Chapter 4
Designing Object Systems

It's easy to think about object-oriented programming as no big thing. It's got objects, classes, and inheritance, but a fast machine, a good hacker, and a few cases of Jolt can make any programming language usable, right? To a certain extent, that's certainly true: You can just sit down and start hacking Squeak like any other programming language.

The problem comes later. Can you still maintain your code? Can you or anybody else reuse your classes in another application?

Just building with objects doesn't insure that you get reusable, maintainable code. Object-oriented programming makes it *easier* to happen, but there are definitely good and bad object programs. Consider a large program that has only one class and creates only one instance. Why should that be any easier to maintain than the equivalent program in an imperative programming language?

There is a process to follow that can lead to a good object-oriented program, that is, one that *is* reusable and maintainable. No process or methodology can *guarantee* a good result. In the end, it really comes down to the programmer thinking through the problem and the program. What a good process or methodology can offer is a set of activities that have worked well for others.

The focus of this chapter is on how to do object-oriented analysis and design. A single chapter can only provide an introduction. *Many* books have been written on the subject of object-oriented analysis and design. The approach of this chapter is to provide a *minimal* process. It's compatible with most of the methodologies and processes out there. It's complete enough that it does actually provide something useful. After presenting the basic process, several examples are presented that apply the process.

1 The Object-Oriented Design Process

There are a variety of definitions of a good object-oriented design process. The basic stages of the one that we'll use are:

1. **Object-oriented analysis**: In the analysis stage, your goal is to *understand the domain*. What are the objects in this domain? What are the services and attributes of each? In other words, what does each object *do* and what do they *know*? How do the objects interact with one another? Analysis is completely programming language independent. The real world is not written in *any* programming language! You don't *need* a programming language here, and trying to

remain language independent at this stage will allow you to switch languages easily later.

2. **Object-oriented design**: In the design stage, your goal is to *figure out the solution*. You get down to nitty-gritty at this point. What instance variables do you need? What methods do you need? There is some contention in the field as to whether object-oriented design can be language independent or not. Try to remain as independent as possible, but at some point, the detailed design involves issues of how existing classes are provided and structured. At that point, your design becomes language dependent.
3. **Object-oriented programming**: Finally, you build the code. That's what we've been talking about thus far in the book, and now we step back and get to the earlier parts of the process.

A very important lesson is that *this is not a linear process*. You are expected to go back and forth between analysis and design, between design and programming, or even going all the way back to rethinking the domain (in the analysis stage) while programming. There are software engineers who argue that it is simply not possible to be able to define all of the specifications before implementing. You can't know all the analysis details before designing, and you can't know all of the designing details before programming. You will go back and forth. That's not a bad thing—it's the way that even experts do it. Iterating on the design is actually one of the signs of an expert designer.

1.1 Object-Oriented Analysis Stage

Throughout all of analysis, what you're really *doing* is figuring out what you're *doing*. You may have a problem statement that you're starting from in any project, but that problem statement is *always* ill-defined and incomplete. You have to fill in the blanks even on the problem itself. What is it that you really want to do? What is really the domain in which you're working?

1.1.1 Brainstorming Candidate Classes

We will use two kinds of activities in our object-oriented analyses. The first one is simply *brainstorming*. Try to write down all the objects that you can think of that relate to the domain in which you are working. The only rule is that everything you write down should be a *noun*, since your focus is on the objects, not the tasks at this point.

A good object should have *attributes* and *services*. If two potential objects differ only in their value for the attribute, then they will be reflected as one *kind* of object (that is, a *class* when we go to implement the objects). If Fred and George are going to be represented in our system, but they only differ because of values of attributes like *name* and *address*,

then they should be reflected as one kind of object. If, however, Fred is a Fireman and George is a Policeman, and you're building a model of an emergency response system, then Fred and George do reflect different services and perhaps different attributes, so they should be different candidate objects. (But you should probably be thinking Fireman and Policeman objects, not Fred and George.)

After you have brainstormed all the possible objects, start filtering and sorting them.

- Separate those objects that have to do with the problem domain from those having to do with the human interface. In most cases, the interface objects are not part of understanding the domain. Most objects in the real world don't have an extrinsic interface, like pencils and soda cans. The interface is their physical structure or shape. Other objects have interfaces that are important but don't relate to the basic services of the object. Refrigerators have to have a handle on the door to be able to get things out, but the handle doesn't interact with keeping things cold. A radio does have a volume knob, but the radio still has to tune a station and play regardless of what the volume is currently set to. Focus on the core capabilities, not how some future user will interact with those capabilities.
- Are some of the candidate objects really attributes of other candidate objects? For example, a Name is rarely an object itself, though it's often an attribute of a class like Person.
- Are some of the candidate objects really subclasses of some other candidate objects? This does not mean that you throw any of the candidate objects out. But it does mean that you can start thinking about structuring your hierarchy of potential objects early on.
- Are some of the candidate objects really instances of some other object? Think about general objects here, even if there will only be one instance of the given object when you finally design the sytem.

The final filter of your brainstormed candidate objects gets back to what is it that you really want to do. Which of these objects are *really* ones that you want to be dealing with? Some of your candidate objects really belong to some other domain. Some of your candidate objects may be related to your domain, but adding them in at the beginning is going to be hard. Be realistic. For example, designing a cash register system can easily extend into updating inventory, making entries in accounts receivable, and immediately informing the head office of every sale. While a great system might involve all those pieces, don't try to do everything at once. What's the minimum objects that you need that lead to functionality that you need?

1.1.2 Class Responsibility Collaboration (CRC) Cards

The second activity in analysis is *CRC cards* as a way of defining the responsibilities for each class that I have defined. CRC cards were invented by Ward Cunningham and Kent Beck as a way of exploring how classes interact with one another and provide services to one another while performing various scenario tasks. CRC cards are really useful because they're concrete (i.e., physical, manipulable) and so easily shared in groups.

Typically, CRC cards are common 3x5 index cards (Figure 1). Across the top, you write the class name. You make two columns, one for Responsibilities of this class, and another for the Collaborators of this class. That's all there is to a CRC Card.

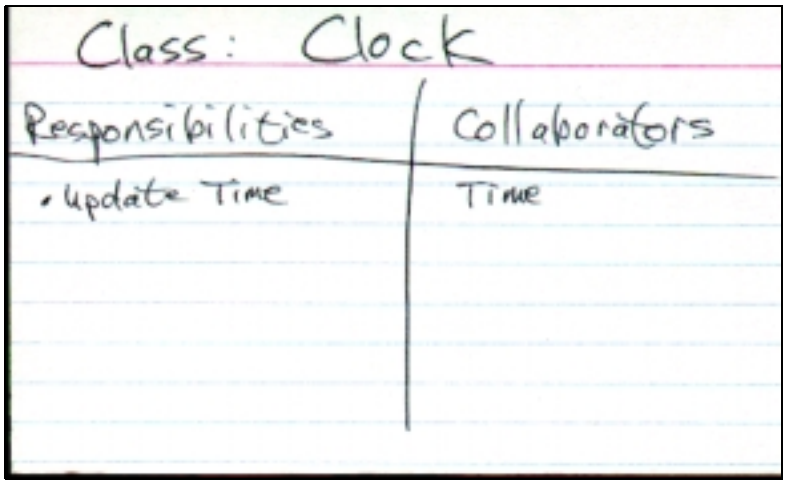

Figure 1: A Typical CRC Card

You are *strongly* encouraged to use real, physical, paper-based 3x5 index cards. Part of the fun of CRC cards is arranging them, fiddling with them, and tossing them around. The physicality of the cards is really part of the method. But if you insist on doing things virtually, a CRCcard morph is available (by the author and Lex Spoon) for manipulating the cards on-line. The source is available on the CD. Once filed-in, you can create new ones by simply choosing New Morph from the World Menu, then choosing CRCCard from the Morphic Windows submenu.

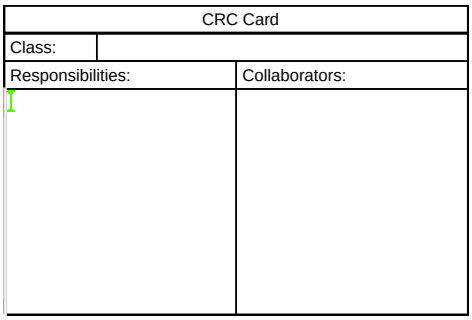

Figure 2: A CRC Morph

Here's what you do with CRC cards.

- Write the name of each class that you plan to define at the top of its own card.
- If you want, you can go ahead and start writing responsibilities for that class on the card.
- Invent some scenarios—functions or sets of activities that go together that you will want your set of objects to handle. If you were designing an inventory system, your scenarios might include handling a delivery and collecting materials for an invoice. If you were designing a class registration system, your scenarios might include checking that a student has the pre-requisites for a class and making sure that there was room in a class.
- Now, play with your cards. Walk through each of your scenarios and use the cards to identify who is responsible for what.
    - What object will get the initial message that starts the scenario process? Lay that card down first on a table or some other large playing surface.
    - What will the object be responsible for? If it's not written in the responsibilities, write it down now.
    - What other objects will that object need to work with? As you encounter each object in the scenario, lay down its card, and iterate through the process. What are its responsibilities? Who will it need to collaborate with?
    - As objects leave the scenario, pick the relevant card back up. Lay down new cards as their objects enter the scenario.

The CRC cards can help you check that you've covered all your responsibilities, that you understand the interactions between the objects, and that the responsibilities make sense. They have other useful attributes, too.

- The cards form a useful record of your early design thoughts. In a big project, capturing why someone made the design decisions they did can be very useful.
- You can play with your cards in a group! Walking through scenarios with CRC cards can be done with a whole development team or even non-programming stakeholders (like, maybe the customer?) gathered around the table. CRC cards are non-technical, so there is no language or notation to learn. Instead, it's all about talking through the process as objects, and talking with the developers and other stakeholders is a great way of making sure that everyone understands the objectives.

1.2 Object-Oriented Design Stage

What the analysis stage hands over to the design stage is a set of objects with their responsibilities and collaborators. These objects will most probably become classes, where the actual objects in the system will be instances of these classes. By making the object definitions into classes, we can create as many of the object as we want and handle growth and complexity in the system.

The outcome of the design stage is a description detailed enough to code from. For this book, the result of the design will have two parts:

- A class diagram defining the attributes and services of each class and formally identifying the connections between each class.
- A detailed description of what each service is supposed to do.

There are many different forms of the detailed description. Peter Coad, an author and object-oriented design methodologist, likes an "I am..." notation. *"I am a Count. I know how to increment. To increment, do..."* Others like notations like Activity, Sequence, or Collaboration Diagrams, which are notations in the new UML Standard. Still others use flowcharts and pseudocode. In this book, we will forego yet another notation to settle for just a natural language description.

There are just as many different class diagrams that one might use, but there is now an accepted standard for this notation. There is an emerging standard for class diagrams, and many other kinds of diagrams, called UML for *Universal Modeling Language*. UML was invented by a group of developers who had different methodologies for object-oriented design and were interested in coming up with a single uniform process. UML has been approved as a standard by the Object Management Group (OMG).

UML is amazingly powerful: It has notations for various stages of analysis (for example, there are notations for describing scenarios) as well as design stages. In fact, a theorist recently showed that you can actually

compile UML correctly to an object-oriented language now! In general the different class diagrams are fairly similar

There are several tools out there for creating and manipulating UML diagrams:

- The standard UML tool (literally, the one in which the standard is first implemented) is Rational Rose. A demo version is available for free at http://www.rational.com. While Rational Rose is a complete implementation of UML, it's also an enormous and complicated program.
- Another popular UML tool is Together, by Object International (http://www.oi.com) which includes Peter Coad. A version of Together is also available for free from http://www.togethersoft.com. The neat thing about Together is that it ties the software to the diagrams: Updating one updates the other. Together does come in an "Enterprise" version which can support multiple languages, but the free version is specific to either Java or C++ (you get to choose which you want). For creation of UML diagrams, either will work.
- Most of the diagrams in this chapter were produced using BOOST, a tool especially designed for student object-oriented designers and programmers by Noel Rappin. BOOST is written in Java. BOOST is available on the accompanying CD, and is available for download at http://www.cc.gatech.edu/gvu/edtech/BOOST/home.html. BOOST actually supports CRC card analysis as well as class diagramming in design.

We have already seen a UML class diagram in the previous chapter, copied to Figure 3. Each class becomes a rectangle in this notation. The class rectangle is split into three parts. The class name appears at the top. The middle section lists the *attributes* of the class, and the bottom section lists the *services* of the class.

The lines between the class boxes indicate *relationships*. Relationships are about all the different ways that two objects can interact with one another. How can one object be related to another? There are three main kinds of relationships that we'll talk about:

- The first is a generalization-specialization or IsA relationship. This relationship indicates that one object (class) is a specialization of another class, where the other class is a generalization of the first class. The NamedBox IsA Box, which means that the NamedBox is a specialization of the more general Box. This often gets implemented later as a superclass-subclass relationship.
- The second is an association or HasA relationship. This relationship indicates that one object has another and uses it. The Box HasA Pen that it uses for drawing. A Car HasA Engine, a Student HasA

Transcript, and a Person HasA Job. The UML standard points to another similar relationship: Aggregation. Aggregation is part-whole. The sum is *only* the collection of its parts. Aggregation is symbolized with a diamond. Aggregation is not used often, so we won't say more about it now.

- The third is a dependency or TalksTo relationship. We don't have any examples in the Box Microworld, but it's not hard to understand. Sometimes, you have one object that sends a message to another, but it isn't a part-whole relationship. For example, your Computer TalksTo a Monitor, though you might be more tempted to say that your Computer HasA Monitor. A dependency relationship is sort of a temporary HasA. It's indicated in UML with a dashed line.

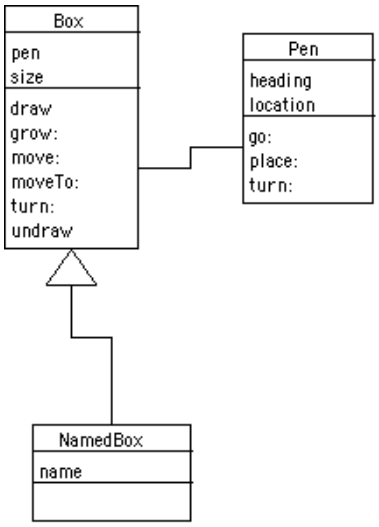

Figure 3: Example UML Class Diagram of Box Microworld

The UML class diagram depicts each of these relationships. The triangle symbol indicates a generalization-specialization relationship. That appears between NamedBox and Box to indicate that the `NamedBox` **IsA** `Box`. The line indicates an association relationship between the Pen and the Box.

There is an additional distinction that we will sometimes use in our class diagrams in this book, that is, the distinction between *concrete* and *abstract* classes. A *concrete* class is one that you will actually create instances of. An *abstract* class is one that you create only to define functionality that you will inherit in other classes. You never create instances of an abstract class. In a UML class diagram, an abstract class has the word "abstract" in the top pane of the class box on the diagram.

The UML class diagram definition includes lots of other distinctions and features, which we will not use in this book. That is not to say that the additional distinctions are not useful, but the philosophy here is to provide

a minimal notation. Some of the useful features of the UML class diagram definition that we will not be using include:

- *Multiplicities*: Knowing that two objects is related is fine, but it is even more useful to know the numbers on the relationships. For example, does a Box have one or more Pens? Can it have none? Can a Pen be shared by more than one Box? The full definition allows for placing numbers and ranges of numbers on the relationship line to indicate the number of objects related.
- *Constraints*: There are frequently constraints on objects, e.g., an Order has more than one ShippingRequest associated with it only if one of the Items ordered is back ordered. UML allows for putting constraint descriptions on a diagram in curly braces.
- *Navigability*: In an association or part-whole relationship, typically one object knows the related object(s). When Object A knows Object B, we say that the navigability of the objects is from A to B. An arrow is drawn on the association line from A to B.

2 Your First Design: A Clock

Let's do our first design example: A Clock. We'll design a good, old-fashioned, not complicated clock. It's a simple artifact that we are all users of, so we're all qualified to be valid analysts of it. The goal, however, is to produce something reusable and maintainable so that we can develop more things with it.

2.1 Doing it Quickly…and Wrong

Let's do a quick design of the Clock—which we identify up front will be wrong. Watch for it—where does the design go wrong?

First, we brainstorm the pieces of a clock. What goes into a clock?

- A face for the clock, to read the time from.
- Some kind of internal ticker that keeps the clock moving at a regular interval.
- Probably some internal representation of hours, minutes, and seconds. (Maybe lower level even than that?)
- Some of way of mapping between the internal ticker into seconds, and then every 60 of those, into mintues, and then every 60 of those, into hours.
- Some kind of knob for setting the clock.

Let's pause there and start filtering. We should filter out the face for the clock and the knob for setting the time, for now. Both of those are about the human-computer interface. Our clock must have a way of

presenting the display-able time, and for setting the current time, but the interface to those methods comes later. For now, let's ignore those pieces.

The rest of it, plus the pieces for displaying time and setting the time, seem like a reasonable definition of a Clock class. We can identify several *instance variables*

- For tracking time: **seconds**, **minutes**, and **hours**.
- For deciding how to display the time (e.g., 12 or 24 hours): **displayFormat**.

Similarly, we can define several methods.

- For accessing the time variables: **getSeconds**, **setSeconds**, **getMinutes**, **setMinutes**, **getHours**, **setHours**
- For ticking the clock: **nextSecond**.
- For getting the time in the appropriate display format: display (the time), **setFormat** .
- Maybe something for getting the time in some kind of raw form, and for setting the time: **getTime**, **setTime**.

2.2 Object-Oriented Analysis of the Clock

There. Did you see it? Did you notice when we started making design process mistakes? That last example had several of them.

- When did we decide to do only a single class, **Clock**? Putting everything in a single class is a bad idea for lots of reasons. It centralizes responsibility and authority, which makes it hard to work on in a group and which doesn't take advantage of object-oriented programming. Further, it makes it hard to reuse. Think about a real clock, say, a clock radio that wakes you up in the morning. Don't you think that there are components of that clock (e.g., some chip, some display) that are used in other devices, too? Shouldn't our clock also be made up of reusable devices?
- We started out with data, listing all the instance variables, rather than thinking about what our class should do, and even before that, what its responsibilities are.
- We jumped to using words that are specific to given programming languages. As long as possible we should remain language independent. We should talk about attributes (that might get mapped to instance variables in Smalltalk and Java, or member data in C++) and services (that might get mapped to methods or member functions), because these describe the objects, not the implementations.

2.2.1 Brainstorming a Clock

We can use on some of those previous analyses, but we really do need to start over. So let's brainstorm again what makes up a clock. But this time, let's consider all the relevant *objects*.

- A Display which would be responsible for displaying the time,
- A Time for tracking hours, minutes, seconds and their relationships,
- A Ticker or SecondsTimer for providing constant time pulses,
- A Clock which would be responsible for tracking time and displaying it on request.

Again, we'll pass on the Display object for now, as being in the realm of human-computer interfaces. But the rest seem like a reasonable assortment of pieces to begin with. Now we need to flesh out the responsibilities of each candidate object and its collaborators. We use the term “candidate” because we can still change our minds to reject some or add some. CRC cards are good for this.

2.2.2 CRC Cards for a Clock

We need some scenarios to use in our CRC card analysis. Here are two relevant ones that seem to capture the most critical pieces in our design.

- When the Ticker ticks out a time pulse, an internal counter must increment, which must increment seconds, minutes, and hours increments as needed.
- When a display is requested, the appropriate format for the display must be determine, and the time must be gathered, then converted (as necessary) to the appropriate format.

Let's play these out using CRC cards. First, we play out a new time pulse.

- Control begins with the Ticker. We’ll call it a **SecondsTicker** because we don't really care (here) about time at levels less than a second. It needs to tell the clock that a second has gone by. We write down this responsibility for the **SecondsTicker**, and we note that it needs to collaborate with the **Clock**.

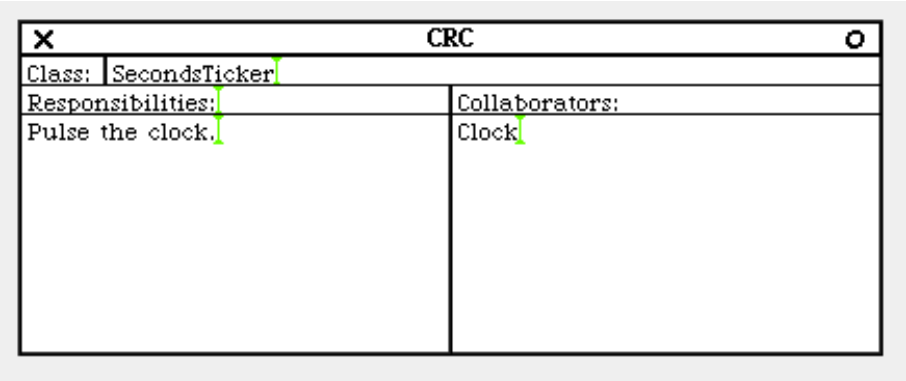

Figure 4: The First CRC Card in the Ticker Scenario

- Now the **Clock** enters the picture. It needs to accept the pulse from the **SecondsTicker**. It doesn't really have a collaborator for that—the **SecondsTicker** is initiating the action. But the **Clock** also must increment the representation of **Time**, so that is a collaborator.

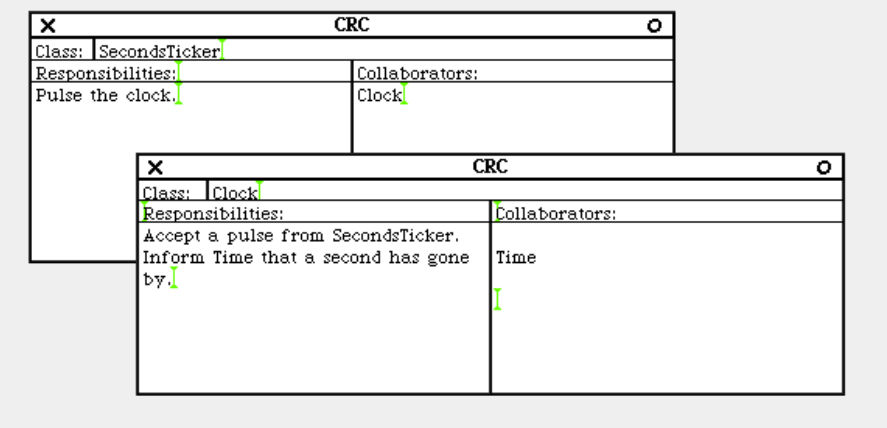

Figure 5: Adding the Clock to the Ticker Scenario

- Now **Time** is informed that a second has gone by, so it must increment its seconds representation, which may in turn trigger the representations for minutes and hours. **Time** doesn't need any collaborators to do this. It would return control to the **Clock**, which is done, and then return control to the **SecondsTimer**, which is done.

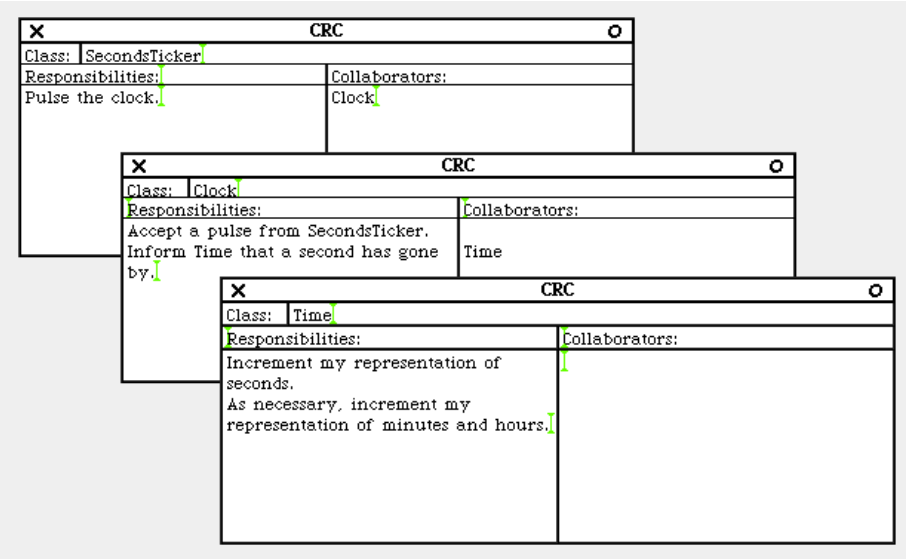

Figure 6: Last stage in CRC analysis of Ticker Scenario

We have now defined several roles and interactions between objects. Let's walk through the next scenario and add to these: When a display is requested.

- When a display is requested, the **Clock** needs to get the time, so **Time** is a collaborator. Note that **Clock**'s other responsibilities and collaborators remain. In the end, we must design **Clock** for all its responsibilities.

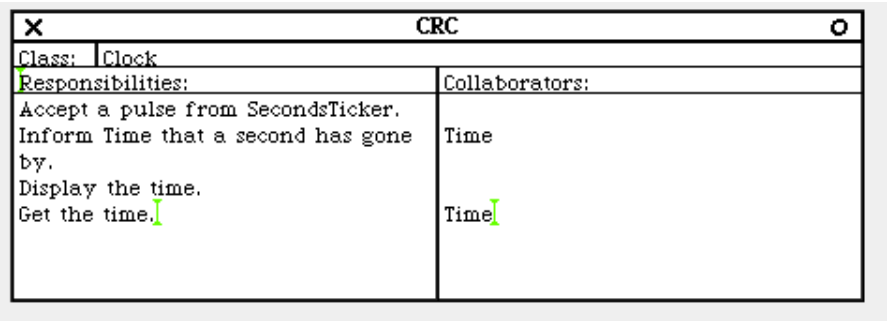

Figure 7: Clock enters the Display Scenario

- **Time** must return the time in a format that the **Clock** can manipulate, since it will be the **Clock**'s responsibility to format it.

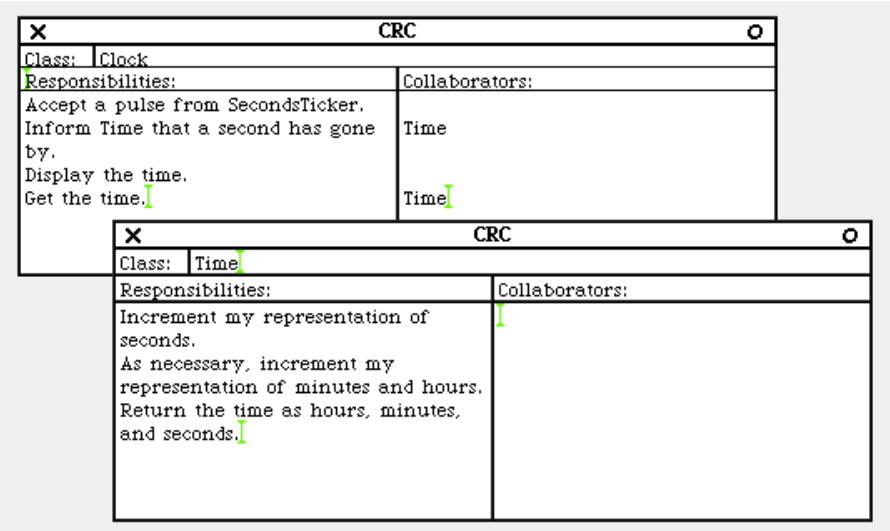

Figure 8: Clock collaborates with Time in the Display Scenario

- Time is then done, and the Clock must format the time appropriately, and then return it to the caller for display.

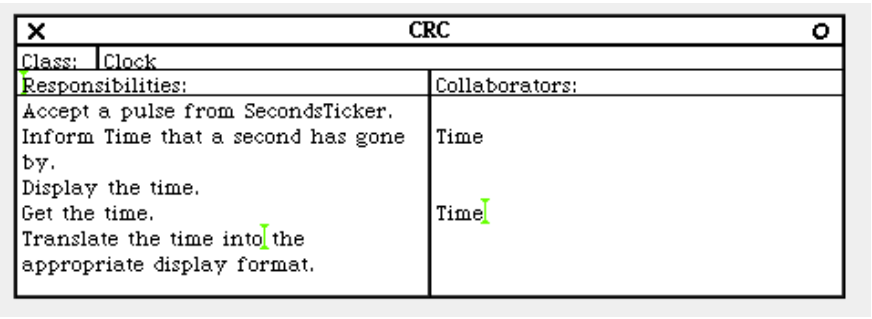

Figure 9: Clock completes the Display Scenario

2.3 Object-Oriented Design of a Clock

At this point, we have some CRC cards that tell us about our classes. We now have a pretty good idea about what each object is going to be responsible for, and what it's not responsible for (that is, what is passed on to its collaborators). We can now talk about what each class *knows* and can *do*.

- *Clock*: The Clock has to be able to set the **displayFormat** (and thus know it, too) and return time in a given format. It needs to be able to respond to a **nextSecond**, and pass that on to **Time**. Clearly, it needs to know about **Time**.

Now, according to our scenarios and CRC Card analysis, the **Clock** does not actually collaborate with the **SecondsTimer**, so the **Clock**

doesn't really need to know about that object. But thinking about it again, it's clear that we did not include a significant scenario. (This is an excellent example of having to step back to the analysis stage and re-think things from there.) What happens when you *start* the clock? When you start the clock, it's really the clock's job to start the timer, too. How will the clock start the timer if it doesn't know about it? It becomes useful for the **Clock** to know its **Timer** when you think about starting and stopping the **Clock**. Starting and stopping the clock is really about asking the timer to stop firing.

- *SecondsTimer*: The **SecondsTimer** has to know its **clock**, in order to be able to tell it when a second has passed. The **SecondsTimer** has to be able to turn on and off (start and stop). It is probably going to use some kind of external process to generate the timing signals, so it will need to know its **process**.
- *Time*: Time must be able to track the hours, minutes, and seconds. It must be able to increment the number of seconds, and have that addition flow into the other units, too. It really doesn't need to know about any other objects.

We characterize the relationship between the **SecondsTimer** and **Time** with the **Clock** as *association* relationships. The **Timer** and **Time** are each *used* by the Clock. The **Clock** *has* them. Yes, they are distinct entities, but it's also clear that **Clock** contains both a **Timer** and a representation of **Time**.

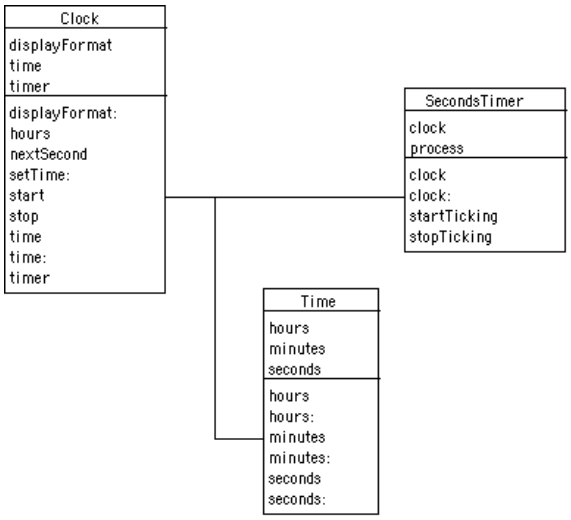

Figure 10: UML Class Diagram for Clock Design

CautionaryNote: We don't always put all the accessors in the class diagram, at least, not at the object-oriented design (OOD) stage. Obviously, all the accessors that you want have to actually be implemented, but they can be left implicit at the OOD stage.

Similarly, we do not always specify the attributes necessary to implement whole-part relationships at the OOD level. It's not incorrect to do it, but it's also okay to leave it to the programmer to figure out how to implement the relationships specified in the model.

In this book, we're the Analysts, Designers, *and* Programmers, so we might as well specify everything fully.

All of this leads to the UML class diagram in Figure 10. Given the UML diagram and the CRC analysis previously, it's possible for someone to move into the language *dependent* aspects of design and actually program our clock. That's our first complete object-oriented design.

Now, in this model, the **Clock** doesn't actually know what hour it is. It can find out by asking its **Time** object. What if you decide that you *want* the **Clock** to know what **hour** it is? What if the user interface that you connect later needs to be able to ask the clock for the hour? What you do then is to use a technique called *delegation*. The Clock *can* provide service in response to the message **hour**, but it does it by simply asking its **Time** to provide the **hours**. That's delegation—asking another object to perform the requested service for the requested object.

SideNote: In a sense, *inheritance* is just a form of *delegation*, where the delegation is implied whenever a message is sent to a subclass and the subclass has not provided a method to override the superclasses' method. Delegation is really the key to *polymorphism* where one message can be handled by any number of classes.

2.3.1 Considering An Alternative Design

An expert designer actually explores many other alternatives to any given design. All the alternatives may not be written down, nor may even appear in a CRC analysis or UML diagram. But expert designers at least *consider* other ways of doing a design.

We should do the same. What are the other ways that we could do the design? Different designs with basically the same functionality are often referred to as different *factorings* of the solution, like when you move variables around an equation without changing its value. Let's brainstorm a bit about different factorings of the **Clock** solution.

What about dropping the **Clock** object and just have the **SecondsTimer** talk directly to **Time**? That removes a middleman, which is often a good idea. However, that kind of a model is somewhat less reusable and less reflective of the real world. Think ahead to when we create variations of the **Clock**, like the **AlarmClock**. Do we subclass **Time** to create **AlarmTime**? But isn't an **AlarmTime** exactly the same as regular **Time**? (While it may seem that **AlarmTime** is awfully fast when you hit the snooze button, the reality is that it's still about seconds,

minutes, and hours.) And when we need **Time** in other contexts (e.g., as a timestamp for when something happens), does it make sense for **Time** in those other contexts to know how to respond to the next second pulse from **SecondsTimer**? Also think ahead to creating a user interface on the **Clock**. Does it make sense to have a user interface on **Time**? Maybe it does. The question that you have to ask yourself is whether **Time** and **Clock** are separate objects, or are really the same.

What about dropping the **Time** object and have the **Clock** know about seconds, minutes, and hours? That's the way that we originally tried to do the design in Section 2.1. That could work, and there would be no user interface problem. But **Time** is still a useful object all by itself, and without the context of a **Clock**. Time is actually a *reusable* object. It's better to keep it separate and just use its services for the **Clock**.

SideNote: The question of when to make two separate objects or whether to combine them into one comes up frequently for an object-oriented designer. There are two questions you have to ask yourself. Do these two objects have different *behaviors*? Do these two objects have different *data* (attributes), or are they the same data with different values? If they don't have different behaviors and they only differ in values for the same attributes, then combine them. But if they do differ in behavior or attributes, separate them.

Exercises: Reviewing the Clock Design

1. Would this design change if we were talking about an analog clock (two or three hands on a face with 12 numbers on it) rather than a digital one? How would it change?

2. How would this design change if we wanted a millisecond resolution on the clock instead of seconds resolution?

2.4 Object-oriented programming for the Clock

Now let's build in Squeak some of those objects we just designed. The definition of the class **Clock** is pretty straightforward.

```
Object subclass: #Clock
        instanceVariableNames: 'time timer displayFormat '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'
```

As is the definition of the class, **SecondsTimer**.

```
Object subclass: #SecondsTimer
        instanceVariableNames: 'clock process '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'
```

But in Smalltalk, we don't need to implement **Time**. **Time** is a pre-defined class in Smalltalk, and it already does all the things that we need it to do. We may need to change some of our method definitions as we go along—that's part of the language dependent aspect of design. But the responsibilities remain the same.

2.4.1 Implementing SecondsTimer

Let's talk about what goes on inside the Clock bottom-up, starting with the **SecondsTimer**. The **SecondsTimer** needs some accessor methods for manipulating its clock instance variable.

```
clock
        ^clock


clock: aClock
        clock ←aClock.
```

But the really tricky part of the **SecondsTimer** is how it ticks off seconds. Smalltalk already knows all about multi-processing. You can have many different processes running at the same time in Smalltalk. What we need the **SecondsTimer** to do is:

Create a **process** that sends the **clock** the message **nextSecond** after each second.

Stop that process when requested.

This turns out to be pretty easy using Smalltalk's built-in classes. To create a process, you simply send a block the message **newProcess**. It is created in a frozen state. To get the process started (out of its frozen state which it is in when it is created), we tell it to **resume**.

We need a process that will forever wait a second, send a message, then go on. **[true] whileTrue: [].** is an effective infinite loop. There is already a class named **Delay** that can create instances for different time durations (as we saw in the last chapter when we made Joe the Box animate well). When an instance is told to **wait**, it pauses its owner process for that long.

Processes have *priorities*, where a higher priority gets more CPU attention than a lower priority. Our timing process needs to do very little, and only once a second (which, in CPU time, is very little time at all). We don't want our timing process to conflict with other processes, so we'll make it a low priority. The class **Processor** defines a bunch of priority levels. We'll take a low, background priority.

```
startTicking
        process := [[true]
                whileTrue:
                        [(Delay forSeconds: 1) wait.
                        clock nextSecond.]]
                newProcess.
        process priority: (Processor userBackgroundPriority).
        process resume.
```

Stopping the process is even easier. Because we create a reference to the process in the instance variable **process**, our **SecondsTimer** just has to tell the process to stop, that is, **terminate**.

```
stopTicking
        process terminate.
```

2.4.2 Implementing the Clock Class

The **Clock** does very little beside talking to its pieces. First it needs some accessor methods (including one to delegate **hour** to **time**.)

```
hours
        ^time hours
time
        ^time


timer
        ^timer
```

Starting and stopping the **Clock** is a matter of setting up and tearing down the timer. The first thing that you may notice is that **start** and **stop** check to see if the **timer** instance variable is **nil** (empty), and if not, the **timer** is asked to stop. And at the end of **stop**, the **timer** is set back to

nil. What's going on here is being very careful to stop the **timer**, even if (by accident) **start** is executed twice in a row. Once the **timer** process is started, it's very hard to stop unless we explicitly **terminate** it. Executing start a bunch of times in a row, without stopping any of the old timers from ticking, will leave *lots* of old processes floating around, eating up CPU time, and slowing down your system.

```
start
        timer isNil ifFalse:
                [timer stopTicking. "Stop one if already existing."].
        timer ← SecondsTimer new.
        timer clock: self.
        timer startTicking.
```

```
stop
        timer isNil ifFalse:
                [timer stopTicking].
        timer ← nil.
```

The rest of the **Clock**'s methods are manipulating the time. Setting the time turns out to be a pre-defined function in the class **Time**. The method **readFrom:** can understand time in a variety of string formats, such as '13:13' and '12:10 am'. All that we have to do is to create a **Stream** on the input string. A **Stream** is a kind of object that can be read or written efficiently one element (in our case, one character) at a time. (We saw them briefly in Chapter 2 when we looked at how files are manipulated in Squeak.) We need a **ReadStream** for reading the input string, so we simply create a **ReadStream** on our string.

```
setTime: aString
        time ← Time readFrom: (ReadStream on: aString).
```

nextSecond is a little more complicated, though it is also just a single line. There is no predefined method to increment seconds in the class **Time**. But there is an ability to add two times together. So, **nextSecond** creates a **Time** instance of only a single second, then adds it to our current **time**, and makes the result the new current **time**.

```
nextSecond
```

```
        time ← time addTime: (Time fromSeconds: 1)
```

Another responsibility of the **Clock** is dealing with displaying the time in the appropriate format. First, we need an accessor method for the **displayFormat**.

```
displayFormat: aType
        "aType should be '24' or '12'"
        displayFormat ← aType
```

The last method is the longest and most complicated: **display**. In **display**, the **Clock** instance gets the hours, minutes, and seconds from the **Time** instance (padding minutes and seconds with 0's). (Recall that the comma is the string concatenation message in Smalltalk.) If the display format is 24 hour, we drop through to the bottom and just output a string of hours, minutes, and seconds. If it's a 12 hour display format, we have to compute a 'pm' time versus an 'am' time versus a just-after-noon time which is 'pm' but doesn't require that we subtract 12 from the time.

display

```
        "Display the time in a given format"
        | hours minutes seconds |
        hours ← time hours printString.
        minutes ← time minutes printString.
        (minutes size < 2) ifTrue: [minutes ← '0',minutes]. "Must be two digits"
        seconds ← time seconds printString.
        (seconds size < 2) ifTrue: [seconds ← '0',seconds].
        (displayFormat = '12')
        ifTrue: [(hours asNumber  12)
                ifTrue: [^((hours asNumber - 12) printString),':',minutes,':',
                        seconds,' pm'].
                (hours asNumber < 12)
                ifTrue: [^hours,':',minutes,':',seconds,' am']
                ifFalse: ["Exactly 12 must be printed as pm"
                        ^hours,':',minutes ,':',seconds,' pm']]
        ifFalse: ["24-hour time is the default if no displayFormat is set"
                ^hours,':',minutes,':',seconds].
```

That’s it! Try out your clock with some workspace code. First, set up a clock and tell it to start.

```
cl := Clock new.
cl displayFormat: '12'.
cl setTime: '2:05 pm'.
cl start.
```

Try this a few times to convince yourself that the clock is running: **Transcript show: cl display**. Finally, end the clock with **cl stop**.

3 Specializing Clock as an AlarmClock

The next design project is an **AlarmClock**. Our first impulse might be to change the **Clock** class, but that would be a bad idea. Clocks are useful in themselves, and there are clocks in the real world that are not alarm clocks. There are mechanisms for modeling different *kinds* of things in Smalltalk, or any object-oriented language. In general, it's better to reuse and extend than to redesign and change.

3.1 OOA for Alarm Clock

An AlarmClock is clearly *a kind of* Clock, so we model it as a *specialization* (later, when we program it, as a *subclass*). The Clock is the *generalization* (*superclass*). We can use a CRC analysis to figure out how the responsibilities of an AlarmClock differ from that of a Clock.

The main difference is in what happens when the second arrives at which the alarm should go off.

- The **AlarmClock** will depend on its generalization to inherit the standard **nextSecond** behavior which increments the time.
- The **AlarmClock** needs a new **Time** object, one that represents the alarm time.
- When the alarm time arrives, the **AlarmClock** needs to execute the alarm behavior.

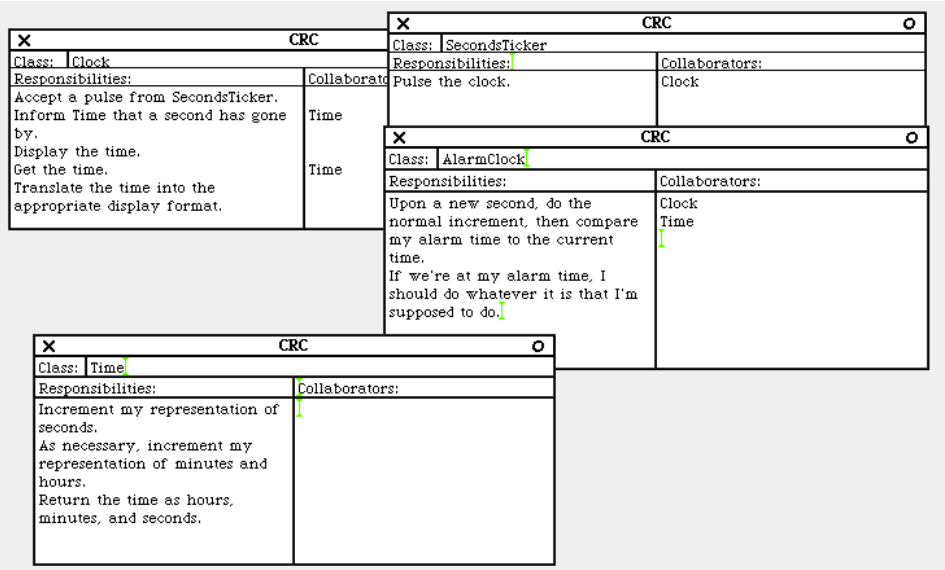

Figure 11: CRC Cards for Alarm Clock

3.2 OOD for Alarm Clock

We can now think more carefully about the definition of the class **AlarmClock**. It needs to handle responsibility for **nextSecond** itself (and within it, call upon Clock's **nextSecond** behavior). Clearly, it must know an **alarmTime**, so the **AlarmClock** has its own **Time** (HasA relationship). The **AlarmClock** needs to know what its alarm behavior is. For now, we will define that alarm behavior as an attribute, an **alarm**, so that we have flexibility later in what happens upon an alarm. We are leaving open the option of defining an **Alarm** class later, though as we will see, we will end up using the attribute for a language-dependent feature.

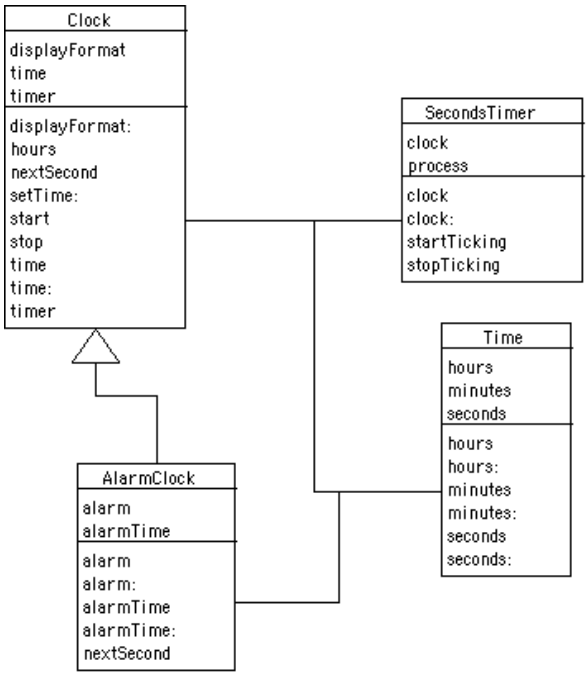

Figure 12: UML Class Diagram for Alarm Clock

The fact that we could use our **Time** class in more than one place (i.e., as the representation of running time in the Clock and in the representation of the alarm time in the **AlarmClock**) is an indication that we are doing well with the design so far. In a sense, future reuse is a good test of current designs. If we can reuse our objects in new situations, then we did come up with definitions of flexible objects, like the ones that inhabit the real world.

3.3 OOP for the AlarmClock

Before we actually implement the **AlarmClock**, let's make the language-dependent decision alluded to earlier. The **AlarmClock** must do *something* when the alarm is to go off. In the design, we simply inserted an **alarm** attributed. As we get language specific, we realize that we can just define the alarm behavior as a *block*. As we program the **AlarmClock**, we will define **alarm** as **alarmBlock**.

The **AlarmClock** class definition is then straightforward from the class diagram.

```
Clock subclass: #AlarmClock
        instanceVariableNames: 'alarmTime alarmBlock '
        classVariableNames: ''
        poolDictionaries: ''
```

category: 'ClockWorks'

The **AlarmClock** has very few methods. There are obviously some access methods.

alarmBlock: aBlock

alarmBlock ← aBlock.

alarmTime

^alarmTime

setAlarmTime: aString

alarmTime ← Time readFrom: (ReadStream on: aString).

The interesting part of **AlarmClock** is **nextSecond**. What it does is:

4. It asks its superclass, **Clock**, to handle the basic **nextSecond** responsibility.
5. Then it compares its **alarmTime** to the current **time**. If they are equal, it's time to fire off the **alarmBlock** by taking its **value**. **value** is the method that asks a block of code in Smalltalk to execute.

nextSecond

super nextSecond.

(time = alarmTime) ifTrue: [alarmBlock value].

Test the **AlarmClock** with some workspace code. When the alarm goes off, Squeak beeps three times and displays "ALARM!" in the Transcript.

```
cl ← AlarmClock new.
cl setTime: '2:04 pm'.
cl alarmBlock: [3 timesRepeat: [Smalltalk beep. Transcript show:
'ALARM!']].
cl setAlarmTime: '2:06 pm'.
cl start
```

After the alarm fires, the clock is still going. (Does your clock radio stop after you turn off the alarm?) You need to explicitly stop it with a DoIt on **cl stop**.

Exercises: Building on the AlarmClock

3. How would you implement a snooze button?

4. Design and implement an alarm clock for two people who have to get up at different times and want different kinds of alarms, e.g., different radio stations for alarms, or one wants an beep (try **Smalltalk beep**) instead of a radio wake-up call. How would you differentiate alarm actions and alarm behaviors?

4 Generating: Programming in Groups

Your code for exercises might be growing large enough that you may want to work with someone else. How do you work together? How do you merge code? How do you keep track of your changes as opposed to someone else's? The issue gets even more complicated when your changes appear in multiple classes. How do you file out all of your changes in multiple classes to give to someone else?

Smalltalk programmers have always worked together in collaborative groups, so the support for programming in groups in Squeak has its roots in the earliest Smalltalk-80. The basic two ideas are *projects* and *change sets*.

- A project stores the state of a complete Squeak desktop, including all the windows (and in Morphic, all morphic objects), as well as the currently active change set. When you change projects, whether by entering or exiting, all the global state is saved into the project being exited, and loaded from the one being entered. We have already seen a project, including entering and leaving it. Whatever changes you make to classes while within a project gets associated with the change set for that project.
- A change set is a collection of changes to the system. Changes include class changes, method changes, class removals, and method removals. All of these changes impact the *whole* system, e.g., you can't have one version of a method in one change set and another version of the same method in another change set. But you can fileOut all of the changes of one change set at once. You can also copy changes between change sets, compare change sets, and generally manage change sets.

4.1 Creating Projects and Change Sets

You have already seen how to create a project, enter it, and exit it. You can do this from any project. Projects can be nested within projects, and you can jump from any project to another.

Change sets go along with projects, but they don't have to. You can create change sets, and declare that new changes go to the new change set, from any project whatsoever.

There are two tools available in the *Open…* menu for managing changes: *simple change sorter* and *dual change sorter*. The simple change sorter (Figure 13) lets you see the list of change sets in the current system (upper left hand corner), the list of classes in the selected change set (upper right), the list of methods changed in the selected class (center pane), and the actual code in the lower pane. From any change sorter, you can use the Yellow Button Menu on the change set pane to create a new change set, fileout all the changes of a change set (so that someone else can load in all the things you've changed in a given project), or even choose to make changes go to a different change set. Even within a single project, you can choose to make changes go to a different change set.

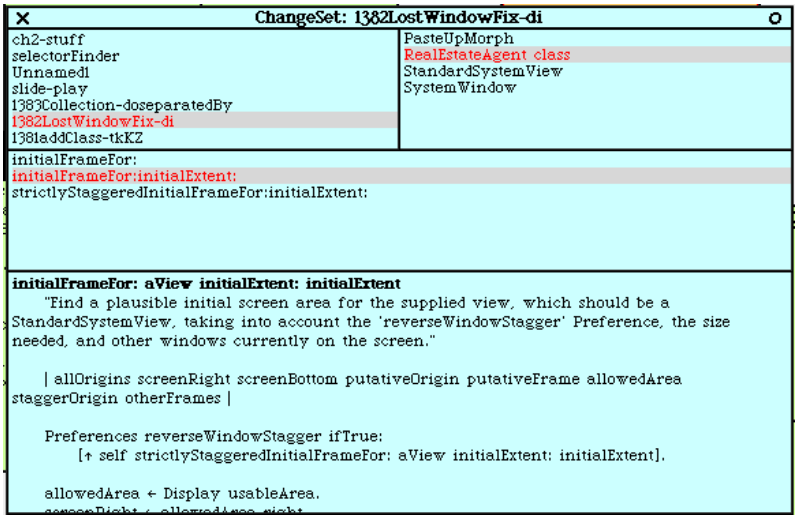

Figure 13: Simple Change Sorter

The dual change sorter (Figure 14) allows you to look at two change sets at once. With two change sets open at once, you can copy changes from one to the other, submerge one change set into another, or even subtract all the changes from one change set that are also in the other change set. By manipulating change sets, you can find any particular changes that you need to include in what you give to your collaborators.

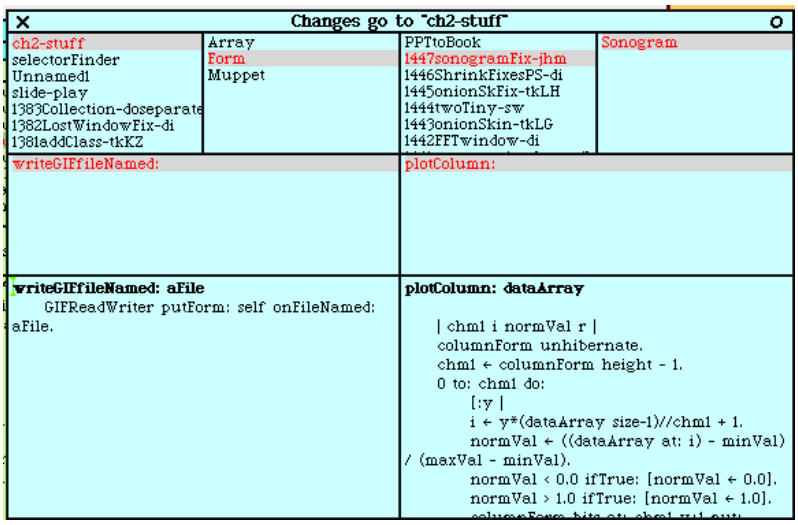

Figure 14: Dual Change Sorter

Change set files are a little different than the ".st" files from filing out classes or categories. When a change set is filed out, the filename ends with ".cs" rather than the ".st" from filing out the class or method. A changeset fileOut contains the name of the change set and the date and time of the file out. The time stamp makes it easier to track versions of changes.

It's also possible to attach a preamble or postscript to a change set. The preamble appears at the top of a change set, and a postscript appears at the end. In a preamble, you can put in a comment describing the change set, or even provide some set-up code, e.g., check that a prerequisite class is defined in the image before the change set is loaded in. In a postscript, you can clean up things, or perhaps start some of the newly loaded code.

4.2 Working with Someone Else's Changes

When someone else gives you a change set, you could simply file it in via the file list. But you might want to check it first, to see how it differs from what you already have (e.g., beware of someone overwriting your methods!) and even just to see what it includes.

From the file list, you have several options from the Yellow Button Menu when a code file (.cs or .st) is selected:

- You can *File into a new change set*. This option puts all of the new changes into one change set that you can easily inspect using a change sorter.
- You can *Browse* changes (Figure 15). This option puts up a window where you can inspect each separate change that the change set is going to do to your system if you file it in. You can choose to file in any one or any set of the changes in the change browser. In addition,

you can use the Yellow Button Menu to select just those that are already in your system (conflicts), or those that are not. (Notice that you can see the initials of the author, as well as the time and date, for each change.)

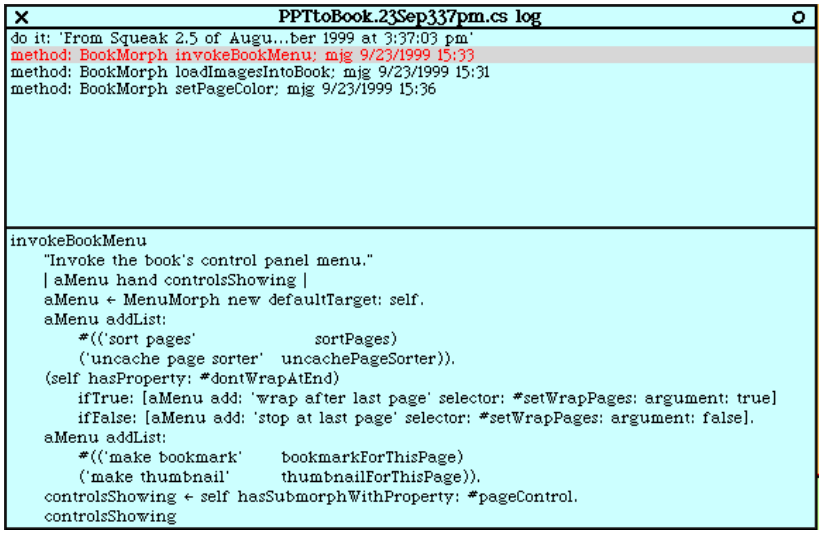

Figure 15: Browsing changes from a change set file

- Perhaps the most powerful option is the ability to *Browse code* (Figure 16). The Package Browser that opens allows you to walk through the code that's in the selected file just as if it were already in your System Browser. As you select each method, it tells you if the method is already in your image or not. If it is, strikeouts and color coding shows you how the new method differs from the method that's currently in your system. As from the changes browser, the package browser lets you file in any particular methods or classes that you choose.

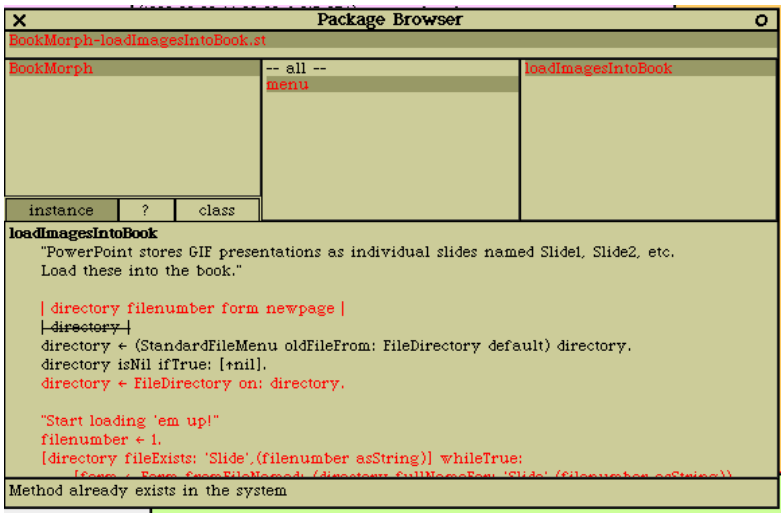

Figure 16: Browsing Code from a Code File

4.3 Recovering from Someone Else's Changes

You have now filed in someone else's code, but missed one critical method, so now all of your code is broken. (Or maybe, you just made a really bad change, and realize that you hadn't filed out the previous, almost-working version.) You really want to turn back the clock to an earlier version of the method.

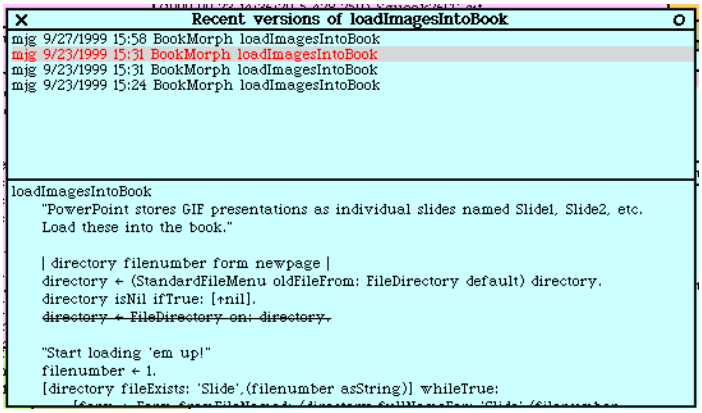

Figure 17: Recent versions of a method

Fortunately, the changes file really does capture *all* changes in the system, which includes all those previous versions of a given method. From any method list (e.g., an implementors list, a System Browser, wherever), you have a Yellow Button Menu item *Versions…* The recent versions browser shows you all the versions of the same method in your changes file (Figure 17). Selecting an older version shows you with color and strikeouts how the selected version differs from your current version. You can copy anything you want out of these old versions and paste them into your browser to re-accept them.

If you open up the *Preferences* (under the *Help* menu item in the Desktop or World menus), you can choose to *useAnnotationPanes.* In several of the browsers (especially Senders and Implementors), when annotation panes are selected, you are shown a pane between the list of methods and the code that gives useful information about the selected method. This information typically includes the initials of the author, the timestamp, the number of implementors of this method, the number of senders, and which change set the method came from (Figure 18). What gets displayed in the annotation pane is actually configurable with an easy drag-and-drop interface, which is brought up when you DoIt on **Preferences editAnnotations**,

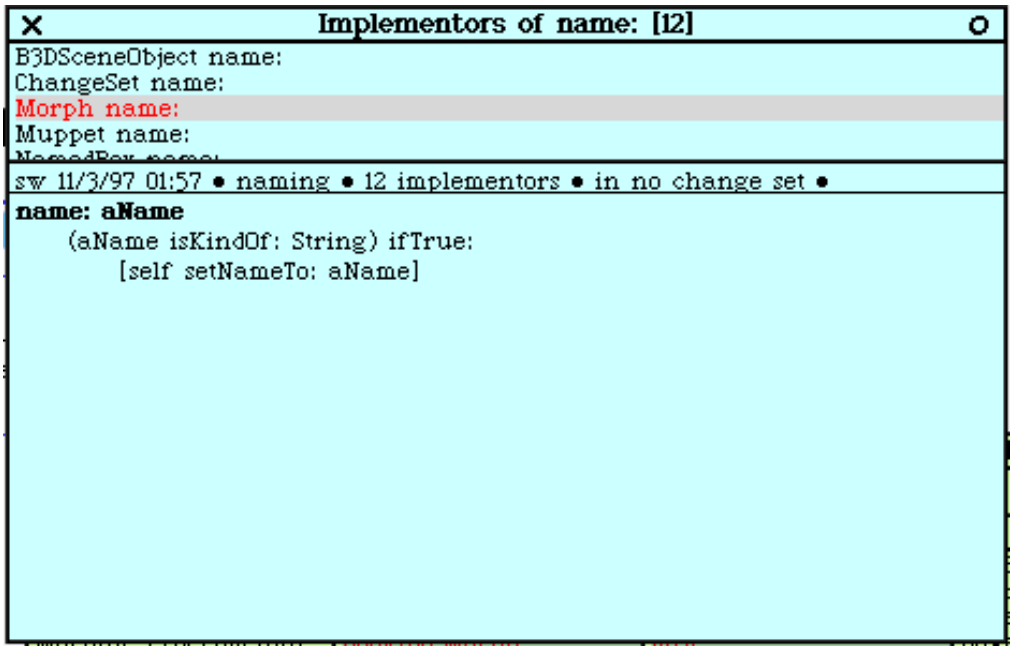

Figure 18: Implementors browser with Annotation Pane

5 Reusing the Clock and AlarmClock

The **AlarmClock** was a reasonable test of the **Clock**'s reusability, but if the design is good, it should be flexible when the larger system is reused, too. In this section, we push harder on the design of the **AlarmClock** (and **Clock**) by reusing them in yet more designs.

5.1 Reuse in a VCR

An **AlarmClock** is itself useful, but we can test this larger design (more than just the single class **Time**) by seeing if we can reuse it in even larger, more complicated projects. Let's start by talking through use of this **AlarmClock** structure in a VCR. A VCR requires something like an alarm clock to start and stop the recorder.

Brainstorming probably would lead to deciding that new objects are necessary: a **VCR** object and a **VCRRecorder**. We can then use CRC cards to work out the responsibilities of each of these objects (Figure 19).

A **VCR** has a recorder, and a current channel, and the **VCR** knows how to do things like start the record process, fast forward, rewind, change the channel, and so on. If we were out to do a really detailed model of a **VCR**, we might want to think about motors that are controlled by fast forward and rewind, and perhaps even sensors and a tape carriage for handling the recording and playing process. But since our focus is on the starting and stopping of the recorder, we'll leave that open for now.

The **VCRRecorder** is where the action is. It is the **VCRRecorder** that tells the VCR when to record on what channel and when to stop. Here's the CRC analysis.

- The **VCRRecorder** has an alarm clock for starting the recording session. When it goes off, it tells the **VCR** to go to the appropriate channel, and start recording.
- The **VCR** knows how to change channels and record.
- The **VCRRecorder** also has an alarm clock for ending the recording session. When it goes off, it tells the **VCR** to stop.

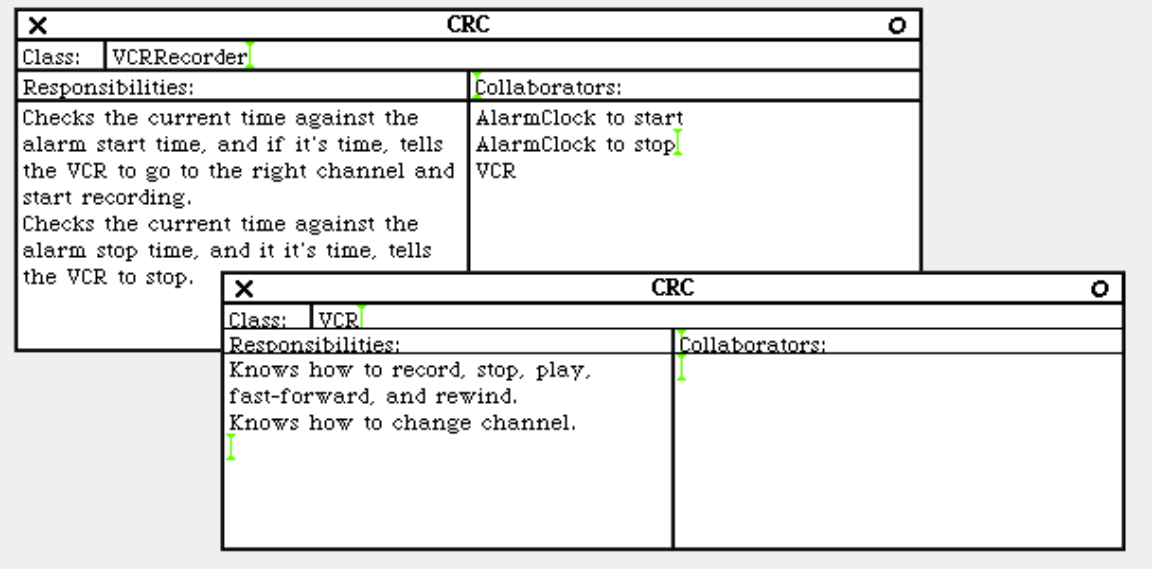

Figure 19: CRC cards for VCR and VCRRecorder Objects

This isn't a very sophisticated extension of an alarm clock. But it is interesting for several reasons. First, it's showing the power of combining objects in object systems. Look how many levels we now have in our system (Figure 20). **Clocks** have **Timers** and **Time**. **AlarmClocks** extend **Clocks** and have an additional **alarmTime**. Now **VCR**s have **VCRRecorders** which themselves have two **AlarmClocks**—and each of those has everything that an **AlarmClock** has: a **Timer** (inherited from **Clock**) and two **Time** objects. Our model is actually getting fairly complicated, but the complexity is quite manageable. When you're dealing with the **AlarmClock**, you can quite forget that the **SecondsTimer** is there at all. When you're dealing with the **VCRRecorder**, you just create a couple of **AlarmClocks**, and tell them their alarm times and what to do when the alarms go off.

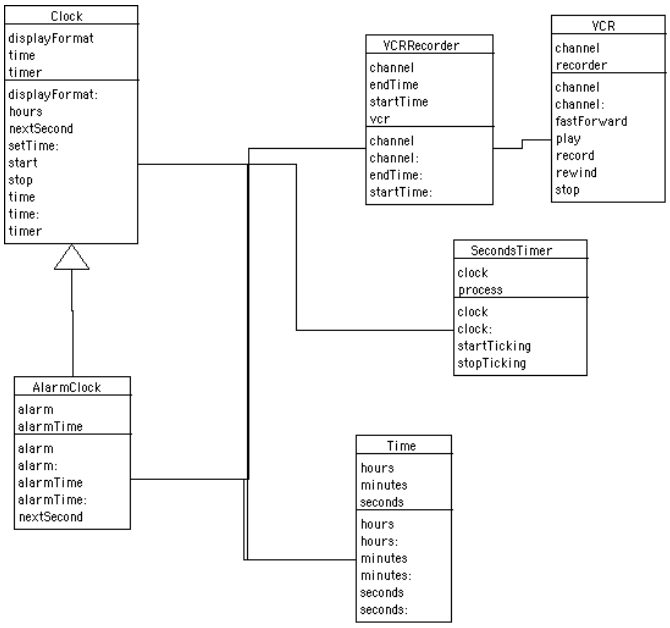

Figure 20: Class diagram for VCR design

When you ask object-oriented programmers what is powerful about object-oriented programming, they will often talk about inheritance. Being able to create a new subclass that can do so much as soon as you create it, because of inheritance, is clearly very powerful. But the real workhorse of object-oriented programming is the ability to combine objects: Connecting objects together, having bunches of objects lying underneath easy-to-use surface objects, having collections of heterogeneous objects. That's where the real power of objects to hide complexity lies.

Exercises: Continuing the VCR Example

5. If you look at the aggregation in our VCR, you'll notice some strangeness. There are actually two **SecondsTimers** in our model, each telling a different **AlarmClock** to tick. In a real VCR, these two would probably be combined to create only a single timing circuit. How would you change our diagram for a single timer? Would there still be two clocks? Or would it have a new kind of **AlarmClock** that keeps track of multiple alarms?

5.2 Reuse in an AppointmentBook

If we can reuse our **AlarmClock** in one situation, that's good. If we can reuse it in two or more, that's better. Our last reuse example of the **AlarmClock** is an **AppointmentBook**. An **AppointmentBook** is a good one because we can actually build and play with it. Unless your

computer happens to have a VCR motor and tape carriage assembly with sensors, you probably can't actually build and use the VCR example.

What are the new objects in an **AppointmentBook**? We can brainstorm a bit here:

- Maybe an AppointmentBook to track appointments.
- A Calendar to associate appointments with their days.
- Which suggests an Appointment which is responsible for alerting the user to a given appointment at the right time.
- Maybe the AlarmClock for triggering the Appointment.
- An AppointmentQueue so as to sort the appointments and set up the AlarmClock for the next appointment.

Now, we filter. We want to do this as simply as possible, so that we can set up a demonstration. But we want it to be flexible to extend later. Let's choose to eliminate the **Calendar** and the **AlarmQueue**. Instead, we'll associate an **AlarmClock** with *each* **Appointment**. It's somewhat inefficient, but it's an inefficiency that can be corrected later. It's simple, and allows us to get started quickly. We'll give the **AppointmentBook** the responsibility to alert all the **Appointments** for a given day, which could be triggered by a **Calendar** object if we ever added one.

A complete class diagram appears in Figure 21.

- **Appointments** know their alarm and the date on which they should be active. They don't need to know their time—that's *delegated* to the **alarm**.
- **AppointmentBook** knows all the appointments, and it is responsible for turning them on for a given day (and off at the end of the day), and to make an appointment for a given day.

CautionaryNote: We don't always want to model the whole. If the parts define all of the whole, and we define all the parts, we don't also need to model the whole. In this case, the AppointmentBook does more than just collect all the Appointments, so we define it separately.

Here is the class diagram that represents this. Again, the level of aggregation says something interesting about the ability to scale up object systems.

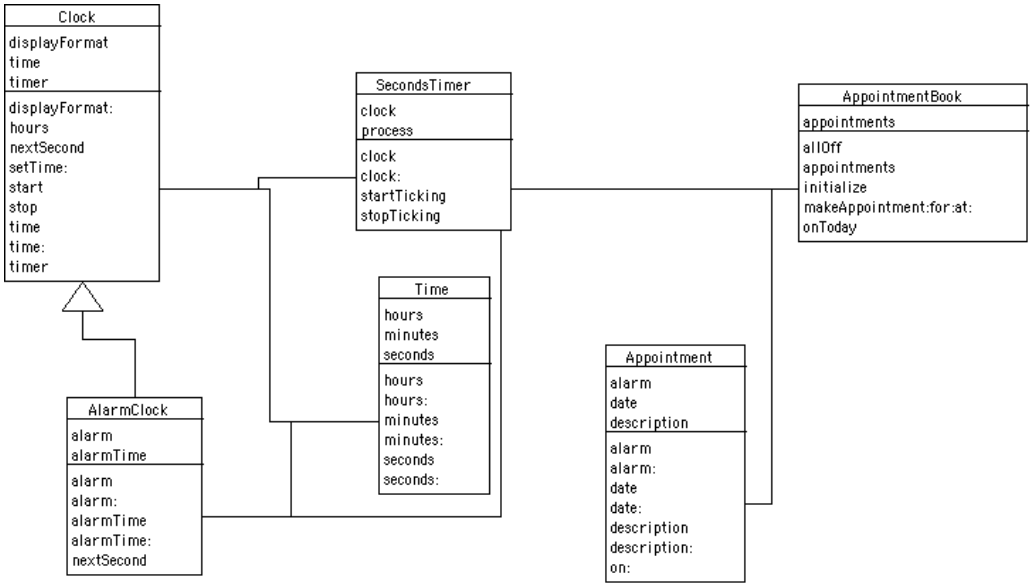

Figure 21: Class diagram for Appointments

Exercises: On the Appointment Book Analysis

6. We skipped the CRC Card analysis in the above Appointment Book example. Fill it in.

7. The **AppointmentBook** shouldn't be responsible for creating well-formed appointments. Instead, the request to make an appointment should be delegated to the **Appointment** class. How would you model that?

8. Now that we've done it the short way, figure out how to add a **Calendar** and an **AlarmQueue** that uses only a single **AlarmClock**.

5.2.1 Programming the Appointment Book

We start out by implementing the two new classes that we need.

```
Object subclass: #AppointmentBook
        instanceVariableNames: 'appointments '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'

Object subclass: #Appointment
        instanceVariableNames: 'alarm description date '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ClockWorks'
```

5.2.2 Implementing the AppointmentBook with Collections

The **AppointmentBook** has a couple of techniques in it that we haven't seen previously, so we'll discuss that one first. To begin with, an **AppointmentBook** must be explicitly initialized. The initialize method sets up the **OrderedCollection** of appointments.

An **OrderedCollection** can be thought of as an array that grows to fit as many elements are placed in it. Smalltalk has many such collections classes (e.g., **SortedCollections**, **Arrays**, **Sets**, **Bags**). Each has its own characteristics, and the implementation of each is optimized for the kind of characteristics that it supports. For example, **Set**s are unordered and allow no duplications, while **Bag**s are unordered and allow duplications.

In fact, it's not obvious that an **OrderedCollection** is the right choice for storing the collection of **Appointments** in an **AppointmentBook**. However, **OrderedCollections** provide an amazingly wide variety of services, so they are often a good choice for early implementation and prototyping. Later, a more optimal collection can be chosen later.

initialize

```
appointments ← OrderedCollection new.
```

We provide an accessor for the **appointments**, particularly for user interfaces that may want to display all of the appointments.

appointments

```
^appointments
```

When we are making an appointment via the **AppointmentBook**, we create an **Appointment** instance, fill it with the appropriate values, then add it to the **appointments OrderedCollection**. You will also notice here a reference to the **Date** class. Smalltalk's **Date** class understands how to manipulate dates and answer questions about dates (e.g., "What day of the week was July 4, 1776?" (**Date readFrom: (ReadStream on: 'July 4, 1776')) weekday** = **Thursday**). We use it for the `date` instance variable in **Appointment**, and we use it's ability to parse dates to create the appointment. Again, we must hand it a **Stream**, not a **String**, so we create one on the fly.

`makeAppointment: aDescription for: aDate at: aTime`

```
| a |
a ← Appointment new.
a description: aDescription.
a date: (Date readFrom: (ReadStream on: aDate)).
a alarm: aTime.
appointments add: a.
```

Note that our **AlarmClock** doesn't know anything about dates, only about time. It doesn't know how to go off on a given day and time, only at a given time. So turning on the alarm when the appointment is created doesn't work. We can only set the alarm on a given day. That's what **onToday** does.

onToday

```
(appointments select:
      [:each | each date = Date today])
            do: [:each | each on].
```

This is a fairly complex piece of code, so let's go slowly through it in its major pieces.

(appointments select: [:each | each date = Date today]) : All **Collections** in Smalltalk understand methods that allow one to iterate over a **Collection** and do something to or with each element in the **Collection**. **select:** is a method that evaluates a block for each element in the collection, then returns a new collection with just those elements that returned **true** for the block, as seen in Chapter 2. In this piece of code, we are selecting all of those appointments whose date is today.

do: [:each | each on] : In this piece, we are walking over each of the appointments that are due today, and telling them that they are **on**.

Recall that each appointment has its own **AlarmClock**, which keeps ticking even after the appointment is done. We need a way to turn them all off again.

allOff

```
appointments do:
      [:appointment | appointment alarm stop].
```

Exercises: Improving the Appointment Book

9. We turned only the appointments on that were due on the given day, but we turned all alarms on the appointments off. That's both inefficent (we

did the selection in the first case, but processed all appointments in the second), and non-orthogonal (we tell the **appointment** on, but we tell the **alarm** stop). Which way do you like better? Why? Fix the wrong method.

5.2.3 Implementing the Appointments

The **Appointments** keep track of the information for each individual appointment in an appointment book. The most important responsibility for an appointment is to track the time, date, and description of each appointment.

```
date
	^date

date: aDate
	"Set date of appointment."
	date ← aDate

description
	^description

description: aDescription
	description := aDescription.
```

Setting the alarm time is a little tricky. Instead of just recording the time, we use the instance variable **alarm** to actually hold the instance **AlarmClock**, and delegate tracking the alarm time to it.

```
alarm: someTime
	alarm ← AlarmClock new.
	alarm setAlarmTime: someTime.
```

We provide an accessor method to access the **AlarmClock** instance.

```
alarm
	^alarm
```

Finally, the method for turning on the alarms is called simply **on**. It creates an **alarmBlock** for the alarm, sets the **AlarmClock**'s current time to the real time, and starts the **AlarmClock**.

```
on
```

```
                "The appointment is today, so turn on alarm."
                alarm alarmBlock: [3 timesRepeat: [Smalltalk beep.].
                      Transcript show: 'Appointment: ',description.
                      alarm stop.].
                alarm setTime: (Time now printString).
                alarm start.
```

We can test the **AppointmentBook** now.

```
b ← AppointmentBook new initialize.
b makeAppointment: 'Testing!' for: '9/27/98' at: '2:34 pm'.
b onToday.
```

When you're declaring the end of the day, be sure to use **b allOff** `or you'll be leaving seconds timers running around.`

Exercises: Extending the Appointment Book

10. Write a **time** method that returns the time of the appointment. (Hint: Delegation!)

11. Why don't we set the **alarmBlock** at the time that the alarm time is set in the **Appointment** instance?

6 Implementing Models

The mapping from a class diagram to a program is fairly straightforward, but there are some decisions to be made. Probably the easiest part of the mapping is from attributes and services to instance variables and methods, but even there, some variations exist. For example, if you were modeling a class of students, you might want to have an attribute called **studentCount** to track the number of students in the class. But when you actually implement this, you may just want to define a method **studentCount** that returns the size of the collection storing the students. That way, you don't have to maintain the count during each addition and deletion, and the count is still fast and up-to-date.

The mapping of classes in the class diagram to actual classes is pretty clean, until you need interactions between objects of the same type. For example, if you had a **LinkedListNode** that pointed to itself, mapping that in a traditional class diagram is a little confusing. The problem is that the class diagram describes classes, and doesn't do a great job of describing individual objects. You need to keep this distinction in mind when creating the mapping. In UML, sequence and collaboration diagrams do a better job of showing how individual objects relate to one another.

Generalization-Specialization relations (*IsA*) always get mapped as superclass-subclass relations, but association (*HasA*) relations can be mapped in a variety of ways.

- The fact that a part (say, tires) is part of a whole (say, a car) doesn't actually tell you anything about who needs to know about what. Does the tire need to send messages to the car? If so, it needs some kind of instance variable that refers to the whole, and it needs to be initialized to set that reference to its car. Does the car need to send messages to its tires? Then, the car needs to be able to reference the tire objects. But you don't always need to go both ways. In UML, this issue is called navigability.
- If there is a many-to-one relationship, then you will probably need some kind of collection to manage the relationship. If a course is composed of students, the course object will probably need a collection of students. In our examples above, the appointment book had a collection of appointments that were composed within it. But it isn't always true that you need a collection if you have more than one part composed within a whole. For example, consider our alarm clock, which had two instances of Time within it: One for the current time (inherited from Clock), and one for the alarm time. In this case, two separate instance variables modeled the relationship better than a collection of Time objects.

There are lots of other variations for object-oriented design, and lots of other mappings from a design to a programming language. Any good book on UML which show a large number of other kinds of relationships between objects, as well as other aspects of design to consider. For example, we haven't considered the best way to involve stakeholders (the people who care about the result of a design) in a design process, nor modeling *roles*, that is, the different kinds of users. The issues covered in this chapter are a good start on design, but there’s a lot more to learn about doing good design.

7 Rules of Thumb for Good Object-Oriented Designs

The process described in this chapter *can* lead to good object-oriented design, where a good design is reusable and maintainable. Through the chapter, we’ve identified several key aspects of a good design:

- It's general and based on real world artifacts. That makes the design more reusable.
- It defines objects as nouns, not functions and not managers.
- The relationship between a subclass and a superclass is always an IsA relationship.

- It avoids computer science terms like "linked list." The real world doesn't have linked lists in it. Implementations of models of the real world do.

But this is only a partial list. There are many other characteristics of good designs. In this book, we can only touch on a handful of them through the examples and exercises.

There are lots of *rules of thumb*, or *heuristics*, that you can use to measure the quality of your design. They don't always work—sometimes, there are very good reasons for breaking a standard "rule" of design. But they work in most cases and can help in measuring up your design.

Here are several useful object-oriented design rules of thumb:

- Almost no good design consists of a single class. The real world isn't made from a single object. A program that has a single class with many services and attributes looks more like a procedural program jammed into an object-oriented language.
- In a good design, information access is enough . Objects don't need information that they can't get to. Objects can get to the information they need: Either directly, or by asking one of the objects that they can access directly.
- Responsibility, control, and communication are distributed in good designs. Not one object does everything.
- There should be little or no redundancy: Code should appear only once. Use inheritance or delegation so that you do not have to replicate code.
- There is a right level of detail for any model, and it depends on what you need. Yes, everything in the world is made up of molecules, but most problems don't require you to model each and every molecule. Create models for the things that you need.

References

An excellent and practical introduction to UML (many more diagrams than what we covered here) is *UML Distilled: Applying the Standard Object Modeling Language* by Martin Fowler and Kendall Scott (Addison Wesley, 1997).

Perhaps the best book on design in Smalltalk is Chamond Liu's *Smalltalk, Objects, and Design* (Prentice-Hall, 1996). The form of Smalltalk that he focuses on is IBM Smalltalk, but the core language is the same, and the discussion of design issues is very nice.

One of the critical themes in object-oriented design today is *design patterns*, identifying common kinds of objects and relationships between these objects. The book that started this theme is *Design Patterns:*

Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison Wesley, 1995). A Smalltalk-specific companion to the book has been written by Sherman R. Alpert, Kyle Brown, and Bobby Woolf, *The Design Patterns Smalltalk Companion* (Addison Wesley, 1998). While design patterns is a pretty abstract idea for beginners, it's an important concept for more advanced study of object-oriented design.