Designing Object Systems

Chapter 4

Squeak: Object-oriented design with multimedia applications
Chapter 4 Story

- (A Small) Object-Oriented Design Process
  - OOA: Brainstorming and CRC Cards
  - OOD: UML Class Diagrams
- Designing a Clock: OOA/D/P
- Specializing the Clock as an AlarmClock
- Generating: Programming in Groups
- Reuse in a VCR and AppointmentBook
- Heuristics and Rules of Thumb
Why an O-O Process?

- You can hack at any language, but how do you get reusable, maintainable code?
- Just using Objects doesn't *insure* good code
  - Most C++ programs have only a single class!
- No process can *guarantee* good results
  - Just makes them more likely
Object-oriented design process

- Object-oriented analysis
  - Goal: Understand the domain

- Object-oriented design
  - Goal: Design a solution, a model of the domain in which the desired activities occur

- Object-oriented programming
  - Goal: Implement the solution

- Note: A Good Design is 2/3 Before You Hit the Keyboard
It's a process, not a waterfall

- Design is an iterative activity
  - Start in OOA, go to OOD, forget something in OOA
  - Get to OOP, realize you didn't understand something and go back
- The stages are there to identify emphases
  - For example, OOA is TOTALLY language-independent
Object-oriented analysis

- Step one: Brainstorming Candidate Classes
  - Write down all the objects that relate
    - Focus on the nouns
    - Good objects have attributes and services
  - Now, filter the candidates
    - Deal with the interface later (Not part of the domain)
    - Are some candidates attributes of others?
    - Are some subclasses of others?
    - Are some instances of others?
OOA: CRC Cards

- Step two: Write CRC cards and work through scenarios
  - Class-Responsibility-Collaborator Cards (Cunningham and Beck)
  - Just 3x5 cards
How to use CRC Cards

- Make one for each candidate class
- Invent scenarios: What should these objects do?
- Play the cards
  - Lay down the card that starts the scenario
  - Write down its responsibility
  - Add collaborator objects to help with that responsibility
  - Pick up cards as they leave the scenario
Why CRC Cards?

- Help you identify objects and their responsibilities
- Help you understand interactions between objects
- Cards form a useful record of early design activity
- Cards work well in group situations and are understandable by non-technical stakeholders
Object-oriented design

- Step one: Create a UML class diagram of your objects
- Step two: Create a detailed description of the services to be performed
  - Peter Coad's "I am a Count. I know how to increment…"
  - Activity, sequence, or collaboration UML diagrams
Unified Modeling Language

- There have been O-O gurus for many years
- Three of them worked together to define UML ("Three amigos": Booch, Rumbaugh, Jacobson)
- Has now been approved as a standard by the Object Management Group (OMG)
- Very powerful, many forms of notation
  - It's even provable!
Recommended Book:
UML Distilled

- Serious O-O designers DO use UML
- UML Distilled by Martin Fowler is a great practical introduction to UML
- Official UML book series published by Addison-Wesley
UML Tools

- Lots of others
  - BOOST (Basic Object-Oriented Support Tool) by Noel Rappin, available on CD/CoWeb
  - Argo-UML, ObjectPlant, etc.
UML Class Diagrams

- Classes are boxes
  - Top is name
  - Middle are attributes
  - Bottom is services
- Lines are relationships
- Arrows show navigability
How can two classes be related?

- **Generalization-specialization** or **IsA**
  - NamedBox IsA Box
  - Diagram: Triangle on the relationship line

- **Association** or **HasA**
  - Box HasA Pen
  - Diagram: Just a relationship line
  - *Aggregation* is a *part-whole* relationship
    - Diagram: Diamond on the line

- **Dependency** or **TalksTo**
  - Dependency is sort of temporary HasA
    - Diagram: Dashed line in UML
The Aggregation-Association Issue

- It’s not clear when, in UML, you notate that one thing “has” another.
  - Coad: “An example of aggregation is the relationship between an organization and its clerks”
  - Rumbaugh: “A company is not an aggregation of its employees”
  - There’s a subtle difference there that I’m missing…
- Safest: Use an association unless the definition of “aggregation” is clear to your audience
More in UML class diagrams

- Navigability
  - Both ways or just one?
  - (Does the Box have the Pen or the Pen have the Box?)
  - Usually added later in the process

- Multiplicities
  - How many Pens per how many Boxes?
Concrete and abstract classes

- Concrete classes are those that we create instances from.
- Abstract classes are never instantiated. They exist to define structure and behavior for other classes to inherit.
- In UML, abstract classes use *italics* when naming the class.
First Design: A Clock

- Let’s do it really quickly the first time…
- Brainstorm
  - Face for the clock, internal ticker, hours, minutes, seconds, knob for setting the clock
- Filter
  - Ticker, hours, minutes, seconds
- Define the Clock class
  - Attributes: seconds, minutes, hours, displayFormat
  - Services: get/set, nextSecond, display, setFormat
Wrong!

How to do the design badly

- Already made LOTS of errors!
  - We assumed there was only one class: Clock
    - What's reusable in that?
  - We started with *data* and not with what the *object* should *do*
  - We were way too programming-ish to start!
  - Thinking in get/set -- be aware of your idioms!
    - Java/Python does that, but typically not Smalltalk
Brainstorming a Clock

- Brainstorm
  - Display, Time, Ticker/Seconds Timer, Clock
- Filter: Drop the Display
- CRC Cards
  - Two scenarios
    - When the ticker pulses the clock, the internal representation of time must increment
    - When a display is requested, the time must be fetched and formatted
Scenario #1: The Ticker Ticks

- The SecondsTicker pulses the Clock

<table>
<thead>
<tr>
<th>Class</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecondsTicker</td>
<td></td>
</tr>
<tr>
<td>Responsibilities:</td>
<td>Collaborators:</td>
</tr>
<tr>
<td>Pulse the clock.</td>
<td>Clock</td>
</tr>
</tbody>
</table>
Scenario #1: The Ticker Ticks

The Clock updates Time
Scenario #1: The Ticker Ticks

- Time updates itself—end of scenario
Scenario #2: Clock Responds with the Time

- Clock Class
  - Accept a pulse from SecondsTicker.
  - Inform Time that a second has gone by.
  - Display the time.
  - Get the time.

- Time Class
  - Increment my representation of seconds.
  - As necessary, increment my representation of minutes and hours.
  - Return the time as hours, minutes, and seconds.
Scenario #2: Clock Responds with the Time

- Clock is responsible for final formatting

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept a pulse from SecondsTicker.</td>
<td>Time</td>
</tr>
<tr>
<td>Inform Time that a second has gone by.</td>
<td>Time</td>
</tr>
<tr>
<td>Display the time.</td>
<td></td>
</tr>
<tr>
<td>Get the time.</td>
<td></td>
</tr>
<tr>
<td>Translate the time into the appropriate display format.</td>
<td></td>
</tr>
</tbody>
</table>
OOD of the Clock

Clock

- Needs displayFormat attribute, to meet the formatting responsibility
- It must respond to nextSecond, and it needs to know about Time.

Discovered Scenario ("Starting the Clock")
  - So Clock needs to know about the SecondsTicker or Timer, too.
OOD of the Clock

- SecondsTimer
  - Has to know its clock, to send the pulse
  - Somehow has to have an external process for generating timing signals

- Time
  - Knows hours, minutes, and seconds—and can do simple math with them
  - Needs no collaborators!
OOD of the Clock

UML Class Diagram

Note: How can Clock know 'hours'?

By delegation to Time
Considering alternatives

- Expert designers *always* consider other alternative designs
- What if SecondsTimer talked directly to Time, no Clock?
  - Less reusable, less like the real world
- What about dropping Time and letting Clock know seconds, minutes, hours?
  - Lose the reusability of a Time object
OOP for the Clock

Now we can be language dependent: Time is built-in to Squeak

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

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

startTicking

process := [[true]
  whileTrue:
    [(Delay forSeconds: 1) wait.
    clock nextSecond.]]
newProcess. "Make a pulser"

"Put it in the background"

process priority: (Processor
  userBackgroundPriority).

process resume. "Start it"
Implementing a SecondsTimer

stopTicking

"Stop the timer process"

process terminate.
Implementing the Clock

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.
Implementing the Clock

- Can delegate some responsibility
  - Time already can read in time formats

```smalltalk
setTime: aString
  time := Time readFrom:
    (ReadStream on: aString).
```
Implementing the Clock

nextSecond

"Increment time by adding a Time instance of only one second"

time := time addTime:
   (Time fromSeconds: 1)
Implementing the Clock

- Tough part is actually display formatting

\[
\text{displayFormat: aType}
\]

"aType should be '24' or '12"

\[
\text{displayFormat := aType}
\]
Implementing the Clock

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].
(Display, part 2)

(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].

"NOTE! THERE’S A BUG HERE! CONSIDER 1 MINUTE AFTER MIDNIGHT!"
Running the Clock

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

■ Display the Clock
  Transcript show: cl display .

■ Stop the Clock
  cl stop
STOP!!!!

- Don't look any further
- Give the AlarmClock design a shot on your own
  - What's different about the AlarmClock?
  - What can you reuse?
  - What has to be new?
Specializing the Clock as an Alarm Clock

- OOA
  - Needs some kind of AlarmTime
  - Needs some kind of AlarmThingToDo
  - Will handle the responsibility for next second differently
# OOA: CRC for AlarmClock

<table>
<thead>
<tr>
<th>Class: Clock</th>
<th>Responsibilities:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accept a pulse from SecondsTicker. Inform Time that a second has gone by. Display the time. Get the time. Translate the time into the appropriate display format.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class: Time</th>
<th>Responsibilities:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Increment my representation of seconds. As necessary, increment my representation of minutes and hours. Return the time as hours, minutes, and seconds.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class: SecondsTicker</th>
<th>Responsibilities:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pulse the clock.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class: AlarmClock</th>
<th>Responsibilities:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Upon a new second, do the normal increment, then compare my alarm time to the current time. If we're at my alarm time, I should do whatever it is that I'm supposed to do.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class: Clock</th>
<th>Collaborators:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class: Time</th>
<th>Collaborators:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OOD for AlarmClock

Clock
- displayFormat
- displayFormat: hours
- nextSecond
- setTime: start
- stop

SecondsTimer
- process
- startTicking
- stopTicking

AlarmClock
- alarm
- nextSecond

Time
- hours
- minutes
- seconds

Clock
- 1
  - clock
  - timer

SecondsTimer
- 1
  - process

AlarmClock
- 1
OOP for AlarmClock

Language dependent: How represent the AlarmThingToDo?

In Squeak: Use a block

Clock subclass: #AlarmClock
instanceVariableNames: 'alarmTime alarmBlock'
classVariableNames: "
poolDictionaries: "
category: 'ClockWorks'
OOP for AlarmClock

alarmBlock: aBlock

    alarmBlock := aBlock.

alarmTime

    ^alarmTime

setAlarmTime: aString

    alarmTime := Time readFrom:
    (ReadStream on: aString).
nextSecond delegates up to Clock, then checks the alarmTime

nextSecond

super nextSecond.
(time = alarmTime)
ifTrue: [alarmBlock value].
Running an AlarmClock

cl := AlarmClock new.
cl setTime: '2:04 pm'.
cl alarmBlock:
    [3 timesRepeat:
        [Smalltalk beep. Transcript show: 'ALARM!']].
cl setAlarmTime: '2:06 pm'.
cl start "Don't forget cl stop later!"
Pause...
Generating: Working in Groups

- **Projects**
  - Stores the state of a Squeak desktop
  - All changes within a project can be contained within a single change set

- **Change sets**
  - A collection of changes to the system, including removals and additions
  - Can be filed out as a set
Manipulating Change Sets

Change set list

Classes in the selected change set

Methods in this class

Descriptions or method text

Change set list

Classes in the selected change set

Methods in this class

Descriptions or method text

Change set list

Classes in the selected change set

Methods in this class

Descriptions or method text
### Dual change set sorter

<table>
<thead>
<tr>
<th><strong>blobs</strong></th>
<th>DavesBlobMorph</th>
<th><strong>blobs</strong></th>
<th>DavesBlobMorph class</th>
<th><strong>Class class</strong></th>
<th>Environment</th>
<th>VersionsBrowser class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unnamed1</td>
<td>1780advanceVersion</td>
<td>1779ShadowedTempFix</td>
<td>1778majorShrinkFix2-DI</td>
<td>1777majorShrinkFix1-JA</td>
<td>1776projRelabelFix-sw</td>
<td></td>
</tr>
</tbody>
</table>

The Blob started out as a simple test of the CurveMorph and ended up as an oozing, pulsating, repulsive mess which will wander across your screen until killed. Each instance has its own rate of oozing, so some are faster than others. It’s not good for anything.

15Jan00 by Bob Arning &lt;arning@charm.net&gt;
What you can do with change sets

- Create a new change set
  - Or switch to an existing one so that changes get added there
- Fileout
- Move or copy changes between change sets
- Set a preamble or postscript
Reviewing Someone Else's ChangeSet

- From the FileList, Browse changes.
- File in, select conflicts, select non-conflicts

```
x     MAT-1/19/2000.cs log

do it: 'From Squeak 2.5 of August...ry 2000 at 10:09:49 am'
do it: 'Change Set'
do it: Model subclass: #MAT/
do it: PluggableTextMorph subclass: ''/
method: HandMorph scriptingMenu: am 1/19/2000 15:39
method: HandMorph openMAT: am 1/19/2000 15:41
method: MAT openAsMorph: am 1/19/2000 14:36
method: MAT richText: am 1/19/2000 15:17
method: MAT richText: am 1/19/2000 15:17
method: MAT title: am 1/19/2000 15:12
method: MAT title: am 1/19/2000 14:34
method: MAT menu: shifted: am 1/19/2000 15:00

openAsMorph
  "open a set of windows for viewing this browser"
  | win m |  

  "create a window for it"
  win + SystemWindow labelled: 'MAT'.
  win model: self.

  "create a title view"
  win edjMorph: (m + PluggableTextMorph on: self text: *title accept: *title) frame: (0.0@0 extent: @0.1).
  m setBalloonText: 'Title of the page'.

  "create the canvas area"
  win
```
Browsing ChangeSet Code

openAsMorph
"open a set of windows for viewing this browser"
| win m |

"create a window for it"
win + SystemWindow labelled: 'MAT'.
win model: self.

"create a title view"
win addMorph: (m + PluggableTextMorph on: self text: #title accept: #title) frame: (0.0@0 extent: 1@0.1).
m setBalloonText: 'Title of the page'.

"create the canvas area"
New method
Recovering from Someone Else's Changes

Someone else's changeSet whumps your system. Now what?

- From any browser, Versions will show you past versions of the method.
- From Preferences (under Help), choose useAnnotationPanels to find out who did what.
- New in 2.7: Save for future revert and Revert to saved copy will checkpoint a project.
Reusing the Clock and AlarmClock

- VCR and VCRRecorder

<table>
<thead>
<tr>
<th>Class:</th>
<th>VCRRecorder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities:</td>
<td></td>
</tr>
<tr>
<td>Checks the current time against the alarm start time, and if it’s time, tells the VCR to go to the right channel and start recording. Checks the current time against the alarm stop time, and if it’s time, tells the VCR to stop.</td>
<td>Collaborators:</td>
</tr>
<tr>
<td></td>
<td>AlarmClock to start</td>
</tr>
<tr>
<td></td>
<td>AlarmClock to stop</td>
</tr>
<tr>
<td></td>
<td>VCR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class:</th>
<th>VCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities:</td>
<td>Collaborators:</td>
</tr>
<tr>
<td>Knows how to record, stop, play, fast-forward, and rewind. Knows how to change channel.</td>
<td></td>
</tr>
</tbody>
</table>
OOD for VCR

Clock
- displayFormat
- displayFormat: hours
- nextSecond
- setTime: start
- stop

SecondsTimer
- process
- startTicking
- stopTicking

Time
- hours
- minutes
- seconds

AlarmClock
- alarm
- nextSecond

VCR
- channel
- fastForward
- play
- record
- rewind
- stop

VCRRecorder
- channel
Reuse in an AppointmentBook

- Brainstorming:
  - AppointmentBook to track appointments
  - Calendar for tracking days
  - An Appointment itself
  - An AlarmClock for triggering the Appointment
  - AppointmentQueue for sorting the next Appointment

- Filtering: Ignore days, and start inefficiently
OOD for Appointments

- **Clock**
  - displayFormat
  - displayFormat: hours
  - nextSecond
  - setTime:
  - start
  - stop

- **SecondsTimer**
  - process
  - startTicking
  - stopTicking

- **AlarmClock**
  - alarm
  - nextSecond

- **Time**
  - hours
  - minutes
  - seconds

- **AppointmentBook**
  - (class) initialize
  - (class) makeAppointment:for:at:
  - onToday
  - allOff

- **Appointment**
  - date
  - description
  - on

- **Relations**
  - Clock --> SecondsTimer
  - Clock --> AlarmClock
  - AlarmClock --> Time
  - Time --> Appointment
  - Appointment --> appointments
  - SecondTimer --> process
  - StartTicking
  - stopTicking
OOP for AppointmentBook

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

Object subclass: #Appointment
  instanceVariableNames: 'alarm description date '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'ClockWorks'
Date class already exists

(Date readFrom: (ReadStream on: 'July 4,1776')) weekday

Thursday
Using collections in AppointmentBook

initialize

  appointments := OrderedCollection new.

appointments

  ^appointments
Making an Appointment

makeAppointment: aDescription for: aDate at: aTime

| a |

a := Appointment new. "Make an appointment, and set its attributes"
a description: aDescription.
a date: (Date readFrom: (ReadStream on: aDate)).
a alarm: aTime.
appointments add: a. "Store it"
Turning on Appointments

onToday

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

allOff

appointments do:
  [:appointment |
    appointment alarm stop].
Appointments Accessors

date

^date

date: aDate

"Set date of appointment."

date := aDate

description

^description

description: aDescription

description := aDescription.
Appointment Alarm

Accessors

`alarm: someTime`

```ooc
alarm := AlarmClock new.
alarm setAlarmTime: someTime.

^alarm
```

9/19/00 Copyright 2000, Mark Guzdial
Turning on an Appointment

"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.
Using the AppointmentBook

b := AppointmentBook new initialize.
b onToday.

- When you're declaring the end of the day, be sure to use \texttt{b allOff} or you'll be leaving seconds timers running around.
Critiquing Clocks

Critiques:

- AppointmentBook is inefficient and non-orthogonal
- AlarmClock doesn't handle multiple alarms well, and it should: For appointments, snooze, etc.
- Clock shouldn't handle Time formatting itself—let Time do it.
Lessons from Critique

Lessons:

- "It works" isn't enough
- It's **hard** to get a good design, and these are actually pretty good!
- Designs can work in one situation, and not in another
Issues in Implementing Models

- Attributes are usually instance variables and services are usually method
  - But not always: Counts are attributes that are more often computed as methods
- Interactions with instances of the same class (e.g., LinkedListNode) are hard to model in UML class diagrams. (Other diagrams useful here.)
- Many-to-one HasA usually means a Collection, but not always.
Goals for Good O-O design

- It's general and based on real world artifacts to enhance reusability.
- It defines objects as nouns, not functions and not managers.
- The relationship between a subclass and a superclass is always an *IsA* relationship.
- O-O models avoid computer science terms like "linked list."
O-O Design Rules of Thumb

• Almost no good design consists of a single class.
• In a good design, information access is *enough*
• Responsibility, control, and communication are distributed in good designs. Not one object does everything.
  • No “God” or Manager Objects!
• There should be little or no redundancy.
• Get the level of detail *right*