(Delay forSeconds: 1) wait.
time value: time value + 1.
self changed: #move: with: time value]
```

starts executing and stops when it reaches

```
(Delay forSeconds: 1) wait
```

At this point, another program (such as our factorial loop - call it X) can take over. If this happens, we now have two programs competing for the CPU - the stopwatch, and program X. In computing terminology, these two independent programs are processes.

Roughly speaking, Smalltalk processes are designed in such a way that if one process runs, all other processes must wait until it is completed. [1]This arrangement is acceptable in most but not in all situations and our case is one of the exceptions: Since the stopwatch must not get behind, its process must be able to execute whenever it needs to update time and this means that the stopwatch process must have priority over other processes. If another process with lower priority is executing when stopwatch needs to update, the stopwatch process must assume control and execute. This is indeed possible because VisualWorks implements the concept of process priority and allows us to assign any one of 100 different priorities to any process.

With the concept of *process priority*, we can describe the operation of Smalltalk processes more accurately as follows: A process runs to completion, unless another process with higher priority wants to gain control. If this happens, the higher priority process becomes active and the lower priority process yields and waits until the higher priority process terminates or yields control. (In the stopwatch infinite loop, this happens when the process executes the Delay expression). This principle is illustrated in Figure 13.3.

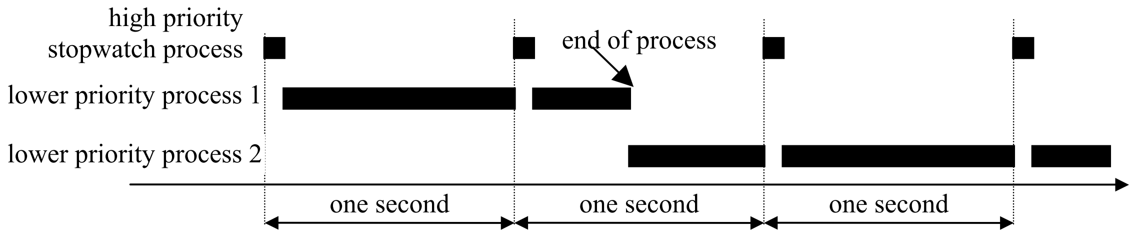


Figure 13.3. Processes and priorities: When a high priority process yields, lower priority processes can start execution. When a higher priority process requires processing, lower priority processes yield.

As we said, VisualWorks distinguishes 100 different levels of priority. Priority 1 is the lowest and 100 the highest. To maintain consistency with previous releases and to provide protection from changes of numbering in new releases, some priorities used by the system have names and can be assigned by sending the appropriate message to Processor. These named priority levels should be used whenever possible and their names and uses are as follows:

Priority number	name (method)	Purpose
100	timingPriority	Processes dependent on real time.
98	highIOPriority	Critical I/O such as network input.
90	lowIOPriority	Normal I/O such as keyboard input.
70	userInterruptPriority	High priority user tasks such as window management.
50	userSchedulingPriority	Normal user programs – default assigned by VisualWorks.
30	userBackgroundPriority	Background user processes.
10	systemBackgroundPriority	System background processes.

As an example, priority 50 can be assigned as

```
Processor userSchedulingPriority
```

although this particular priority normally does not have to be assigned because it is the default assigned when a user program starts executing. A user program with higher than normal priority - such as our stopwatch which should be able to interrupt other user programs - should thus run at *user interrupt*

[1] Process designs that do not allow other processes to run until the running process stops are called *non-pre-emptive*. In spite frequent statements to the contrary, we will see that VisualWorks processes are pre-emptive because a running process can be interrupted by a higher priority process.

priority or at *timing priority* if the displayed time is really critical and must not suffer from any interruptions at all. We will run our stopwatch process at *user interrupt priority*.

In our modified start method, we want to create the stopwatch process, assign priority to it, and schedule it immediately. The simplest way to do this is to use forkAt: aPriority instead of fork as follows:

```
forkAt: Processor userInterruptPriority
```

The only change in our program is thus in process creation which now becomes

```
start
"Start a new separately running unending process and create a reference to it."
        process := [Delay forSeconds: 1.
                    time value: time value + 1.
                    self changed: #move: with: time value] repeat] forkAt: Processor userInterruptPriority
```

When you now start the stopwatch and execute, for example,

```
[Transcript show: 'test'; cr] repeat
```

the stopwatch will continue running without interruption and the Transcript will continue printing, being interrupted (imperceptibly) by the higher priority stopwatch whenever its Delay expires (Figure 13.4).

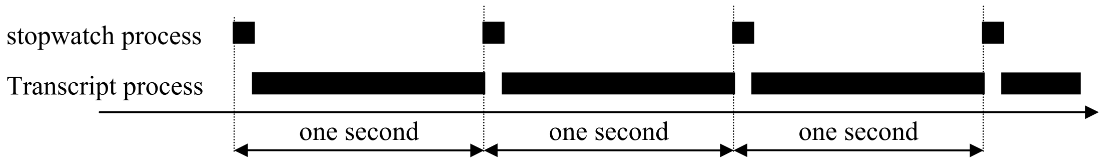


Figure 13.4. Higher priority stopwatch process taking turns with a lower priority process.

After this introduction to processes, we are now ready to take a closer look at their role and operation in VisualWorks.

Processes

Let's summarize what we have learned about Smalltalk processes so far: A process is a standalone thread of execution that can run on the same CPU and in the same Smalltalk image independently of other Smalltalk programs. Processes can be assigned priorities between 1 and 100 and at any given time, the highest priority process that is ready for execution runs. If a higher priority process requires processing, the running lower priority process is interrupted and its execution resumes when its priority becomes the highest waiting priority.

To create a process, send a process creation message to a block. We have introduced two process creation messages - fork and forkAt:. The first creates a process with the same priority as its parent process (the process from which it was forked) and starts its execution. The second assigns the process the specified priority and starts its execution if appropriate.

Creation of Smalltalk processes using fork essentially evaluates a block and message fork is thus quite similar to message value.[2] There is, however, an essential difference the two messages: The value message immediately evaluates its block and the message following the block must wait until block evaluation is finished. As an example,

```
[3 timesRepeat: [Transcript show: 'test']] value.
Transcript show: 'end of test'
```

prints

[2] Just as there are variations of value that send one or more arguments to a block, there is also a process creation message that allows forking a process with any number of arguments

```
test
test
test
end of test
```

This is not so for forking processes. As an example,

```
[3 timesRepeat:
        [Transcript show: 'test']] forkAt: Processor userBackgroundPriority.
Transcript show: 'end of test'
```

creates a *low priority* process, delays its execution until the parent process (the encompassing program) ends execution, executes the last line (thus ending the parent process), and if there are no other higher priority processes waiting, execution now returns to the suspended lower priority process and executes it. The code will thus print

```
end of test
test
test
test
```

in the Transcript. In fact, we will get the same result even if the new process does *not* have lower priority than the parent process as in

```
[10 timesRepeat:
        [Transcript show: 'test']] forkAt
Transcript show 'end of test'
```

because the new process will be placed in a queue behind the parent process.

The only fundamental question that remains unanswered is who manages the processes and their priorities. Implementing process management is the role of class ProcessorSheduler which schedules the order in which the processor (the CPU executing the image) executes all existent processes. Since a single Smalltalk image needs only one scheduler, an image has only one instance of ProcessorSheduler and its name is Processor. Processor is a global variable like Smalltalk.

To close this section, we will now examine how Processor handles processes starting with a look at class Process. Class Process has several instance variables but for our purpose the most important ones are priority (an Integer between 1 and 100, naturally), and myList. Variable priority is, of course, process priority, and myList is a list holding all processes of the same priority in the order in which they will be activated after this process. When a new process with this priority is created, it is added to the end of this queue. This is why the forked process in our last example executed after its parent.

Normally, the first process in the queue must execute to completion before the process following it in the queue can start executing. One can, however, suspend a process (blocking its operation until it gets a receive message) or send it to the back of the queue by executing

```
Processor yield
```

To force a process that has not yet terminated to stop and to remove it from the queue, send it the message terminate. This is what we did in our stopwatch program - the stopwatch process runs in an infinite loop and the only way to stop it is to terminate it. Finally, for each priority, Processor (the single instance of ProcessorScheduler) keeps an array holding all process queues and uses it to determine which process to execute next.

This completes our introduction to processes and we will give further examples of their use and explain additional concepts in the remaining sections of this chapter. One important point that we have not covered at all is the coordination of processes. The point is that processes sometimes depend on one another and one cannot proceed before another completes some operation. As an example from the real world, consider the following somewhat unusual arrangement of traffic lights at a crossing of a road and a railroad designed to prevent the possibility of a collision: When a train or a truck arrive at the crossing, they change the other vehicle's traffic lights to red to prevent collision. Upon leaving the crossing, the vehicle changes the other vehicle's traffic lights back to green. We will write a computer simulation of this arrangement in Section 13.3 and show how the desired coordination can be achieved by using the concept of a semaphore.

Main lessons learned:

- A process is a block of code – a program that shares the CPU processor and other resources with other programs under the control of a processor scheduler.
- Non-pre-emptive process scheduling lets a process run until it terminates or until it suspends its execution.
- Smalltalk processes are instances of class Process; they are pre-emptive because an available higher priority process interrupts a lower priority one..
- Three important Process messages are suspend, resume, and terminate. Message suspend takes its receiver out of the queue of runable processes but does not destroy it. Message resume puts a suspended process at the end of the queue of runable processes, and shuts down and destroys the process.
- Smalltalk processes have priorities between 1 (the lowest) and 100. By default, user processes run at priority 50, I/O processes such as mouse and keyboard events run at priority 90, real-time processes run at priority 100, and system background processes run at priority 10.
- To create a process, send fork or forkAt: priority to a block. This puts the new process at the end of the queue of runable processes of the same priority and schedules it for execution.
- Smalltalk processes are managed by Processor, the single instance of class ProcessScheduler.
- Processor always selects for execution the first process in the highest priority non-empty queue of runable processes.

Exercises

1. In our reset method, we used reset: time value. Shouldn't this be reset: 0?
2. Modify the stopwatch program to eliminate any effect of clicking *Start* or *Reset* while the stopwatch is running.
3. Add numeric time marks next to 0, 15, 30, and 45 second ticks.
4. Reimplement the stopwatch by creating an array of 60 hands, one for each possible hand position and each capable of drawing itself in a specified color. Then implement drawing and erasing by redrawing the current hand object in the background color and displaying the new one in the foreground color. Comment on the relative speed of both approaches.
5. Generalize the stop watch design by letting the creation method specify both the foreground and background colors.
6. Develop a digital version of a minute/hour watch.
7. Develop a Watch application with an analog face for hours and minutes.
8. Combine StopWatch and Watch into one application showing hours, minutes, and seconds.
9. Create other user interfaces for digital or analog hours, minutes, and seconds, and allow the user to select any desired combination.
10. Combine stopwatch, watch, and calendar into a new time machine application/widget.
11. Redefine the background paint of the stopwatch to a nice inconspicuous *pattern*.
12. Assume that an application can get blocked because a process that it is running can be suspended. Assume that the user interface contains an action button designed to restart the application. Since the button is a part of the application, the button cannot be used to unlock the process. True or false?
13. Assume that a forked off process has higher priority than the parent process. Will the parent process run to completion before the child process starts executing?

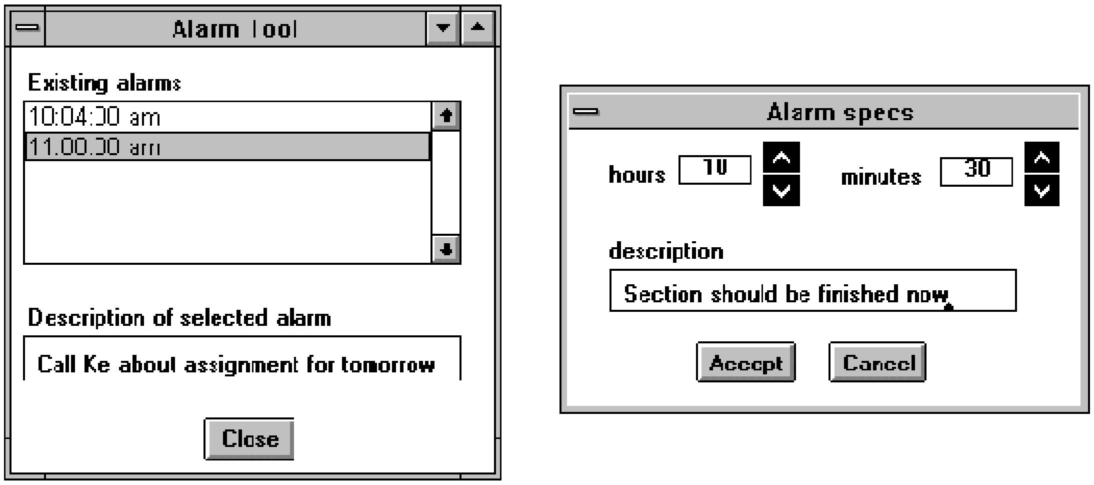
13.2 Alarm Tool

After implementing a stopwatch, the obvious next candidate in our exploration of time machines is an alarm clock. Since a flexible implementation of automated alarms opens some interesting new questions and helps to explore the concept of processes, we will dedicate this section to the design of a simple application allowing the user to create, edit, and delete any number of alarm notifications that pop up on the screen when their time arrives.

Specification

Develop a tool for creating, modifying, and deleting alarm events ('alarms', for brief), each characterized by expiry time and a brief description. The user can create any number of alarms at any time and in any order, and they will mature in the order of their expiry time, independently of the order in which they were created. When an alarm matures, it opens a notifier displaying its description. If two or more alarms mature at the same time, their notifiers are displayed in the order in which they were created. The operation of alarms must not interfere with user programs running at default priority, and operation of user default priority programs must not interfere with the alarms.

The desired user interface is as in Figure 13.5 and its behavior is as follows: The list of unexpired alarms has a pop up menu with command *add*. When an alarm is selected is the list, the pop up menu displays additional commands *edit* and *remove*. When the user selects *add*, the program opens the form shown on the right and when the user completes the form and clicks *Accept*, the new alarm is added to the list at a place corresponding to its maturity time. The window opens with current hours and minutes and does not accept time earlier than `Time now`. Clicking *Cancel* closes the form and aborts the operation. Editing uses the same form but the form opens with the parameters of the selected alarm. An attempt to remove an alarm elicits confirmation from the user.



10:04:30 am
11.00.30 am

Call Ke about assignment for tomorrow

hours 10 minutes 30

Section should be finished now

Figure 13.5. Main Alarm Tool window (left), form for creating and editing alarms (right).

The first iteration of design and implementation described below is intended as a test of the concept and implements the required functionality in a minimal way and further expansion is proposed as a series of exercises. The reason for this progressive enlargement of functionality is that the problem opens unfamiliar questions and we don't want to get bogged down in too many new questions at once.

Scenarios

For lack of space, we will leave out the more obvious scenarios and restrict ourselves to the more challenging ones.

Scenario 1: User adds an alarm to a non-empty list

1. *User* clicks *add* in the list pop up menu.
2. *Program* opens form.
3. *User* enters information and clicks *Accept*.
4. *Program* closes form, inserts new alarm in the list, and displays the updated list in the main window.

<u>Scenario 2: User edits an alarm in the list</u>

1. *User* selects an alarm and clicks *edit* in the list pop up menu.
2. *Program* opens the form, showing all parameters of the selected alarm. All fields are editable.
3. *User* edits the form and clicks *Accept*.
4. *Program* closes the form and changes the list to show the new alarm.

<u>Scenario 3: User deletes an alarm</u>

5. *User* selects an alarm and clicks *remove* in the list pop up menu.
6. *Program* displays a confirmation dialog.
7. *User* confirms.
8. *Program* deletes the alarm and changes the list to show the new alarm.

Context Diagram

The specification is so simple that there is no need to draw the Context Diagram.

Preliminary design

This application can be implemented with a single class - the application model (to be called AlarmTool). We don't even need to develop custom view classes because the user interface uses only standard widgets and windows. For cleaner design, however, we will add a class called Alarm to represent individual alarm objects. The preliminary description of our two classes is as follows:

- AlarmTool manages the Alarm Tool user interface, allows the user to create, edit, and delete alarms, and schedules and terminates alarms.
- Alarm holds all information about a single alarm.

Class level scenarios

Scenario 1: User adds an alarm to a non-empty
1. *User* clicks *add* in the list pop up menu.
2. AlarmTool opens a form (a modal dialog window) with default parameters.
3. *User* enters information and clicks *Accept*.
4. AlarmTool closes the form.
5. AlarmTool creates new alarm process with specified parameters and schedules it.
6. AlarmTool creates new Alarm with maturation time, description, and process created in the previous step.
7. AlarmTool adds the new Alarm to its collection and displays the new collection list ordered by alarm maturation time.

Scenario 2: User edits an alarm in the list
1. *User* selects an alarm and clicks *edit* in the list pop up menu.
2. AlarmTool terminates the process associated with the selected Alarm.
3. AlarmTool opens a form with parameters of the selected alarm.
4. *User* edits the form and clicks *Accept*.
5. AlarmTool closes the form.
6. AlarmTool deletes the selected alarm from its collection.
7. AlarmTool creates a new alarm process with specified parameters and schedules it.
8. AlarmTool creates a new Alarm with maturation time, description, and process created in the previous step.
9. AlarmTool adds the new Alarm to its collection and displays the new list.

Scenario 3 is similar to Scenario 2 and we leave it as an exercise.

Preliminary specification of class responsibilities

The class scenarios and the implied behaviors suggest the following responsibilities:

Alarm

Responsibilities	Collaborators
• Know and provide access to maturation time, description, and scheduled process.	

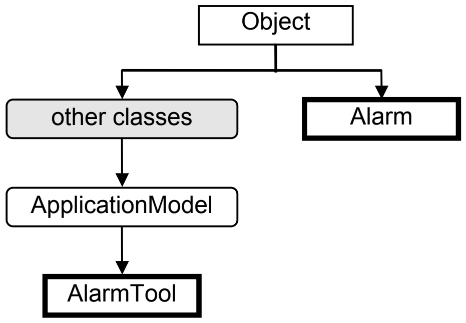
AlarmTool

Responsibilities	Collaborators
• Create and manage user interface including the main window and the form.	
• Maintain list of alarms in order of maturation.	Alarm
• Schedule alarms according to their maturation times.	

Design Refinement

Class Hierarchy

AlarmTool is an application model, and its superclass is therefore ApplicationModel. Alarm does not have any relatives in the class library and it will thus be a subclass of Object (Figure 13.6).



13.6. Hierarchy Diagram.

Specification Refinement

Alarm
This class is a simple data model and requires only accessing methods. There is no need for further details.

AlarmTool
The first two groups of responsibilities (user interface and management of alarm list) are rather routine and we will thus start by examining how to implement scheduling, maturation, and expiry of alarms.

Each alarm is a simple process independent of others. Because of the requirement that an alarm be independent of other user programs, alarm processes must run at user interrupt priority. We will use Delay to schedule an alarm for maturation and proper action. The question is whether this will ensure that alarms will indeed mature in the correct order. The point is that alarms may be added in arbitrary order independent of the order of their maturation, and that the process scheduler activates processes in the order in which they occur in the queue which is normally the order in which they were added. It seems that our situation is peculiar in that all alarm processes immediately delay themselves and wake up only when their Delay expires. As a consequence, alarm processes automatically schedule themselves in the correct order.

To test this hypothesis, we will write a code fragment to simulate a mismatch between the desired order of execution of alarms and the order in which they are created by the following code fragment:

```
|seconds|
"Create an array of alarm maturation times."
seconds := #(7 4 9).
" Use these maturation times to create alarm processes and schedule them in the given order."
1 to: seconds size do:
[:index | | block |
        block := [(Delay forMilliseconds: (seconds at: index)  * 1000) wait.
                Dialog warn: 'Alarm number: ' , index printString].
        block fork]
```

We expect that the alarm created second will mature first, the alarm created first will mature next, and the alarm added last will also mature last. And this is indeed what happens. The basic idea thus works - but there are two little twists that we must examine next:

1. What if the user edits the maturation time or the description of a scheduled alarm before it matures?
2. What if the user deletes a scheduled immature alarm?

Our view of alarm editing is that editing an alarm is equivalent to deleting an existing alarm and adding a new one. Since we have just tested how to add a new alarm at any time, the only remaining question is deletion of an existing alarm.

To delete an existing alarm, we must terminate its corresponding process and remove the alarm from the list. Quite frankly now, we don't yet have enough experience with processes to satisfy ourselves with this formulation - we are not confident that this strategy will really work. We will thus create a skeleton of AlarmTool implementing just this part of its behavior and test that it works before we proceed to the design of the user interface and the rest. Our implementation may have little to do with our definitive implementation because the user interface will eventually be totally different and there will be some additional requirements, but that's OK. We only jump into the implementation phase for a moment to test the concept, and return to the rest of design later. Don't feel bad about this violation of orderly development, it is quite normal. We cannot continue designing responsibly if we don't know whether an essential part of our construction is valid or not and this detour is thus not only legitimate but even necessary. It also demonstrates that realistic development is loosely structured.

Partial and preliminary implementation of AlarmTool

For our test, AlarmTool needs only one instance variable (alarms), initialized to a SortedCollection of Alarm instances sorted by maturation time. (Since this is a test class, we called it AlarmToolTest.) According to our analysis, we only need two methods to handle alarm management - one for adding, and one for removing. After the user enters time and comment, the *add* operation creates and schedules a new process, combines it with time and description into an Alarm, and adds it to alarms. For the time being, we replaced Alarm with a three-element array and implemented the idea as follows:

```
addAlarm: anArray
"Create and schedule a process corresponding to alarm (represented by anArray with time and description),
and the complete alarm information to alarm collection."
        | process |
        process := self scheduleAlarmProcessWith: anArray.
        alarms add: (anArray copyWith: process)      "Concatenate 2-element array into 3-element array."
```

where the scheduling of the process is performed as follows:

```
scheduleAlarmProcessWith: anArray
"Create and schedule a process with parameters described by anArray."
        | block |
        block := [:seconds :comment |
                (Delay forMilliseconds: seconds * 1000) wait.
                alarms removeFirst.          "Time expired, remove yourself from the list."
                Dialog warn: 'Alarm: ' , comment].
        ^(block newProcessWithArguments: anArray)
                priority: Processor userInterruptPriority; resume
```

where newProcessWithArguments: is a message defined in BlockClosure that constructs a process from a block with arguments, in our case time and comment. Since VisualWorks does not have a method to create a process over such a block *and* schedule it in a single step, we created the process without scheduling it and sent it resume to schedule it. To test our ideas we now used the fragment

```
| alarm alarms |
"Create an instance of the application."
alarm := AlarmToolTest new.
"Create suitable alarm descriptions."
alarms := #(#(5 'number 1') #(2 'number 2') #(7 'number 3')).
"Add and schedule these alarms one-by-one."
alarms do: [:anAlarm | alarm addAlarm: anAlarm]
```

This should open alarm notifiers in the order 'number 2', 'number 1', and 'number 3' - and it does. Everything is OK thus far.

Now for *alarm removal*. In the final implementation, the alarm will be obtained from the user interface and thus guaranteed to be an element of alarms. For our test, we must remove alarms by manipulating them very explicitly as in

```
| alarm alarms |
“Create an instance of the application.”
alarm := AlarmToolTest new.
“Create suitable alarm descriptions.”
alarms := #(#(6 'number 1') #(2 'number 2') #(10 'number 3')).
“Add and schedule these alarms one-by-one.”
alarms do: [:anAlarm | alarm addAlarm: anAlarm].
“Remove and terminate an alarm.”
alarm removeAlarm: #(6 'number 1').
“Add and schedule a new alarm.”
alarm addAlarm: #(7 'number 4')
```

The *remove* operation must find the appropriate element in the collection, get its process component,and terminate it:

```
removeAlarm: anArray
"Remove alarm from collection and terminate its alarm process."
	| anAlarm |
	anAlarm := alarms detect: [:el | (el at: 1) = (anArray at: 1)].
	(anAlarm at: 3) terminate.
	alarms remove: anAlarm
```

With this method, we can now run the test to add, remove, and add processes listed above. We are extremely happy to report that everything works as desired: Notifiers 2, 4, and 3 open in this order and nothing else. However, we must also warn you about another crucial experience.

During the coding and testing, we made a few small mistakes and it turns out that mistakes that affect process scheduling and termination may have very serious consequences because they can affect the operation of the whole VisualWorks environment which is itself based on processes. We suggest that in situations such as these you use the following strategy: Write your code but before you run it, file it out. Then do the test. If a disaster occurs, interrupt or terminate your session and exit without saving if necessary, restart and file in your code, correct it, file out the ‘correct’ code, and repeat. This way, you will not lose your work or damage the image.

We are now confident that our approach works and we return to design to refine the description of class Alarm.

<u>Back to design refinement</u>

Our current description of AlarmTool is as follows:

AlarmTool

Responsibilities	Collaborators
• Provide and manage user interface including the main window and the form.	
• Maintain list of alarms in order of maturation.	Alarm
• Schedule alarms for maturation according to their maturation times.	

We will now explore the three responsibilities further and summarize our findings in a final class description.

Provide and manage user interface including the main window and the form

The user interface consists of two windows:

- *Main window*. This window does not need any special widgets but the list requires two different menus depending upon whether an item is selected in the alarm list or not. The list itself is interesting because it displays only one of the three components of each Alarm object, namely its time of maturation. This will be easy to achieve because the list widget displays its constituent objects by sending them displayString and we can redefine displayString in Alarm to return only the Time component. This will ensure the desired display, while selection in the list will return the whole Alarm object.
- *Form*. This is a dialog window rather than the usual application window and its only special features are as follows:
    - When the form opens on a new alarm, it shows the current time but no description. When the form opens on an existing alarm, it shows the alarm's current parameters.
    - Time control buttons must not allow time less than Time now.
    - Time controls are restricted to 0 - 23 hours and 0 to 59 minutes. We will design them so that when the user clicks ^ at the end of the scale, the time component (minutes or hours) will start wrapping around. As an example, incrementing 59 minutes will produce 0 minutes, and decrementing 0 minutes will give 59 minutes.

Maintain list of alarms in order of maturation.

We will keep Alarm instances in alarms, a SortedCollection sorted by maturation time.

Schedule alarms for maturation according to their maturation times.

A summary based on our earlier experimental findings is as follows:
- Adding a new alarm: Open an un-initialized form. If the user closes the form by *Accept*, schedule a process with time and description obtained from the form, create an Alarm with these parameters and the process, and add it to alarms and the list widget.
- Removing an alarm: Ask user for confirmation, extract the process part of the selected Alarm and terminate it, remove selection from alarms and the list widget.
- Editing an alarm. Extract the process from the selected Alarm. Open a form with existing time and description. When the user accepts, terminate the selected alarm and remove it from the list, schedule a new process and construct new Alarm object with new parameters, and add the new Alarm to alarms.

We can now write our final class descriptions.

Alarm: I provide access to all information about a single alarm.
Components: time (Time), description (String), process (Process).

Responsibilities	Collaborators
• Creation - with time, description, and process. Method time:desc:process:	
• Accessing: time, description, and process	
• Printing: return string representation with time only via method displayString	

AlarmTool: I manage the Alarm Tool user interface, allow the user to create, edit, and delete alarms, and schedule and terminate alarms.
Components: alarms (SortedCollection), aspect variables of main window and alarm form.

Responsibilities	Collaborators
• Creation and initialization - as for an application model.	
• User interface including the main window, the form, menus, etc.	
• Main window (windowSpec)	
• Form window (form)	
• Resources (*up* and *down* buttons)	
• Pop up menus for the list widget in the main window (add, add/edit/remove)	
• Menu methods	
• Add alarm: open form initialized to current time, get data on *Accept*,	Alarm

schedule new process, create new Alarm, update list; ignore on *Cancel*. Method addAlarm.

- Remove alarm: terminate selection's process, update list. Method removeAlarm.
- Edit alarm: open form initialized to current parameters, get data on *Accept*, extract process from selection and terminate it, remove selection from alarms, schedule new process, create new Alarm, update list. Method editAlarm.

Alarm

- Maintain list of alarms in order of maturation.

Implementation

We will now show details of implementation of selected methods in the order of classes and their specification.

Class **Alarm**

This is a simple domain class and its only interesting method is displayString which is used by the list widget in the main window. It extracts the time component and converts it to String:

```
displayString
"Returned time component for display in List."
	^time printString
```

The creation method initializes all three instance variables:

```
time: aTime desc: aString process: aProcess
"Create fully initialized Alarm."
	^(self new) time: aTime; description: aString; process: aProcess
```

Class **AlarmTool**

User interface specs

User interface creation is routine, except for the *up* and *down* buttons in the form window which use images created with the Image Tool. The corresponding action methods are as follows:

```
hoursInc
"User clicked to increment hour. Keep hours within limits and wrap around if necessary."
	| h |
	hours value: ((h := hours value + 1) > 23
				ifTrue: [0]
				ifFalse: [h])
```

and

```
hoursDec
"User clicked to decrement hour. Keep hours within limits and wrap around if necessary."
	| h |
	hours value: ((h := hours value - 1) < 0
				ifTrue: [23]
				ifFalse: [h])
```

for hour buttons, and the action methods for minute buttons are similar.

Initialization

The initialization process is standard; alarmList is the Aspect of the alarm list widget:

```
initialize
"Initialize alarms and aspect variables, define change behavior and initial pop up menu."
	alarms := SortedCollection sortBlock: [:x :y | x time < y time].
	alarmList := SelectionInList new.
	alarmList selectionIndexHolder onChangeSend: #changedAlarmSelection to: self.
	alarmListMenuHolder := self menuWithAdd asValue.
	description := '' asValue
```

User interface behaviors

According to initialization, the following message is sent when the user makes a new alarm list selection:

```
changedAlarmSelection
"The list  widget has a new selection. Update pop up menu and alarm description aspect variable."
	| selection |
	selection := self alarmList selection.
	self alarmListMenu value: (selection isNil
				ifTrue: [self menuWithAdd]
				ifFalse: [self menuWithAll]).
	description value: (selection isNil
				ifTrue: [alarms isEmpty
						ifTrue: ['']
						ifFalse: ['Select alarm to see its description.']]
				ifFalse: [selection description])
```

where description is the Aspect of the description widget. Method alarmListMenu was specified as the *Menu* property of the list widget when we painted the interface, and the menu building methods are as follows:

```
menuWithAdd
"No alarm is selected, only add is allowed."
	| mb |
	mb := MenuBuilder new.
	mb add: 'add' -> #addAlarm.
	^mb menu
```

and

```
menuWithAll
"An alarm is selected, all three commands are available."
	| mb |
	mb := MenuBuilder new.
	mb add: 'add' -> #addAlarm; add: 'edit' -> #editAlarm; add: 'remove' -> #removeAlarm.
	^mb menu
```

The critical operations of adding, editing, and deleting menus are all triggered by the messages specified in these methods and are implemented as follows:

```
removeAlarm
"To remove an alarm, terminate its process and remove it from the alarms collection and the list widget."
	| selection |
	selection := alarmList selection.
	selection process terminate.
	alarmList list: (alarms remove: selection; yourself)
```

Note that we have not included a confirmation dialog; this is left as an exercise.

Adding and editing have much in common (both open the dialog window, respond to *Accept* and *Cancel*, schedule a new process, and update the list) and factoring the shared behavior out simplifies their definition. We define the *add* operation as follows:

```
addAlarm
"Prepare parameters for form (current time, no description) and send doAddition to do the rest."
	self initializeFormForNewAlarm.
	self doAdd
```

where the critical work is done in

```
doAdd
"Shared part of add and edit. Open form as dialog, ignore on Cancel, obtain time from hours and minutes,
schedule new process, update list."
	| process |
	(self openDialogInterface: #form)
		ifFalse: [^self].
	time := Time fromSeconds: hours value * 3600 + (minutes value * 60).
	process := self scheduleAlarmProcessWith: (Array with: time with: descriptionInForm value).
	alarms add: (Alarm	time: time
				desc: descriptionInForm value
				process: process).
	alarmList list: alarms
```

where hours and minutes are Aspect variables of the corresponding input fields. This method is also used by the editing method:

```
editAlarm
"To execute edit, prepare parameters for form and send doAddition to do the rest."
	self initializeFormForExistingAlarm.
	self doAddition
```

Initialization of the form for a new alarm (see addAlarm) is done by

```
initializeFormForNewAlarm
“Prepare form aspect parameters for new alarm.”
	self hours value: Time now hours.
	self minutes value: Time now minutes.
	self descriptionInForm value: ''
```

and editing an existing alarm uses

```
initializeFormForExistingAlarm
“Prepare form aspect parameters for alarm selected in list.”
	| selection |
	selection := alarmList selection.
	alarms remove: selection.
	selection process terminate.
	self hours value: selection time hours.
	self minutes value: selection time minutes.
	self description value: selection description
```

By the way, when you open your alarm tool, you may get strange values for starting hours and minutes. To correct this, change you VisualWorks time zone. We leave this as an exercise.

The only remaining action methods are for the *Close* button in the main window, and for *Accept* and *Cancel* in the form. The *Close* button closes the application in the usual way. *Accept* and *Cancel* use the standard behavior of dialog windows and use methods accept and cancel defined in SimpleDialog; these methods close the window and return true (on *Accept*) or false (on *Cancel*). We relied on this behavior in the first statement in method doAdd above.

This leaves scheduleAlarmProcessWith: which uses the data obtained from the form to create a new process with given maturation time and description, and schedules it. Its definition is

```
scheduleAlarmProcessWith: anArray
"Create and schedule a process with data obtained from the form and returned in anArray."
        | block |
        block := [:maturationTime :comment |
                (Delay forMilliseconds: maturationTime asSeconds - Time now asSeconds * 1000) wait.
                Dialog warn: ('Alarm at: ' , Time now printString , '\' , comment) withCRs].
        ^(block newProcessWithArguments: anArray)
                priority: Processor userInterruptPriority; resume
```

When the process resumes after the prescribed delay, it open the notification window showing the alarm's time and description, and terminates, removing itself from the process queue.

Main lessons learned:

- Since processes are created from blocks and blocks may have arguments, processes may also be created with arguments using newProcessWithArguments:.
- Method newProcessWithArguments: creates but does not schedule a process. Scheduling is by message resume.

Exercises

1. Processes are expensive in terms of CPU requirements and having many of them will slow down VisualWorks. A better approach is to have a single process keeping all alarms in a list and handling them as needed. Reimplement Alarm Tool using this alternative approach.
2. Extend the alarm's notion of time to include date.
3. Add the following behaviors if they are not yet functional:
   a. Set alarms remain functional even when Alarm Tool is closed.
   b. Set alarms remain functional from session to session.
   c. When the Alarm Tool is closed and then reopened later in the same session, non-expired alarms remain set and can be individually edited and modified.
   d. Extend Exercise c across sessions.
4. Make it possible to specify alarms in relative terms such as 'in ten minutes'.
5. Add recurrent alarms of various kinds such as 'every ten minutes', 'every ten minutes until canceled by user', and 'every ten minutes, five times in a row.'
6. Add alarms that occur on a specified schedule.
7. According to the current specification, there is no restriction on alarm interference with other higher priority processes such as the stopwatch. Comment on the possibility of running alarms and the stopwatch at the same time, possible problems (if any), and possible remedies.
8. Add a digital clock to Alarm Tool.

13.3 Coordinating mutually dependent processes - Train Simulation

In many important situations, two or more processes are mutually dependent because they share some resource. In situations like these, conflicts may arise and processes require coordination. A typical example of such a situation is printing on a shared network printer. A print job cannot be started while another job is printing and the processor managing the print queue must therefore wait for the printer to signal that it is ready before it sends another job.

A physical world example of the need to coordinate processes is the previously mentioned crossing of railroad track and a highway road: If a train is crossing, vehicles coming to the crossing must stop and this is normally implemented by a semaphore - a traffic light activated by the arriving train, and deactivated when the train leaves the crossing area. We will use this example to illustrate process coordination in Smalltalk. First, however, some background.

Semaphores

A Smalltalk semaphore is an analogy of a train crossing semaphore implemented as an instance of class Semaphore. The essential messages understood by semaphores include new (to create a new instance), wait (to indicate the desire to use the resource guarded by the semaphore and to turn the 'red light' on), and signal (to indicate that the resource is no longer needed and to turn the 'green light' on).

When two or more processes interest in the same resource, they use a shared Semaphore. Note that a Semaphore has no explicit connection to a process and behaves very much like a portable signaling device that can be carried around and placed to guard a resource anywhere the user wants.

When a process needs to use the resource guarded by a Semaphore, it sends wait to it. The wait message performs two functions: It first checks whether the 'light is green', in other words, whether the resource is available. If it is, it 'turns the light to red' to indicate that the resource is now in use and unavailable to other processes, and the process that sent it can continue executing its block. If the light is red, the Semaphore puts the process in a queue waiting for this Semaphore to turn green again.

When a process wants to indicate that it is finished using the protected resource, it sends the Semaphore the signal message. This message turns the light to green, and if there are any processes waiting at this Semaphore, the one at the head of the queue (the one that was the first to send wait) can resume. As we can anticipate from our knowledge of processes, the process will actually resume only when the currently active process gives up control by terminating, by suspending itself, or by being sent to the end of the queue by Processor yield. Note that the consequence of the described mechanism is that a process that issues wait can only proceed if the semaphore has received a signal message before.

The principles that we have just explained are summarized in Figure 13.7.

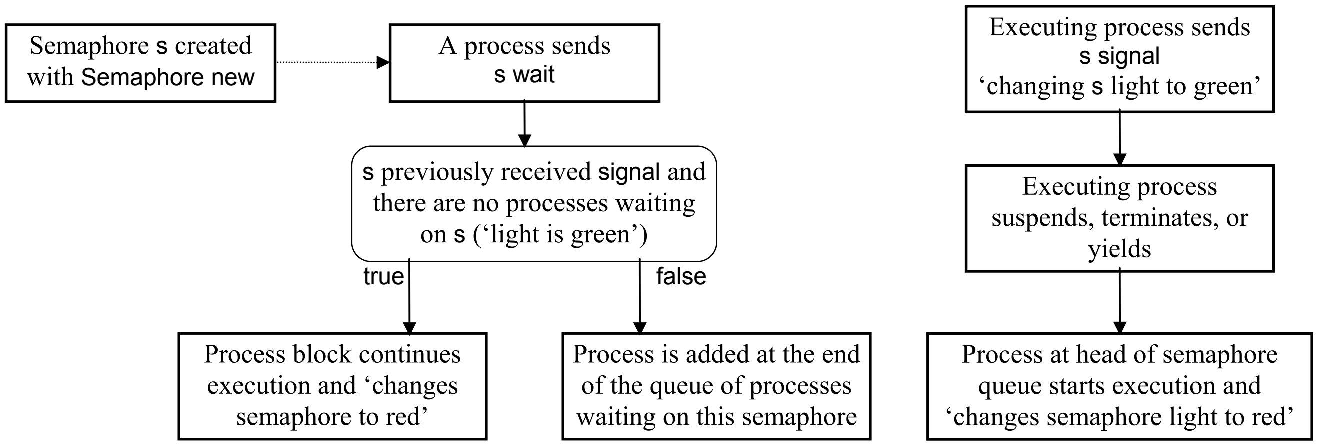

Semaphore s created with Semaphore new
A process sends s wait
s previously received signal and there are no processes waiting on s ('light is green')
true
false
Process block continues execution and 'changes semaphore to red'
Process is added at the end of the queue of processes waiting on this semaphore
Executing process sends s signal 'changing s light to green'
Executing process suspends, terminates, or yields
Process at head of semaphore queue starts execution and 'changes semaphore light to red'

Figure 13.7. Operation of Semaphore. Any number of semaphores may exist at a time.

Train simulation - specification

After this introduction, we are now ready to give an example of the use of semaphores.

Problem: Develop a simulation of crossing truck and train tracks with the user interface in Figure13.8. At the beginning of the simulation, the truck is at the top of its track and the train at the left end of its track. When the user clicks *Go*, the train and the truck start moving at a constant speed, changing direction when they reach the end of the track. If a vehicle reaches the beginning of the crossing area (its own traffic light) and its semaphore light is green, it turns the other vehicle's semaphore lights to red. It then proceeds, moving in the usual way. When it reaches the end of its crossing area, it turns the lights on the other vehicle's track back to green. If the semaphore light on a vehicle's track is red when it arrives at the crossing, it stops and waits until the light turns green.
Simulation continues until the user clicks *Stop* and may be restarted by clicking *Go* again.

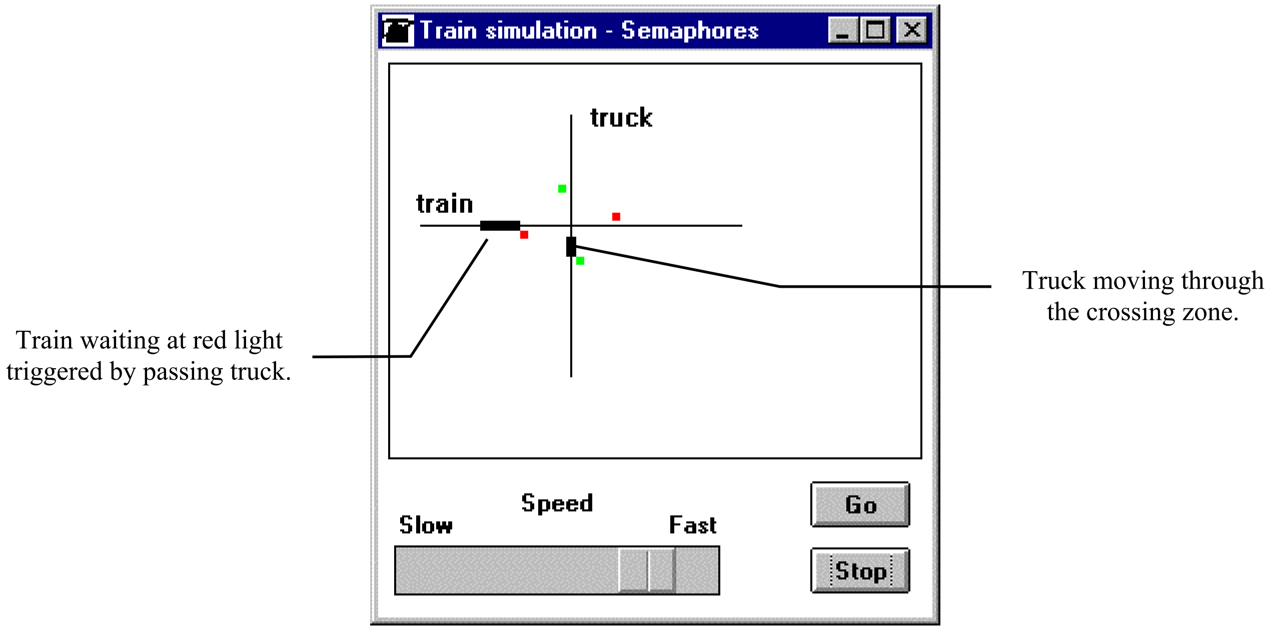

Train simulation - Semaphores
truck
train
Train waiting at red light triggered by passing truck.
Truck moving through the crossing zone.
Speed
Slow
Fast
Go
Stop

Figure 13.8. User interface for train simulation.

Scenarios

Scenario 1: Starting or restarting simulation
1. *User* clicks *Go*.
2. *Program* starts moving both vehicles from their current position. Motion is 'simultaneous'.

Scenario 2: Stopping simulation
1. *User* clicks *Stop*.
2. *Program* stops both vehicles in their current position.

Scenario 3: Changing speed
1. *User* moves the speed control slider to a new position.
2. *Program* changes speed setting for both vehicles and if the simulation is running, the new setting immediately takes effect.

Scenario 4: Train reaches the start of the crossing and its semaphore light is green
1. *Program* changes light on the truck side to red.
2. *Program* advances train to next position.

Scenario 5: Train reaches the start of the crossing and its semaphore light is red
1. *Program* stops the train at the light.
2. *Program* keeps advancing the truck until it reaches the end of the crossing zone.
3. *Program* turns train lights to green.
4. *Program* moves train and turns truck lights to red.

Scenario 6: Train reaches end of crossing
1. *Program* changes truck lights to green.

Scenario 7: Train reaches end of track
1. *Program* reverses direction of train motion.

Truck scenarios are analogous to train scenarios.

Preliminary Design

The domain objects include the train and the truck and we will leave it to the application model to take care of the simulation. It could be argued that the simulation object could be a part of the domain model and separate from the application model but we don't expect it to be complex. Finally, a layout view will implement the track layout subview but there is no active controller because the simulation subview does not allow any user interaction. Altogether, we need the following classes:

Train. Knows its position and direction of motion, knows how to calculate its next position, check that it reached the end of the track or the beginning or the end of its crossing, and how to turn around (change direction). Knows its simulation parameters such as its length (the length of the rectangle).

Truck. Same responsibilities as Train but truck-specific.

TrainSimulation. Application model. Contains the specification of the window and knows about the view class implementing the simulation subview. Knows global simulation parameters including start and length of train and truck tracks, and the limits of the crossing area (distance from crossing to lights). The *Go* action button is the heart of the simulation, *Stop* suspends simulation.

LayoutView. Display tracks, lights, and vehicles.

We will now examine the role of our classes in the previously listed scenarios.

Scenarios

Scenario 1: Starting or restarting simulation
1. *User* clicks *Go*.
2. *Program* starts moving both vehicles from their current position. Motion is 'simultaneous'.

This scenario is one of the essential components of the problem and before we start expanding it, we must have a conceptual model of the simulation. The problem invites the use of processes, one for each vehicle; after all, each vehicle is autonomous. The requirement that vehicle motion be 'simultaneous' means that the two processes must systematically take turns.

An important part of the motion of vehicles in our problem is the crossing and we must thus consider it in this context. The behavior of the crossing suggests the use of a semaphore shared by the two vehicles. When a vehicle reaches the crossing zone, it tells the semaphore about its desire to move and if the semaphore is ready (has been signaled), it allows the vehicle to proceed and change lights.

Another question is the relation of the train and truck *processes* to the Train and Truck *classes*. Our train process will use the Train domain object to perform all the calculations and trigger the display, and the truck process will use the Truck class in a similar way. Process objects thus use domain objects but are separate from them. With this background, we can now expand Scenario 1.

Scenario 1: Starting or restarting simulation
1. *User* clicks *Go*.
2. TrainSimulation creates a train process and a truck process.
3. TrainSimulation starts the train process and the vehicle processes automatically start taking turns executing single step motion while respecting the semaphore at the crossing.

Scenario 2: Stopping simulation
1. *User* clicks *Stop*.
2. TrainSimulation terminates the two processes.

The immediate question is whether clicking *Stop* can actually freeze simulation. The point is that while the application is executing the simulation loop, other methods can only be accessed from the loop. Under these conditions, will the application be able to respond to the *Stop* button? The answer is that the *Stop* button will still be effective because our processes run at UserSchedulingPriority (the priority of their parent process - our simulation), whereas keyboard and mouse events are managed by a process running at LowIOPriority whose priority is higher. Clicking *Stop* will thus take precedence over simulation and activate the action method of the *Stop* button even though the train and the truck processes are running[3].

Scenario 3: Changing speed
1. *User* moves the slider to a new position.
2. *Program* changes speed setting and if the simulation is running, the new setting immediately takes effect.

To control speed, we will end each simulation step with a Delay whose duration will be derived from the setting of the slider. For slow speed, the delay will be long, for fast speed, the delay will be short. Since slider control runs under LowIOPriority, the effect of dragging the slider marker to a new position immediately changes its value aspect. To control speed during simulation we must thus obtain the current slide position in each simulation step and use it to control the delay. (We will explain the simple rules for using a slider in the Implementation section.) We can now expand the scenario as follows:

Scenario 3: Changing speed
1. *User* moves the slider to a new position.
2. The active vehicle Process reads the new slider setting and uses it to create and schedule a Delay of appropriate duration.

Scenario 4: Train reaches the start of the crossing area and its semaphore lights are green
1. The Train object checks its current position and finds that it has just reached the start of the crossing zone. This test involves checking the current direction of motion, the position of the crossing, its width, and the length of the vehicle.
2. The Train object sends a wait message to the semaphore.
3. Since the semaphore is presumably *not* blocked, the train process is not suspended, wait turns the semaphore lights on the truck track to red, and the process advances the train to the next position.

Scenario 5: Train reaches the start of the crossing area and its semaphore lights are red
1. The Train object checks the current train position and finds that it has just reached the start of the crossing zone.
2. The Train object sends wait to the semaphore.
3. Since the semaphore *is* blocked, it suspends the train Process.

[3] We now realize that we should have asked the question of how the user interface can take over from the application a long time ago - this principle is the essence of all interactive applications.

4. The truck which is moving through the crossing eventually leaves the crossing area, sends signal to the semaphore (Scenario 6), and suspends itself temporarily (see the alternation of train and truck processes in Scenario 1). The blocked train Process resumes, turns the semaphore lights on the truck track to red, and advances the train to the next position.

Scenario 6: Train reaches the end of the crossing area

1. The Train object recognizes the 'end-of-crossing' position, changes truck lights to green, and sends signal to the semaphore, unblocking a truck possibly waiting for green light.

Scenario 7: Train reaches end of track

1. The Train object recognizes the 'end-of-track' position. It changes its direction state which controls whether the next position is calculated by incrementing or decrementing the current position, and makes a move.

Preliminary object diagram

Except for the class hierarchy and possible abstract classes, we now have a good understanding of the mutual relationships of our classes. The corresponding diagram is in Figure 13.9.

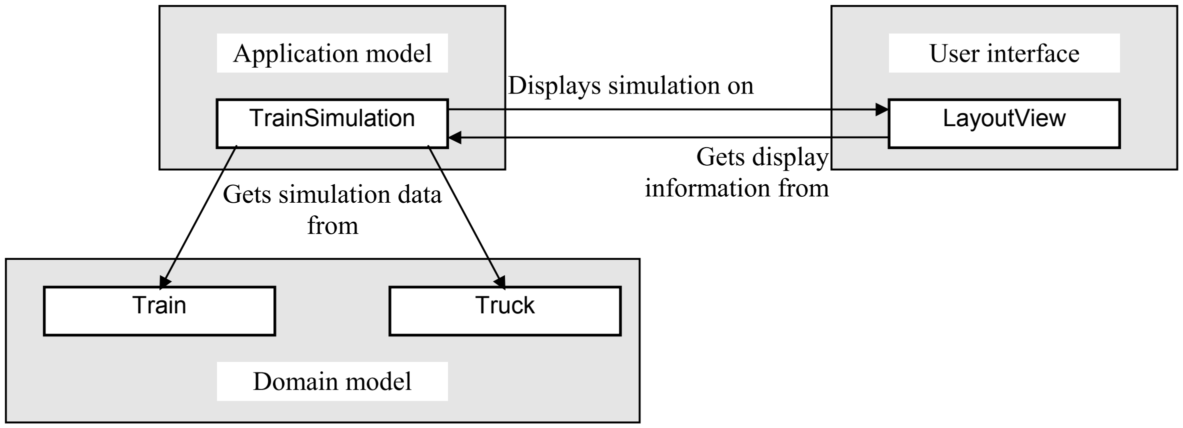


Figure 13.9. Preliminary Object Model of train simulation.

Design refinement

Class hierarchy

Domain objects Train and Truck have a lot in common: Both use the same principle to move, and both must know how to check crossing boundaries and end of track, and how to turn. We will thus create an abstract class called Vehicle implementing as much of the shared functionality as possible, and delegate vehicle-specific details to Train and Truck. TrainSimulation is our application model and a subclass of ApplicationModel, LayoutView is a subclass of View. The resulting class hierarchy is as in Figure 13.10.

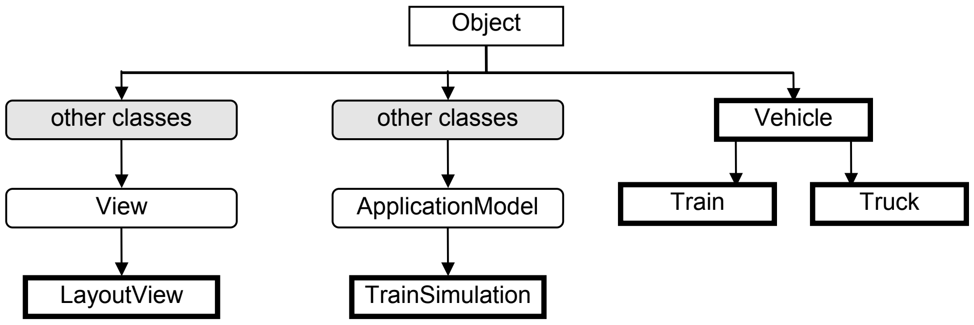


Figure 13.10. Class Hierarchy Diagram with classes to be designed shown as heavy rectangles.

Refined class descriptions

We now understand enough of the behaviors of our classes to be able to start refining class descriptions:

Domain classes

Vehicle. Abstract superclass of domain classes Train and Truck. Factors out shared behavior and knowledge such as vehicle position and direction of motion, calculation of the next position, testing for the end of the track or crossing, and turning.
Superclass: Object
Components: limit (distance from crossing to lights), position (distance of vehicle from start of track), direction (#incrementing or #decrementing position), vehicleLength (pixels), model (reference to the TrainSimulation object running the simulation).

Behaviors	Collaborators
• initialization - position and direction; method initialize	
• moving - trigger display in new position, activate Delay	TrainSimulation, Train, Truck
• move - check position with respect to crossing, send wait or signal to semaphore and redisplay lights if necessary, check for end of track and turn if necessary, move one step	
• moveOneStep - calculate new position using direction information, display (ask vehicle for its display object), delay	Train, Truck
• semaphores: aColorValue - ask model to change color of semaphore lights	TrainSimulation
• testing - check whether entering or leaving crossing zone (enteringSemaphore, leavingSemaphore), check for end of track (endOfTrack)	

Vehicle-specific parts of these behaviors are left as subclass responsibilities for Train and Truck.

One note is in order about our design: The name of the variable model suggest dependency and a possible better implementation. We leave this question as an exercise.

Train. Concrete train vehicle. Implements train specific responsibilities and specializes those inherited from Vehicle.
Superclass: Vehicle
Components: Inherited components

Behaviors	Collaborators
• initialization - execute inherited Vehicle initialization and initialize train specific track and vehicle parameters - method initialize	

Truck. Same as Train but truck-specific.

Application model

Before we can write the description of TrainSimulation, we must refine our understanding of its critical task - the creation and nature of the processes that represent the train and the truck. We have already decided that each of these processes will be implemented as an infinite loop that asks the corresponding domain object (a Train or a Truck) to make a move (including redisplay), and then somehow yields to the other process. Our first hunch is to do something like

```
trainProcess := [ [train move.
                   trainProcess suspend.
                   truckProcess resume] repeat] fork.
truckProcess := [ [truck move.
                   truckProcess trainProcess suspend.
                   trainProcess resume] repeat] fork
```

In this implementation, a vehicle process performs a one-step move, resumes the other process (currently suspended), and suspends itself, activating the resumed process. Unfortunately, this approach will not work: Assume that the truck is in the middle of the crossing zone when the train reaches the semaphore light. Its move operation issues

```
semaphore wait
```

and is placed in the semaphore waiting queue because the light is red. The truck makes one step, suspends itself (see above), and waits for the train to resume it. However, the train is waiting for the truck to turn the semaphore to green – which it can't do because it cannot move. The train and truck processes become *deadlocked*. To solve this problem, we decide to rely entirely on the semaphore and code the processes as follows:

```
trainProcess := [[train move.
                  Processor yield] repeat] fork.
truckProcess := [[truck move.
                  Processor yield] repeat] fork
```

To understand how this approach works, assume that none of the vehicles reached the crossing zone yet and that it is the train's turn to move. The following then happens:

1. The train makes a move and executes

```
Processor yield
```

asking Processor to put it at the end of the queue of runable processes (processes that are not suspended or waiting for a semaphore). This leaves the truck at the head of the queue.
2. Since the truck is not waiting on the semaphore, it executes a move and asks Processor to put it at the end of the queue. This leaves the train at the head of the queue, the operation proceeds as described in Step 1, and so on.

Eventually, one of the vehicles (for example the train) reaches the beginning of the crossing zone. As a part of executing its move (see description of Train and Vehicle above), it sends

```
semaphore wait
```

turns the truck's light to red, and yields via Processor. If the truck is not yet in the crossing zone, it makes its move, yields to the train, and so on. If the train reaches the end of the crossing zone before the truck reaches its start, it turns truck's light to green and everything proceeds as in steps 1 and 2. If, however, the truck reaches the crossing zone while its semaphore is still red, its

```
semaphore wait
```

blocks the truck process. The truck process is now removed from the queue of runable processes and the train is now the only process in that queue. The train makes a move, and yields. But since there is no other process in the queue of runable processes, this does not stop it, it executes another move, and yields. If the did not reach the end of the crossing zone, the yield again does not change anything and the train makes another move (while the truck is still waiting on the semaphore). Eventually, the train reaches the end of the crossing zone, sends

```
semaphore signal
```

and this removes the truck from the semaphore queue and puts it in the queue of runable processes. When the train now executes

```
Processor yield
```

and goes to the end of the queue, it now finds itself behind the truck which makes its move, yields, the train makes its move and yields, and so on.
With this background, we can now write a description of class TrainSimulation as follows:

TrainSimulation. Responsible for user interface and simulation.
Superclass: ApplicationModel
Components: delay (Integer - current value of delay between vehicle moves – controlled by slider), layout (View), semaphore (Semaphore), train (Train), truck (Truck), trainTrackLength (Integer), truckTrackLength (Integer), trainTrackStart (Point), truckTrackStart (Point), trainProcess (Process), truckProcess (Integer), speed (Integer), aspect variables.

Behaviors	Collaborators
• initialization - initialize instance variables, send signal to semaphore to get it ready to pass the first incoming vehicle; method initialize	Train, Truck, Vehicle, LayoutView
• displaying - pass requests to display vehicle or change lights to LayoutView; methods display: and semaphores:for: (color and type of vehicle)	LayoutView
• actions - response to action buttons	
• go - create the two mutually yielding processes and start one of them	Train, Truck
• stop - terminate both processes	

At this point, we realize that we should not allow the user to close the window and terminate the application if the processes are running. We thus add a changeRequest method that intercepts each attempt to close the window, checks whether there are any running processes (Boolean variable oKToClose), and if there are, warns the user and refuses to close the window.

User Interface

LayoutView. Display tracks, lights, and vehicles.
Superclass: View
Components: Parameters of layout geometry obtained from TrainSimulation – cached for greater efficiency.

Behaviors	Collaborators
• displaying - view, train and truck in new position, lights	LayoutView
• displayOn: - the standard required display method	
• display Train or Truck upon direct request from domain objects via TrainSimulation; method display: with type of vehicle	
• display of semaphore lights; methods displayTrainSemaphores: and	

displayTruckSemaphores: with a ColorValue argument

- controller specification - specify NoController in method defaultControlleClassr

Object Model Diagram

The only difference from the preliminary diagram is that we now know that Train and Truck have superclass Vehicle (Figure 13.11).

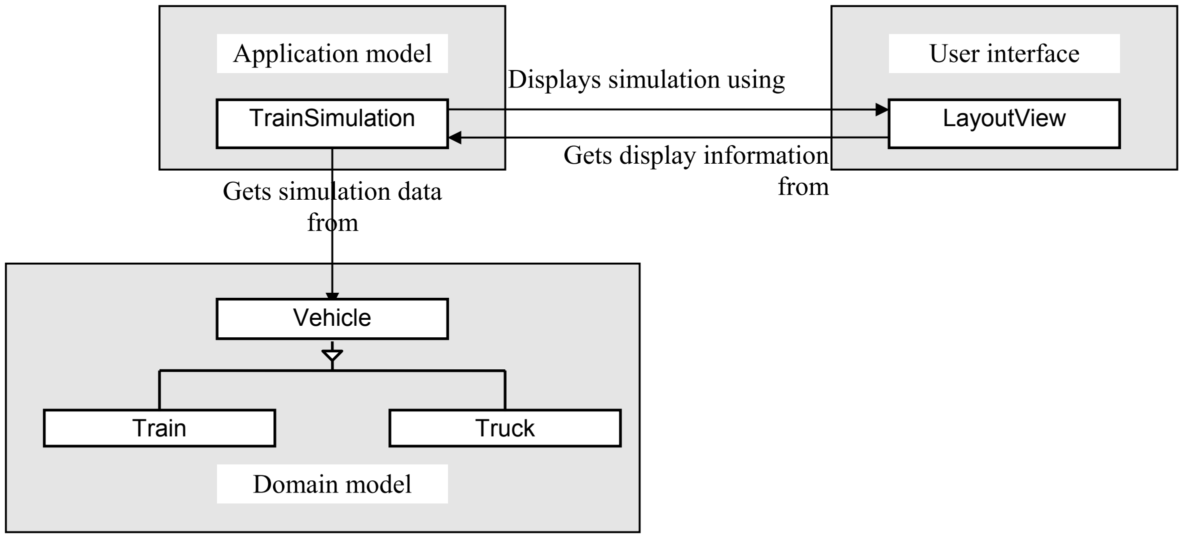


Figure 13.11. Final Object Model of train simulation.

Implementation

We will now show implementation details of several selected methods class by class.

Class Vehicle

Initialization is shared between the abstract vehicle and the concrete subclass. The shared part is as follows:

```
initialize
"Initialize parameters shared by Train and Truck. Leave the rest to Train and Truck classes."
	position := 0.
	direction := #incrementing
```

Most of the move operation is implemented in this abstract class and we split it into methods that perform tests for crossing limits and end of track, and a method that performs a one-step move. The principle is

```
move
"Test for crossing limit and end of track, make one-step move if appropriate."
        “Test whether we are at a crossing limit. If so, message semaphore and make one-step move.”
        self comingToSemaphore
                ifTrue:   [model semaphore wait.
                          self semaphores: ColorValue red].
        self leavingSemaphore
                ifTrue:   [model semaphore signal.
                          self semaphores: ColorValue green].
        “Check for end of track and ‘turn’ if required.”
        self endOfTrack ifTrue: [self turn]
        “Make a one-step move.”
        self moveOneStep.
```

but to speed up execution, we perform one-step moves on each successful test and exit to avoid further tests:

```
move
"Test for crossing limit and end of track, make one-step move if appropriate."
        “Test whether we are at a crossing limit. If so, inform semaphore and a make one-step move.”
        self comingToSemaphore
                ifTrue:   [model semaphore wait.
                          self semaphores: ColorValue red.
                          ^self moveOneStep].
        self leavingSemaphore
                ifTrue:   [model semaphore signal.
                          self semaphores: ColorValue green.
                          ^self moveOneStep].
        “Check for end of track and ‘turn’ if required.”
        self endOfTrack ifTrue: [self turn]
        “No special situation - make a one-step move.”
        self moveOneStep.
```

The move itself is performed by moveOneStep as follows:

```
moveOneStep
"Move one step in the current direction and pause briefly to provide the required control of speed."
        direction == #decrementing
                ifTrue: [position := position - 1]
                ifFalse: [position := position + 1].
        model display: self symbol.
        (Delay forMilliseconds: model delay) wait
```

We delegate display to TrainSimulation by asking it to display a vehicle identified by its symbol. The symbol method is defined in Train as

```
symbol
        ^#train
```

and similarly in Truck. The turn that occurs at the end of the track is implemented as follows:

```
turn
"Reached end of track, turn."
        direction == #incrementing
                ifTrue: [direction := #decrementing]
                ifFalse: [direction := #incrementing]
```

and the test methods are

```
comingToSemaphore
```

```
"Return true if I have just reached the start of the crossing zone."
	^(direction == #incrementing and: [position = (crossing - limit - vehicleLength)])
		or: [direction == #decrementing and: [position = (crossing + limit)]]
```

and

```
endOfTrack
"Return true if this is the end of the track. Remember that vehicle might have turned."
	^(direction = #decrementing and: [position = 0])
		or: [direction == #incrementing and: [position = (trackLength - vehicleLength)]]
```

The method that triggers redisplay of semaphore lights uses the same principle as moveOneStep and asks TrainSimulation to display semaphores controlled by the vehicle in the specified color:

```
semaphores: aColorValue
“Ask TrainSimulation to display sempahores controlled by vehicle identified by its symbol.”
	model semaphores: aColorValue for: self symbol
```

As you can see, Vehicle implements most of the functionality of its concrete subclasses.

Class Train

Initialization performs inherited initialization and sets initial values of train-specific parameters:

```
initialize
“Initialize vehicle parameters via super and Train specific values.”
	super initialize.
	limit := model trainCrossingLimit.
	vehicleLength := 20.
	crossing := model truckTrackStart x -  model trainTrackStart x.
	trackLength := model trainTrackLength
```

The rest are accessing methods including symbol (listed above) and

```
crossing
“Calculate distance from start of track to intersection.”
	^model truckTrackStart x - model trainTrackStart x
```

Class Truck

All methods in class Truck are truck-specific variations on Train methods.

Class TrainSimulation

Initialization of TrainSimulation consists of setting the parameters of the layout, creating a Train and a Truck, creating a Semaphore and signaling it to get ready for the first train, creating and assigning the layout view, assigning initial simulation speed, and setting the variable that reflects the state of the train and truck processes and determines whether the application can be closed or not.

```
initialize
	trainTrackLength := 160.
	truckTrackLength := 130.
	trainTrackStart := 15 @ 80.
	truckTrackStart := 90 @ 25.
	trainCrossingLimit := 25.
	truckCrossingLimit := 20.
	truck := Truck newOnModel: self.
	train := Train newOnModel: self.
```

```
        semaphore := Semaphore new.
        semaphore signal.
        layout := LayoutView new.
        layout model: self.
        speed := 25 asValue.        “Aspect variable associated with the speed slider.”
        oKToClose := true
```

The essential go action method is designed along the lines decided during Design Refinement:

```
go
“User clicked Go. Clear oKToClose to indicate that we have active processes and cannot close the
application, create and schedule the train and truck processes.”
        OKToClose := false.
        trainProcess := [  [train move.
                            Processor yield] repeat] fork.
        truckProcess := [  [truck move.
                            Processor yield] repeat] fork
```

The *Stop* button terminates the two processes and sets oKToClose to indicate that the application can now be closed:

```
stop
“Terminate train and truck processes, sts OKToClose to True to indicate that the application may close.”
        truckProcess terminate.
        trainProcess terminate.
        OKToClose := true
```

Attempts to close the window send changeRequest. The method returns true when it is OK to close the application (when the vehicle processes are not running), and false otherwise:

```
changeRequest
“Decide whether it is OK to close the application.”
        oKToClose
                ifFalse:  [Dialog warn: Click Stop to stop running processes before closing.'].
        ^oKToClose
```

Class TrainSimulation also defines a group of methods responsible for display. In essence, they pass requests for display coming from vehicles to the layout view. They include

```
display: aSymbol
“Pass request to display a Train or a Truck to layout view.”
        layout display: aSymbol
```

and

```
semaphores: aColorValue for: aSymbol
“Pass request to display lights to the layout view.”
        layout displaySemaphoresFor: aSymbol inColor: aColorValue
```

Finally, the implementation of the slider. The slider is a simple widget whose value holder holds a number corresponding to the position of the marker of the slider bar. When the position of the marker changes, the value changes and when the value is changed programmatically, the marker moves to the corresponding position. All we need to do to implement a slider is to specify the start and the end values corresponding to the left and right endpoints using the Properties Tool, define the aspect variable (we called it speed), and define its initial value which we did in the initialization method above. When assigning the endpoint values, don’t forget that the leftmost slider position represents the *slowest* speed which corresponds to the *largest* value of delay. The value of speed is used to calculate the delay

```
delay
        ^speed value
```

and both vehicles use this method to determine the length of their delay.

Class LayoutView

In addition to specifying NoController as the default controller class, the major responsibility of LayoutView is displaying. This responsibility can be divided into two tasks: Response to damage notification, and request for redisplay initiated from the domain model by vehicle motion. Response to damage notification is, as always, implemented by a displayOn: aGraphicsContext method. It draws the layout, the train and the truck, and the lights, and its definition as follows:

```
displayOn: aGraphicsContext
"Display tracks, vehicles, semaphores, and labels."
        aGraphicsContext
                displayLineFrom: self trainStart to: self trainEnd;
                displayLineFrom: self truckStart to: self truckEnd;
                displayString: 'train' at: self trainStart x + 4 @ self trainStart y - 6;
                displayString: 'truck' at: self truckStart x + 4 @ self truckStart y + 6.
        self displayTrainSemaphoresOn: aGraphicsContext withColor: model trainSemaphoreColor;
                displayTruckSemaphoresOn: aGraphicsContext withColor: model truckSemaphoreColor;
                displayTruck;
                displayTrain
```

The display a vehicle is implemented as follows:

```
displayTrain
        self graphicsContext displayRectangle: (0 @ 0 corner: self trainLength @ 5)
                at: self trainStart + (self trainPosition @ -2)
```

and the methods displaying the semaphores are explained below.

Vehicle requests for redrawing arrive from TrainSimulation but are triggered by the vehicles. They require redrawing of vehicles and semaphore lights. To redraw the vehicles, we can use one of two approaches. One is based on the view that we want to move a geometric object (the train rectangle) from one position to another. This high level approach deals with redrawing as a simple form of animation and to implement it, we can use methods follow:while:on: or moveTo:on:restoring: defined in class VisualComponent. We leave this approach as an exercise and use the low level perspective explained next.

Instead of thinking about moving a visual object, we can take the view that moving a rectangle horizontally or vertically by one pixel which requires only erasing the trailing edge and drawing a line in front of the leading edge. If the current position of the trailing edge corresponds to the crossing, we don't erase it because it also represents the track. To 'erase' the back of the vehicle, we redraw the corresponding line in white. After erasing the end of the rectangle, we must also redraw the piece of track that we erased in the process. This brings up the important fact that to display a point in VisualWorks, you must draw a one-pixel long line. The whole definition, as implemented for the train, is as follows:

```
displayTrainOn: aGraphicsContext
"Calculate the position of the trailing and leading edges and redraw them to simulate motion."
        | length direction position oldEnd newStart |
        length := model trainLength.
        direction := model trainDirection.
        position := model trainPosition.
        direction == #incrementing
                ifTrue:   [oldEnd := position - 1.
                          newStart := oldEnd + length]
                ifFalse:  [oldEnd := position + length + 1.
                          newStart := position].
        "Erase end of train except when on crossing."
        oldEnd = (crossing x - trainStart x) ifFalse:
                          [aGraphicsContext paint: ColorValue white;
                                  displayLineFrom: self trainStart + (oldEnd @ -2)
                                          to: self trainStart + (oldEnd @ 2)].
        aGraphicsContext paint: ColorValue black;
                          "Draw leading edge of train."
                          displayLineFrom: self trainStart + (newStart @ -2)
                                  to: self trainStart + (newStart @ 2);
                          "Redraw erased part of track."
                          displayLineFrom: self trainStart + (oldEnd @ 0)
                                  to: self trainStart + (position @ 0)
```

We had to correct the arithmetic expressions in this method several times to get the display to work properly, mainly because we did not bracket the binary messages + and @ and they executed differently from our expectations. This is a frequent and annoying mistake.

To display semaphore lights, we wrote separate methods for the train and for the truck. They calculate some essential points and display a colored rectangle along the track on each side of the crossing:

```
displayTrainSemaphoresWithColor: aColorValue
"Display train semaphore lights as rectangles of specified color."
        | limit rectangle |
        limit := self trainCrossingLimit.
        rectangle := 0 @ 0 corner: 4 @ 4.
        self graphicsContext
                paint: aColorValue;
                displayRectangle: rectangle at: self crossing + (limit negated @ 3);
                displayRectangle: rectangle at: self crossing + (limit - 4 @ -6)
```

The application now works and it only remains to improve the implementation of a few awkward methods. As an example, the last method could be improved by defining a lightRectangle variable and calculating the coordinates from the rectangle instead of using fixed numeric values. We could then change the size of the lights without having to change the calculation of the position of the rectangle. We leave this as an exercise. A non-trivial improvement of the user interface is the subject of the next section.

Main lessons learned:

- Class Semaphore makes it possible to synchronize processes.
- The essence of the operation of a Semaphore are messages signal and wait.
- If a Process sends wait to a Semaphore and the difference between the number of signal and wait messages received by the Semaphore is not at least 1, the Process is suspended until the condition is satisfied.
- A deadlock occurs when several processes wait for one another and cannot proceed.
- Processor yield moves the currently active process to the end of its queue of runable processes.

Exercises

1. Complete and clean up the implementation as suggested. One of the questions to resolve is whether the model instance variable of Vehicle implies that TrainSimulation should be a Model of Vehicle and if so, how this part of the application should be reimplemented.
2. Replace train and truck rectangles with arrows and reverse their direction at the end of the track.
3. Modify Train and Truck so that they know how to display themselves.
4. Change the simulation program so that the train and the truck each have their own speed controlled by its own slider.
5. Implement vehicle redrawing using VisualComponent's animation protocol. In our case, the code is not simpler but its implementation is an interesting introduction to animation.

6. In our design, the train and the truck can never collide or become deadlocked. Real life crossing are not, unfortunately, so perfect because a driver can overlook a traffic light and a traffic light may misfunction. Modify our design so that the user can set a <0,1> probability that a failure occurs.
7. Add an animation of a crash of the two vehicles.
8. Extend the simulation program to several parallel train and truck tracks with a single shared crossing. Each track has its own vehicle running at its own speed and each vehicle's speed is controlled independently. Speed is controlled by a single slider and a set of radio buttons, one for each vehicle.
9. Simulation of events occurring in parallel, such as our train simulation, are common. Define a simulation framework in which such simulations could be implemented as easily as possible and test it on our example.
10. Does process priority have any effect on its behavior with respect to Semaphore?
11. What would happen if we removed a Semaphore with a queue of waiting processes?
12. What happens when you close the simulation window without terminating the two vehicle processes?
13. Study class Semaphore and provide detailed answers to the following questions:
    a. What happens to a process when it sends wait and the Semaphore is ready to let it run?
    b. What happens if the Semaphore is not ready to let the process run?
    c. What happens to the process that sends signal to a Semaphore?
    d. What happens when several wait messages are sent to the same Semaphore?
    e. What happens when several signal messages are sent to the same Semaphore?

13.4 Making train simulation layout customizable

We now have a working train simulation but we would like to make it more flexible by allowing the user to customize it. In particular, we would like to make it possible to stop simulation and grab the line representing a track and move it to a new position without changing its length, and grab an endpoint of a track and move it to change the track's length.

In both cases, horizontal lines will remain horizontal and vertical lines will remain vertical. Both actions will use a special cursor displayed only when the mouse moves over a track or a track endpoint (Figure 13.12).

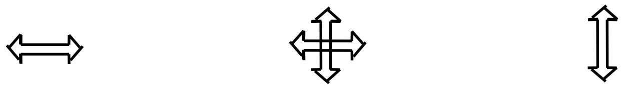


Figure 13.12.Special cursor shapes for dragging tracks and changing their length.

The details of the desired behaviors are described in the following scenarios:

<u>Scenario 1: User moves the cursor over the layout subview</u>
1. *User* moves cursor into the hot area around the right end of the horizontal track.
2. *Program* changes cursor to the shape shown on the left in Figure 13.12.
3. *User* continues moving the cursor until it leaves the hot area.
4. *Program* changes cursor to normal shape.
5. *User* moves mouse cursor into the hot area surrounding the inner portion of the horizontal track.
6. *Program* changes cursor to the shape shown in the center of Figure 13.12.
7. *User* moves mouse cursor into the hot area surrounding the inner portion of the vertical track.
8. *Program* maintains cursor shape unchanged.
9. *User* moves mouse cursor into the hot area around the upper end of the vertical track.
10. *Program* changes cursor to the shape shown on the right of Figure 13.12.

<u>Scenario 2: User drags the horizontal track to a new position</u>
1. *User* moves mouse cursor into the hot area surrounding the inner portion of the horizontal track.
2. *Program* changes cursor to the shape shown in the center of Figure 13.12.
3. *User* presses the <operate> button and moves the mouse while holding the button down.

4. *Program* drags the track maintaining its length and horizontal orientation, and follows the cursor without changing its shape. The program also moves semaphore lights and the vehicle and enforces constraints such as keeping the track within the subview.
5. *User* releases the mouse button.
6. *Program* stops dragging the track.

Scenario 3: User drags the left end of the horizontal track to a new position
1. *User* moves the cursor into the hot area surrounding the left end of the horizontal track.
2. *Program* changes cursor to the shape shown on the left of Figure 13.12.
3. *User* presses the <operate> button and moves the mouse while holding the button down.
4. *Program* drags the end of the track, restricting motion to horizontal displacement and following the cursor without changing its shape. The program also moves the vehicle if necessary and enforces constraints such as limiting the track to the subview. The rest of the track is not affected.
5. *User* releases the mouse button.
6. *Program* stops dragging the endpoint of the track.

The scenarios show that the details are not as trivial as we may have thought - and this is one of the reasons why scenarios are useful. Among other things, the definitive solution will require a specification of the details of the constraints on track motion. We suspect that the details will not be trivial and we will thus ignore them in the first iteration (leaving them as an exercise:), concentrating on unconstrained motion and leaving the details for another iteration.

Preliminary Design

The major modification of the existing implementation is that we must replace the current NoController with a controller that can handle the desired interactions. We will call this controller LayoutController and its essential responsibilities are as follows:

1. Control the shape of the cursor.
2. As the mouse moves in the dragging mode of operation, keep redrawing the appropriate part of the layout without affecting the rest. This task consist of
   a. continuously redrawing a *track*,
   b. redrawing the *vehicle* on the track being moved,
   c. redrawing *semaphore lights* of both tracks, and
   d. updating instance variables describing the tracks.

To get a better understanding of the details, let's refine our scenarios into class level conversations.

Scenario 1: User moves the cursor over the layout subview
1. *User* moves cursor into the hot area around the right end of the horizontal track.
2. LayoutController tracks mouse position and changes cursor to the appropriate shape.
3. *User* continues moving the cursor until it leaves the hot area.
4. LayoutController tracks mouse position and changes cursor to normal shape.
5. etc.

Scenario 2: User drags the horizontal track to a new position
1. *User* moves mouse cursor into the hot area surrounding the inner portion of the horizontal track
2. LayoutController tracks mouse position and changes cursor to the appropriate shape.
3. *User* presses the <operate> button and moves the mouse around while holding the red button down.
4. LayoutController detects mouse button press and starts converting mouse movement events into the following actions:
   a. Calculate mouse displacement with respect to the previous position.
   b. Obtain current track position and other relevant information from TrainSimulation and combine it with mouse displacement to calculate new parameters.
   c. Send new layout parameters to TrainSimulation.

   d. Calculate rectangles that must be redrawn and ask LayoutView to invalidate them immediately.
5. *User* releases the mouse button.
6. LayoutController senses mouse button release and changes its mode of operation to 'no dragging.'

Scenario 3: User drags the left end of the horizontal track to a new position
Very similar to Scenario 2; only the details of parameters and calculations are different.

The preliminary Object Diagram updated from previous section is shown in Figure 13.13.

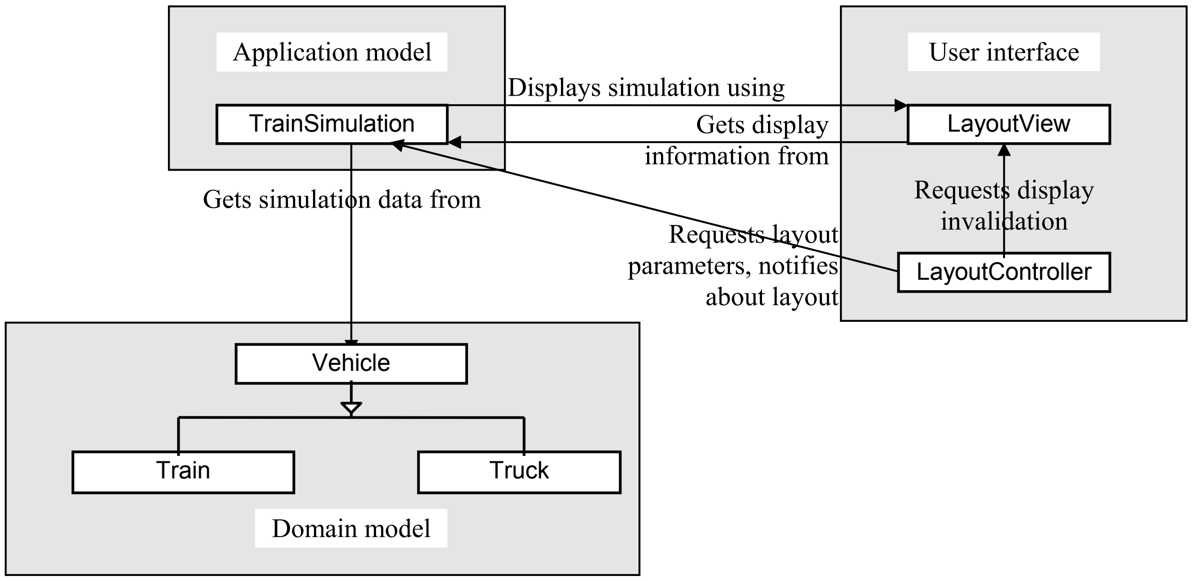

Application model
TrainSimulation
Displays simulation using
Gets display information from
User interface
LayoutView
Requests display invalidation
Gets simulation data from
Requests layout parameters, notifies about layout
LayoutController
Vehicle
Train
Truck
Domain model

Figure 13.13. Preliminary Object Model of customizable train simulation.

Design Refinement

Class Descriptions

The existing classes don't acquire any new responsibilities but this does not necessarily mean that we don't have to change them. In our existing implementation, we cached layout parameters in the view and initialized them once and for all; this would not work any more. Those cached layout parameters that affect the display and the simulation now have to be accessed via accessing methods, and the code that uses them must change accordingly. We are getting a lesson in the useulness of using accessing methods rather than accessing variables directly.

Since the essential responsibilities of existing classes don't change, we will not rewrite their descriptions and limit ourselves to the controller class.

LayoutController. Responds to mouse events by displaying appropriate cursor shapes and performing drag operations on tracks and their end points. Does not have an <operate> menu.
Superclass: Controller
Components: endArea (Rectangle) holds the shape of the hot area for end points, isDragging (Boolean) keeps track of current mode of operation, crossArrowsCursor (Cursor), horizontalArrowsCursor (Cursor), verticalArrowsCursor (Cursor), tracking (Symbol such as #horizontalTrack or #topVerticalEnd) - keeps track of the kind of area under the cursor, newCursorPosition (Point) - current position, oldCursorPosition(Point) - cursor position before last mouse move event

Behaviors Collaborators
- initialization - initialize cursors and define end point area
- event handling - intercept cursor movement and red button events

- cursor display - display area-dependent shape
- dragging - drag tracks or their endpoints in response to mouse move event LayoutView
- accessing – obtain layout parameters from domain model TrainSimulation

The Hierarchy Diagram updated from the previous section is shown in Figure 13.14; the Object Model does not change.

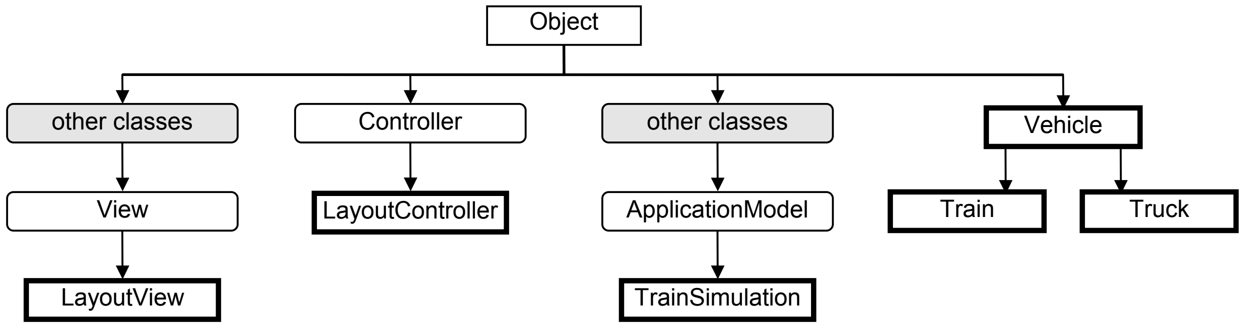


Figure 13.14. Class Hierarchy Diagram with classes to be designed shown as heavy rectangles.

<u>Implementation</u>

We will now show some of the code of LayoutController. Note again that this is a first iteration and that we are leaving many details such as dragging constraints and optimized invalidation for the next iteration.

<u>Class</u> **<u>LayoutController</u>**

Initialization

```
initialize
"Initialize mode of operation, cursors, and hot area rectangle for track end points."
        isDragging := false.
        horizontalArrowsCursor := Cursor
                                image: self class horizontalArrows asImage
                                mask: self class horizontalArrowsMask asImage
                                hotSpot: 8 @ 8
                                name: 'horizontal arrows'.
        verticalArrowsCursor := Cursor
                                image: self class verticalArrows asImage
                                mask: self class verticalArrowsMask asImage
                                hotSpot: 8 @ 8
                                name: 'horizontal arrows'.
        crossArrowsCursor := Cursor
                                image: self class crossArrows asImage
                                mask: self class crossArrowsMask asImage
                                hotSpot: 8 @ 8
                                name: 'horizontal arrows'.
        endArea := -4 @ -3 corner: 4 @ 3
```

As in the Chess program, we used Image Editor to create cursors and their masks.

Mouse events

These methods redefine some of the events protocol inherited from class Controller.

```
mouseMovedEvent: event
"The mouse has moved. Respond according to current mode of operation."
        | areaCursor |
        isDragging
                ifTrue: "Perform method corresponding to parameter being tracked."
                        [newCursorPosition := self sensor cursorPointFor: event.
                        self perform: tracking.        "Execute message appropriate for parameter."
                        oldCursorPosition := newCursorPosition]
                ifFalse: "We could have entered a new area. Check and update cursor."
                        [areaCursor := self hotSpotAreaCursorFor: (self sensor cursorPointFor: event).
                        Cursor currentCursor: areaCursor]
```

The auxiliary method hotSpotAreaCursorFor: selects cursor shape according to cursor location:

```
hotSpotAreaCursorFor: aPoint
"Determine which parameter is being tracked from the location of the cursor. Return appropriate cursor."
        self getTrackingParameterFor: aPoint.
        ^self selectCursor
```

where

```
getTrackingParameterFor: aPoint
"Return symbol representing tracking parameter."
        (self bottomVerticalTrackAreaEndpoint containsPoint: aPoint)
                ifTrue: [^tracking := #bottomVerticalEnd].
        (self topVerticalTrackAreaEndpoint containsPoint: aPoint)
                ifTrue: [^tracking := #topVerticalEnd].
        (self verticalTrackArea containsPoint: aPoint)
                ifTrue: [^tracking := #verticalTrack].
        (self leftHorizontalTrackAreaEndpoint containsPoint: aPoint)
                ifTrue: [^tracking := #leftHorizontalEnd].
        (self rightHorizontalTrackAreaEndpoint containsPoint: aPoint)
                ifTrue: [^tracking := #rightHorizontalEnd].
        (self horizontalTrackArea containsPoint: aPoint)
                ifTrue: [^tracking := #horizontalTrack].
        ^tracking := nil
```

determines which parameter we are tracking, and

```
selectCursor
        (#(#leftHorizontalEnd #rightHorizontalEnd) includes: tracking)
                ifTrue: [^horizontalArrowsCursor].
        (#(#bottomVerticalEnd #topVerticalEnd) includes: tracking)
                ifTrue: [^verticalArrowsCursor].
        (#(#horizontalTrack #verticalTrack) includes: tracking)
                ifTrue: [^crossArrowsCursor].
        ^Cursor normal
```

finds the appropriate cursor. We think that the two last methods could be implemented more elegantly and leave this task as an exercise. Individual hot areas are returned by auxiliary methods such as

```
bottomVerticalTrackAreaEndpoint
"Calculate hot area for the bottom of the current vertical track by shifting the pre-initialized endArea rectangle
by a value equal to the end point of the vertical track."
        ^endArea translatedBy: (self truckTrack at: 2)
```

where truckTrack is an array of track end points calculated dynamically from current track coordinates as follows:

```
truckTrack
```

```
"Return top and bottom end points of vertical track."
        | start |
        ^Array   with: (start := model truckTrackStart)
                 with: start + (0 @ model truckTrackLength)
```

Returning now to red button mouse events, the following method is responsible for starting dragging:

```
redButtonPressedEvent: event
"User pressed red button. If we are 'tracking' (cursor in a hot area), change mode to 'dragging' and initialize
oldCursorPoint for further updates of layout during dragging. The rest is done by mouseMovedEvent:."
        tracking isNil      "Are we 'tracking' a parameter - is cursor in a hot area?"
                ifFalse:    [isDragging := true.
                            oldCursorPosition := self sensor cursorPointFor: event]
```

and once this method is executed, dragging occurs via the mouseMovedEvent: defined above. When the user releases the button, the following event method returns operation to the non-dragging mode:

```
redButtonReleasedEvent: event
"User released red button, switch mode to non-dragging."
        isDragging := false
```

Dragging

Dragging operations are performed only in the dragging mode (see mouseMovedEvent: above) and are based on invalidation. As an example, dragging of the entire horizontal track is implemented as follows:

```
horizontalTrack
"User moved the cursor in the dragging mode. Drag the horizontal track to reflect the new cursor position."
        | shift track trackStart trackLength |
        "Update affected track parameters in the model - TrainSimulation."
        shift := newCursorPosition - oldCursorPosition.
        trackStart := model trainTrackStart + shift.
        model trainTrackStart: trackStart.
        trackLength := model trainTrackLength.
        "Calculate the end points of the track for the view."
        track := Array with: trackStart with: trackStart + (trackLength @ 0).
        "Ask LayoutView to invalidate the area around the track immediately."
        view invalidateRectangle: (self horizontalTrackInvalidationAreaForTrack: track)
                repairNow: true
```

We use horizontalTrackInvalidationAreaForTrack: to calculate the rectangle to be invalidated by LayoutView. Since the current iteration is just a proof of concept, we define the area simply as the rectangle given by the old and the new cursor position, the track, and 30 pixels around on all sides. This area includes all semaphore lights, the vehicle, and the text. It is, of course, unnecessarily large since we only need to redraw the track itself, the vehicle, the lights, and the text, and the combination of these rectangles is only a fraction of the rectangle that we are invalidating. We leave an optimized implementation as an exercise.

```
horizontalTrackInvalidationAreaForTrack: anArray
"This is the area to redraw. It is too generous but it works. Optimization is left until next iteration."
        ^Rectangle origin: (anArray at: 1) + (-30 @ -30)
                corner: (anArray at: 2) + (30 @ 30)
```

As another example of dragging, the left end of the horizontal track is dragged as follows:

```
leftHorizontalEnd
```

```
"Drag the left end point of the horizontal track. Position of vehicle is not changed because it is relative to the
start of the track."
        | shift trackStart trackLength track |
        "Recalculate affected TrainSimulation parameters."
        shift := newCursorPosition x - oldCursorPosition x.     "Movement is restricted to horizontal."
        trackStart := model trainTrackStart + (shift @ 0).
        trackLength := model trainTrackLength - shift.
        model trainTrackStart: trackStart.
        model trainTrackLength: trackLength.
        "Calculate both end points of track."
        track := Array with: trackStart with: trackStart + (trackLength @ 0).
        "Ask view to redraw the affected part."
        view invalidateRectangle: (self horizontalTrackInvalidationAreaForTrack: track)
                repairNow: true
```

The method for dragging the right end is slightly simpler because it affects only the length and not the start of the track:

```
rightHorizontalEnd
" Drag the right end point of the horizontal track. Position of vehicle is not changed because it is relative to
the start of the track."
        | shift trackStart trackLength track |
        "Recalculate affected TrainSimulation parameters."
        shift := newCursorPosition x - oldCursorPosition x.     "Movement is restricted to horizontal."
        trackStart := model trainTrackStart.
        trackLength := model trainTrackLength + shift.
        model trainTrackLength: trackLength.
        "Calculate both end points of track."
        track := Array with: trackStart with: trackStart + (trackLength @ 0).
        "Ask view to redraw the affected part."
        view invalidateRectangle: (self horizontalTrackInvalidationAreaForTrack: track)
                repairNow: true
```

The two methods for dragging track end points are almost identical and it would be better if we factored out the shared behavior. This would make the code more readable and guarantee that if we make changes, they will affect both methods identically. This stylistic improvement is left for the next iteration.

The remaining dragging methods follow the same pattern and the only other required code is simple accessing methods.

Main lessons learned:

- To simulate dragging of geometric objects, monitor the mouse-moved event and repeatedly erase the object in its original position and redraw it in the new position. Use invalidateRectangle: aRectangle repairNow: true to obtain clean response..An alternative approach is to use the animation protocol in class VisualComponent.
- Using accessing methods instead of accessing instance variable directly makes code easier to modify.

Exercises

1. Test and explain what happens if we don't specify immediate invalidation.
2. Complete and test the implementation.
3. Optimize the calculation of the invalidation area and implement constraints on moving a track and its end points. Note that if the user moves the cursor fast and the invalidation rectangle is not sufficiently large, invalidation may not be satisfactory.
4. Extend the controller to allow the user to grab a crossing light and move it to a new position. Its corresponding other light should move accordingly.
5. Extend the controller to allow the user to reposition a vehicle on its track. Note that the operation of the crossing imposes constraints on allowed vehicle configurations.

Conclusion

Computer operation generally requires controlled 'simultaneous' operation of more than one program. Such parallel threads of execution are called processes. Our definition of the concept of a process is that a process is a program that shares the CPU and other resources with other programs under the control of a process scheduler. VisualWorks processes use pre-emptive process scheduling which leaves a process to run until it terminates or suspends its execution or is suspended by a higher priority process or a semaphore. VisualWorks processes can be divided into two broad categories - system processes and user processes. Applications written by users and code segments executed from the workspace are normally executed as user processes and system processes include tasks such as garbage collection and input/output tracking..

VisualWorks processes are instances of class Process. Processes are always derived from blocks and the creation protocol can be divided into methods that create a process and schedule it for execution immediately, and methods that create a process but don't schedule it for execution. The first ones are defined in BlockClosure, the second in class Process.

Processes can be active, suspended, or terminated and the methods that change process state are suspend, resume, and terminate. Message terminate shuts down and destroys the process, suspend takes the process out of the queue of runable processes but does not destroy it, and resume puts a suspended process at the end of the queue of runable processes.

Every Smalltalk process has a priority expressed as an integer number between 1 (the lowest) and 100. User processes run by default at priority 50, I/O processes such as mouse and keyboard events run at priority 90, real-time processes run at priority 100, and system background processes run at priority 10. The fork creation message creates a Process at the priority of the parent process which created it, message forkAt: priority creates a Process at the specified priority. Both fork messages immediately schedule the new process which means that it is placed at the end of the queue of runable processes of their priority.

Smalltalk processes are managed by Processor, the single instance of class ProcessScheduler. Processor always selects for execution the first process in the highest priority non-empty queue of runable processes. Since Processor controls all priority queues, its protocol provides a very useful complement to other process control messages. Sending yield to Processor moves the currently active process to the end of its queue and activates the process at the head of the queue.

Processes often need to coordinate their operation in a way similar to traffic control at an intersection. Smalltalk's process coordination is based on analogy with traffic lights implemented by class Semaphore. The essence of the operation of a Semaphore are messages signal and wait. A signal is a notification by a Process to a Semaphore that it no longer needs its protection and corresponds roughly to turning a traffic light green.

When a process wants to gain access to a resource protected by a Semaphore, it sends the semaphore the wait message which is equivalent to attempting to proceed through and intersection and to turn the traffic light of the other street red. If the difference between signal and wait messages previously sent to the Semaphore is not positive, the sending Process is suspended until another process sends signal.

When several processes run, a deadlock can occur making it impossible for any of the processes to continue. As an example, a deadlock occurs when process X is suspended, and process Y waits for X to send signal to its Semaphore.

In the last section of this chapter, we extended our process-oriented example of train simulation with interactive control of the graphical layout. To simulate dragging of geometric objects, we monitored the mouse-moved event and repeatedly erased the object from its original position and redrew it in the new position. We used invalidateRectangle: aRectangle repairNow: true to obtain fast response without missed display operations. The same principle (erasing and redrawing) is the basis of all computer animations.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

Process – represents a thread of execution that monopolizes the CPU and other computer resources.
ProcessScheduler - class whose single instance Processor is in charge of scheduling processes. The basis of the whole operation of Smalltalk.
Semaphore - a process-synchronizing class used to restrict access to a shared resource. An analog of traffic lights.

Widgets introduced in this chapter

slider - interactive graphical representation of a floating point number model.

Terms introduced in this chapter

process - independent thread of execution that takes turns using the CPU and other computer resources with other processes; created from and executing a block
process scheduler - object controlling the order in which processes are activated
process priority - a number between 1 and 100 which determines relative priority of a process; the highest priority runable processes are executed first; 1 is the lowest process priority
runnable process – a process ready to be executed and waiting for its turn
semaphore - an object controlling activation of processes sharing a common resource; computer equivalent of traffic lights
scheduling - controlling the order in which processes execute
signal - message to a semaphore equivalent to changing a traffic light to green
wait - message to a semaphore equivalent to attempting to pass an intersection and set the traffic light of the other street to red; can only be satisfied if the semaphore is currently “green”
suspending a process - stopping the execution of a process and taking it out of the queue of runable processes without destroying it; can be reversed by resume
resuming a process - returning a suspended process to its priority queue of runable processes
terminating a process - terminating process execution and destroying it
yielding a process - sending the currently active process to the end of its priority queue