Sessions 22

Revisiting the idea of OO “design”
Learning to Design Programs

You can learn to design programs by **studying** good and bad designs, and coming to understand why they are good and bad.

But that isn’t enough:

- Do you think a violinist listens to music and reads a book and then steps onto stage to perform?
- “Professional writers are always writing, so it's no wonder they are good.” (Richard Gabriel)
Learning to Design Programs

You also learn to design programs by designing programs and studying the results.

- Good design comes from experience.
- Experience comes from bad design.
“Released early due to good behavior”

• We often begin by considering the behavior of the system.
• A thorough understanding of behavior normally leads to a better design
  – Which normally leads to a more efficient use of time spent coding.
Object-Oriented Design

Designing any system requires:

• naming the key components, or objects, in the system

• identifying the main responsibilities of each object

• identifying the main communication paths among them
Object-Oriented Design

We focus on responsibility and behavior first because:

• we can understand the behavior of a system by looking at interactions among parts and users
• behavior makes no commitments to implementation
• designing to behavior promotes maximum reuse
Remember our discussion about the software design cycle?

• Begin with analysis for design phase
  – What are the objects in my program?
  – In order to identify similar objects, what do they need to know (state)?
  – What actions should each object be able to take? (behavior)
  – What messages should each object be able to respond to? (behavior)
Finding Classes

• Primary goals of design phase
  – Identify classes
  – Determine relationships between classes
• Decisions require talent and experience
• Majority of classes identified in requirements and specification documents
• Nouns in requirements document provide list of potential classes
  – List must be refined
Object-Oriented Design

Several tools capture these features of a design:

• CRC cards
• UML class diagrams
• Interaction diagrams
CRC Cards

CRC cards focus us on the who and the what and discourage the how.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong>: Timed Book Morph</td>
<td>- Start TimingMorph</td>
<td>Timing Morph</td>
</tr>
<tr>
<td><strong>Super class</strong>: BookMorph</td>
<td>- Inform TimingMorph of current slide and total slides.</td>
<td>Timing Morph</td>
</tr>
<tr>
<td><strong>Subclass</strong>: PresentationMorph</td>
<td></td>
<td>Timing Morph</td>
</tr>
</tbody>
</table>
CRC Cards

CRC cards focus us on the who and the what and discourage the how.

We design by decomposing responsibilities into small, cohesive, and well-defined sets.

– When the responsibilities won’t fit on a CRC card, the set probably isn’t small enough.

– When the responsibilities are not cohesive, we split off new objects.

– When the responsibilities are not well-defined, we consider whether some other object should have some of the responsibilities.
Class Diagrams (continued)

[FIGURE 2-13] Clock class diagram

- State
- Name
- Behavior

Clock

- seconds: int
- minutes: int
- hours: int

- start()
- adjustTime()
- reset()
Class Diagrams (continued)

**Figure 2-16** Adding navigability information to an association

**Figure 2-17** Multiplicity information added to an association

Modern Software Development Using Java, 2e
Class Diagram

Student
- Name
- Address
- Phone Number
- Email Address
- Student Number
- Average Mark
- Is Eligible To Enroll
- Get Seminars Taken

Enrollment
- Marks Received
  - Get Average To Date
  - Get Final Mark

Seminar
- Name
- Seminar Number
- Fees
- Add Student
- Drop Student

Professor
- Name
- Address
- Phone Number
- Email Address
- Salary

?Some seminars may not have an instructor?
Interaction diagrams

Interaction diagrams show us how different object *collaborate* to solve a specific problem.
Object-Oriented Design

Interaction diagrams show us how different object *collaborate* to solve the problem.
Things we watch for…

• Cohesion
  – A measure of how strongly-related and focused the various responsibilities of a software module are.
  – Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability.
  – High cohesion == good
Things we watch for…

• Coupling (or dependency)
  – The degree to which each program module (class or method) relies on each one of the other modules.
  – Low coupling refers to a relationship in which one module interacts with another module through a stable interface and does not need to be concerned with the other module's internal implementation
  – Low coupling == good
The Coffee Machine Problem

Our class just got hired as the contractors to design a custom coffee vending machine. Arnold the person hiring us is the owner of Acme Sprockets and, like the common software designer, hates standard solutions. He wants his own custom design. He is, however, a cheapskate. Arnold tells us he wants a simple machine. All he wants is a machine that serves coffee for 35 cents, with or without sugar and with or without cream. That’s all. He expects us to be able to put this little machine together quickly and for little cost. We decide that the machine will consist of:

• a coin slot and a coin return,
• a coin return lever, and
• four buttons: black, white, black with sugar, and white with sugar.
The Coffee Machine Problem

Simplified Description to previous slide:

a machine that serves coffee for 35 cents, with or without sugar and/or creamer. That’s all.
The Coffee Machine Problem

Design the program that runs the machine using objects. What are the components, what are their responsibilities, and how do they collaborate to deliver this simple service:

• Kim puts in a quarter and a dime and then selects a coffee.

• [Activity TIME!]
The Coffee Machine Problem

Design the program that runs the machine using objects. What are the components, what are their responsibilities, and how do they collaborate to deliver this simple service:

• Kim puts in a quarter and a dime and then selects a coffee.

• What did you come up with??
A Typical First Coffee Machine Design

Cash Box
• Knows amount of money put in.
• Gives change.
• Knows the price of coffee.
• Turns the Front Panel on and off.

Front Panel
• Captures the selection.
• Knows what to mix in each.
• Tells the Mixer what to mix.

Mixer
• Instructs the dispensers to dispense some amount of each product.

Dispensers
• Knows how to dispense a fixed amount.
• Knows when it is empty.
• (cup, coffee powder, sugar, creamer, water)
An Object Diagram for the Design

- Front Panel
  - selection
  - turn on & off
  - coins

- Mixer
  - dispense
  - dispenser is empty

- Cash Box
  - dispense

- Dispenser
  - dispense
  - dispenser is empty
A sample interaction diagram

User

Cash Box

Front Panel

Mixer

Dispenser

35¢

turn on

coffee w/sugar button

coffee w/sugar

coffee

sugar

drink
Test your design against other test scenarios:

- Kim puts in a quarter and then selects a coffee.
- Kim puts two quarters in and then selects a coffee.
- Kim puts in a quarter, then pushes the coin return lever.
- Kim puts in two quarters, walks away from the machine, and forgets to come back.
- Kim buys two coffees, white with sugar. The sugar dispenser runs out of sugar after the first.
The Coffee Machine Problem

Our class just got hired as the contractors to design a custom coffee vending machine. Arnold the person hiring us is the owner of Acme Sprockets and, like the common software designer, hates standard solutions. He wants his own custom design. He is, however, a cheapskate. Arnold tells us he wants a simple machine. All he wants is a machine that serves coffee for 35 cents, with or without sugar and with or without cream. That’s all. He expects us to be able to put this little machine together quickly and for little cost. We decide that the machine will consist of:

• a coin slot and a coin return,
• a coin return lever, and
• four buttons: black, white, black with sugar, and white with sugar.
Arnold Visits

After five machines are installed and have been operating for a while, Arnold comes along and says, "I would like to add chicken soup, at twenty-five cents. Change the machine."

We add to the machine one more button for chicken soup, and one more container for instant soup powder.
(Activity #1)

• What’s the problem with our old design for interaction between cashbox and front panel?

• How do you change your software design?
One design

Cash Box
• Knows amount of money put in.
• Gives change.
• *Answers how much money has been put in.*

Front Panel
• Captures the selection.
• *Knows the price and recipe for each selection.*
• *Asks Cash Box how much money was put in.*
• Knows what to mix in each.
• Tells the Mixer what to mix.

Mixer
• Same as before.

Dispensers
• Same as before.
One design

Front Panel

Mixer

dispense

dispenser is empty

dispense

dispenser is empty

Cash Box

Dispenser

selection

coins

ask how much $ has been entered
One Solution

User → Cash Box:
- 35¢
- coffee w/ sugar button

Cash Box → Front Panel:
- how much?

Front Panel → Mixer:
- coffee w/ sugar

Mixer → Dispenser:
- coffee

...
Arnold Visits—Again

Arnold comes back a while later with a brilliant idea. He has heard that some companies use their company badges to directly debit the cost of coffee purchases from their employees’ paychecks. Arnold’s employees already have badges, so he thinks this should be a simple change. And he hates to be behind the curve. We add to the hardware a badge reader and link to the payroll system.

How do you change your design?
One solution

Cash Box
• Knows amount of money put in.
• Gives change.
• *Accepts cash or charge.*
• *Answers whether a given amount of credit is available.*

Front Panel
• Same as before, but only *asks Cash Box if sufficient credit is available.*

Mixer
• Same as before.

Dispensers
• Same as before.
A Sample Interaction

User

Cash Box

Front Panel

Mixer

Dispenser

35¢

coffee w/sugar button

enough(35)?

yes

coffee w/sugar

coffee

sugar

drink
Evolving Designs Over Time

Change is unpredictable, but we can try to predict change.
By considering different scenarios, we are evolving toward a design that is more flexible, more evenly balanced. Most designers find it easier to grow a design than to create a finished design from scratch.
He’s Ba-a-a-a-ack...

People are starting to buy Starbucks’ lattes instead of Arnold’s coffees. So Arnold wants the machine modified just slightly, so that he can create a "drink of the week". He wants to be able to add new drinks and change prices any time, to match his competition. He wants to be able to add espresso, cappuccino, hot chocolate, latte, choco-latte, steamed milk, lemon-lime Kool-Aid—in short, anything that he can mix together in a cup with water.

We add a couple more buttons, a milk steamer and dispenser, and some more powder dispensers to the hardware configuration.

*How do we change the software design?*
Principle of Continuity

• The problem is the design violates the **Principle of Continuity**:

• A change that is small in the business sense should be small in the program. There should be a continuity between the *problem domain* and the *solution domain*.

• OOP seeks to achieve such continuity by modeling the problem domain more closely.

• We have been talking about *coffees* and *recipes*, but they don't show up anywhere in our designs.
A Candidate Design for the Coffee Machine
Sample Interaction for Final Version

User → Cash Box
- 25¢
- 25¢
- button 1

Cash Box → Product Selector
- enough? +
- yes → yes

Product Selector → Product Register
- get #1
- how much?
- yes

Product Register → aBlack Coffee
- make
- confirm quantities

aBlack Coffee → aRecipe
- make

aRecipe → Dispenser Register
- get dispensers

Dispenser Register → (individual dispensers)
- 1, 2, 3
- dispense
Our Final Design for the Coffee Machine

The class diagram is a reasonable "model" of the world.

• We have product and recipe objects. Before, they were hard-coded as "constants".

• Responsibilities are evenly distributed throughout the system.

• Component names and component responsibilities match.

With responsibilities evenly distributed, we can distribute control. Look at the decentralization in the interaction diagram!
Designs that Accommodate Change

The design allows Arnold to change configurations easily.
- To add a product, Arnold adds a new product to the product register.
- To change a price or recipe, Arnold changes a product in the register.
- We now have a dynamic object: the product. The product has behavior and thus is not a static value. It rolls through the system sharing its knowledge. Using an object of this sort localizes the state and behavior of a product into one component.
- We minimize communication among the other components by passing one object around that carries what needs to be known!
- We have no mixer. An object with no (software) responsibility dies.
Design and Change

- "Clients don't know what they want until you don't give it to them." Or until they see what you do give them. And their business needs change, too, because their clients and environment change.

  Change is unpredictable, but we can try to predict change

- By considering different scenarios, we guide the evolution of a design that is more flexible, more evenly balanced among its collaborators. Most designers find it easier to grow a design than to create a finished design from scratch.
Designs that Accommodate Change

Discussing the quality of a design is...

discussing the futures that it supports naturally