Objectives

- Design Patterns
- Midterm Review

Assignment 12 Discussion

Review

- How can we avoid writing lots of empty methods when we implement listeners?
- What is the relationship between low-level events and semantic events?
- What is a design pattern?
- What is the common theme to all our design principles/solutions?

Follow-up: Window Events

- `WindowEventDemo.java`

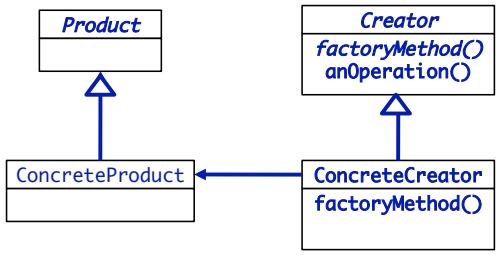
Follow Up: Compiler's Names of Classes

- Anonymous class names
  - `ClassName$#.class`
- Look inside `<workspace_dir>/ScreenSavers/bin/screensaver/nomodify`

Review: Design Pattern

- General reusable solution to a commonly occurring problem in software design
- Not a finished design that can be transformed directly into code
- Description or *template* for how to solve a problem that can be used in many different situations

Review: Factory Method Pattern



UML Diagram

Review: Dependency Inversion Principle

Depend upon abstractions.
Do not depend upon concrete classes.

- High-level components should not depend on low-level components
  - Both should depend on abstractions
- Abstractions should not depend upon details. Details should depend upon abstractions
- "Inversion" from the way you think
- Other techniques besides Factory Method for adhering to principle

Dependency Inversion Principle

Depend upon abstractions

Motivating Example

- Birds
  - Various flying behaviors (some fly, some don't)
  - Make different sounds
- How would we create classes to represent different birds?

Designing Flexible Behaviors

- Include behaviors in abstract `Bird` class
  - `FlyBehavior`, `ChirpBehavior` interface
  - `performFly()`, `makeSound()` methods
- Could have setter methods in `Bird` class to change these
  - Example: bird gets wings clipped

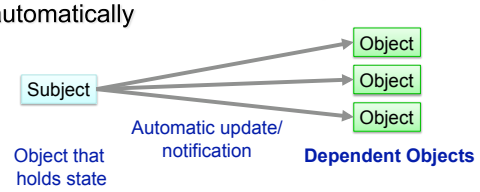
Design Pattern: Strategy

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable
- Lets algorithm/behavior vary independently from clients that use it
  - Allows behavior changes at runtime
- Design Principle:

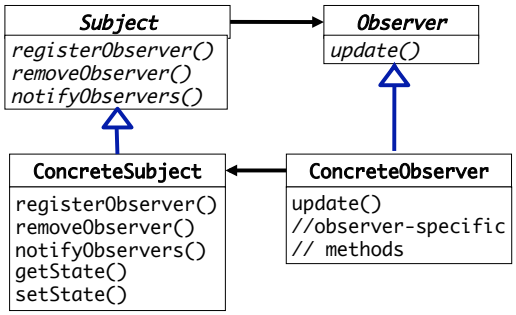
Favor ***composition*** over inheritance

Design Pattern: Observer

- Defines a 1-to-many dependency between objects
- When one object changes state, all of its dependents are notified and updated automatically



Observer Pattern

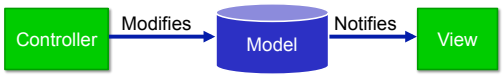


Design Principle: Loose Coupling

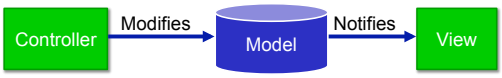
- Strive for loosely coupled designs between objects that interact
- Loosely coupled objects can interact but have very little knowledge of each other
  - Minimize dependency between objects
  - More flexible systems
  - Handle change

Model - Viewer - Controller (MVC)

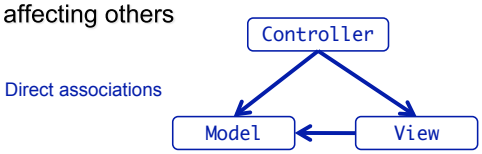
- A common **design pattern** for GUIs
- Separate
  - Model: application data
  - View: graphical representation
  - Controller: input processing



Model-Viewer-Controller



- Can have multiple viewers and controllers
- Goal: modify one component without affecting others



Model

- Code that carries out some task
- Nothing about how view presented to user
- Purely **functional**
- Must be able to register views and notify views of changes

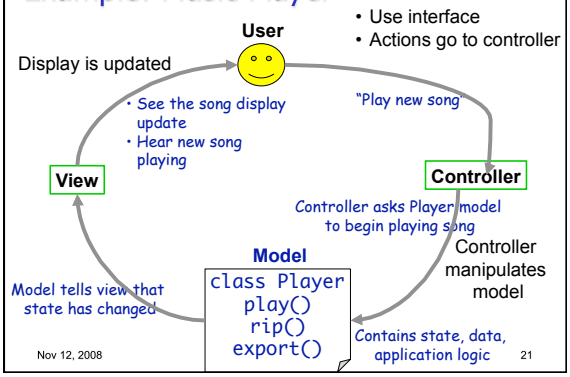
Multiple Views

- Provides GUI interface components for model
  - Look & Feel of the application
- User manipulates view
  - Informs controller of change
- Example of multiple views: spreadsheet data
  - Rows/columns in spreadsheet
  - Pie chart

Controller(s)

- Takes user input and figures out what it means to the model
  - Makes decisions about behavior of model based on UI
- Update model as user interacts with view
  - Calls model's mutator methods
- Views are associated with controllers

Example: Music Player



MVC: Combination of Design Patterns

MVC: Combination of Design Patterns

- Observer
  - Views, Controller notified of Model's state changes
- Strategy
  - View can plug in different controllers
  - View does not know how model gets updated
- Composite
  - View is a composite of GUI components
  - Top-level component learns about update, updates components

Code Analysis

- Consider GUIs we've seen
  - Which use the MVC pattern?
    - Identify M, V, and C in applicable GUIs

ASSIGNMENT 10

Excerpts from Good Critiques

- Unfortunately, because of the linear style of programming used by the original author, we can't debug this program quickly or efficiently. Debugging would basically consist of checking to make sure that each section of the code works as intended, which means we'd have to test the code basically by every 10 lines or so….

Excerpts from Good Critiques

- Added a static field "ID" to track the ID of a disk rather than wasting the extra code lines of having an extra constructor to specify the ID and forcing [others to] track the IDs of the disks it is creating…

The downside of this approach is that we can't directly specify what we want the ID of a disk to be. On the other hand, it is a much more direct and efficient way to ensure that we are always getting a unique set of IDs for a set of disks.

Excerpts from Good Critiques

- The majority of the bin-fitting process was handled inside the main method. This probably made the code easy to write, but is disadvantageous for a number of reasons:
  - Readability: …
  - Maintainability: …
  - Testing: *unit testing does not break down into small pieces to test*. There is just one big main method
  - Debugging: …

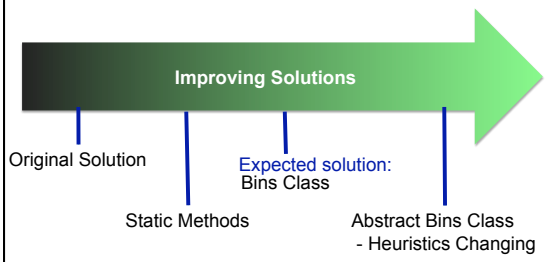
Excerpts from Good Code Critiques

- After looking back over the code and the changes I've made, I think there will almost always be more changes possible. For example, the code for the different heuristic types could be extracted to a separate class thats [sic] only job is to define the heuristics.
- Also, the Disk class could be changed to accommodate any type of storage media, not just DVDs.

Excerpts from Good Critiques

- I thought about how this program *is likely to change*. Right now we have two different methods to fit files onto disks; however, these two are certainly not the only two methods, and in the future *perhaps we will want to use other methods* in the Bins class. For this reason, I decided to make the fitFilesToDisk method abstract in the Bins class and to make a WorstFit class that inherits from the Bin class….

Reviewing Refactoring: Bins



Review: JUnit Testing Project

- Good for the most part
  - Got most of the correct implementation/"normal" behavior
  - At least some of the bad implementation/"error" behavior
  - Small/narrow test cases
  - Good naming of test cases
- Common problems
  - Not using Car's gear constants
  - Not checking erroneous input
  - Insufficient spec: Car in PARK when refueling?
  - Sometimes inappropriate expected exceptions

Midterm Prep

- Document posted online
- Software Development
  - Models
  - Testing
  - Design Principles
  - Code smells
  - Refactoring
- GUI programming
  - Event handling, inner classes, animation
- Jar files
- Unix commands

Questions About Midterm