Object Oriented Programming and Design in Java

Session 12
Instructor: Bert Huang
Announcements

- Midterm exam Wednesday, Mar. 10th
- Midterm sample problems and solutions posted on courseworks
Review

- Java Types
- Arrays, enums
- The Object Class
  - toString(), equals(), clone(), hashCode()
- Hash tables
Today's Plan

- Design tools (UML, CRC cards, etc)
- Designing classes, programming by contract
- Interfaces and polymorphism
- Programming patterns
- Inheritance and hierarchy
- Types in Java
Ideas to Programs

Analysis (common sense)

Design (object-oriented)

Implementation (actual programming)
Use Cases

• Use cases specifically describe the operation of the program
• Narrows down exactly what you want your program to do
• Useful as test cases
• Implementation and design don’t matter
Identifying Classes

- Good first step: look for **tangible nouns** in use cases. Then...
- **Agents** - objects that perform tasks
- **Events** - store information about events
- **Systems, interfaces** - run the program, talk to user or other programs
- **Foundational classes** - String, Date, etc.
Identifying Responsibilities

- Good first step: look for verbs, actions in use cases
- These actions may directly describe responsibilities, or
- may depend on other responsibilities
CRC “Cards”

- Class - Responsibility - Collaborators
- Brainstorming tool for setting up classes and responsibilities
- Collaborators loosely define class relationships; we get more precise later

<table>
<thead>
<tr>
<th>Class Name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>responsibility 1</td>
<td>Collaborator 1</td>
</tr>
<tr>
<td>responsibility 2</td>
<td>Collaborator 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Walkthroughs with CRC

- Play out (partial) use cases using CRC
- Who does what during the use case?
- Do some objects have too much responsibility?
  - Create helper objects or agents
- Are some classes never used?
UML Class Diagrams

- Each class is a rectangle

- Connect classes by their relationship
Class Relationships

- **Dependency** - any time one class needs the other
- **Aggregation** - one class contains elements of the other class
- **Association** - other relationship
- **Inheritance**
- **Interface Implementation**
Sequence Diagrams

- Draw objects as they interact over time
- UML: underline to indicate instances
- Each object has dotted life-line
- **Activation bars** indicate object running
- Arrows indicate method calls
State Diagrams

- Useful for visualizing how an object changes over time
- Rounded rectangles represent states
- Arrows and text describe triggers for state changes
Why Encapsulation?

```
MyClass
  int data
  String name
  OtherClass thing

void doSomething()
int getSomething()

/* interface methods */
```

The rest of your program...

Encapsulation
Why Encapsulation?

• Easier changes to implementation
• Control of inputs and outputs
• Less old code to have to maintain when updating
• When changes are made, easier to find what code is affected
Good Interfaces

- **Cohesion** - represent only one concept

- **Completeness** - does everything you’d expect

- **Convenience** - some syntactic sugar,
  BufferedReader(new InputStreamReader(System.in))

- **Clarity** - behavior of class should be easy to explain accurately

- **Consistency** - naming conventions, etc
Accessors vs. Mutators

- Methods to handle data members
- **Accessors** for reading
- **Mutators** for writing/modifying
- Keep them separate
Side Effects

• Avoid methods with side effects
• Calling accessors repeatedly should yield same result
  • counterexample: Scanner.nextLine()
• Mutators should change things in an obvious way
Programming by Contract

• Another formalism to help organization
• All methods and classes have “contracts” detailing responsibilities
• Contracts expressed as **preconditions, postconditions, and invariants**
Preconditions

- Condition that must be true before method is called
  - e.g., indices must be in range, objects must not be null
- Limits responsibilities of your method
Postconditions

• Conditions guaranteed to be true after method runs
  • e.g., after calling sort(), ToDoList elements are sorted by due date
• Useful when in addition to @return tags
  • I.e., usually involves mutators or side effects
Invariants

• General properties of any member of a class that are always true
  • e.g., ToDoList is always sorted
• Implementation invariants are useful when building the class
• Interface invariants are useful when using the class
Why Interfaces?

• Interchangeable parts are essential in modern engineering
• Allows tools and parts to be used for various applications
• Without establishing standard interfaces, every part must be custom-built for each application
Interfaces of Screws

Flat

Philips
**Iterator Interface**

- An Iterator\(<T>\) lets you look at one element at a time from a Collection\(<T>\)
- boolean hasNext(), T next()
- Using Iterators, you can write code that doesn't know what kind of Collection you have
Iterators Preserve Encapsulation

- Iterator user doesn't know how the items are stored
- Iterating through linked list:
  - Do work on current node
  - Go to current.next()
- Need to know linked list structure, and private next() links
Patterns are defined by a general context, the design challenge.

And a solution, which prescribes how to design your program in the context.

Since patterns are general, they will feature many interfaces.
Iterator: Context

- An aggregate object contains element objects
- Clients need access to the elements
- The aggregate should not expose its internal structure
- There may be multiple clients that need simultaneous access
Iterator: Solution

• Define an iterator class that fetches one element at a time
• Each iterator object keeps track of the position of the next element to fetch
• If there are variations of the aggregate and iterator class, implement common interface types.
Patterns in GUI Programming

- We saw in our example GUI programs that GUI code can get messy
- Thus, there are many useful patterns people have established for GUIs
Model-View-Controller

• Context: GUI displays some data that the user can affect via GUI

• Solution: separate objects into a model, a view and a controller
  • Model - stores the data
  • View - displays the data from Model
  • Controller - maps user actions to model updates
## MVC Responsibilities

<table>
<thead>
<tr>
<th>Component</th>
<th>Role</th>
</tr>
</thead>
</table>
| **Model** | Stores text and formatting markup (fonts, sizes, colors)  
Notifies View to update when Model changes |
| **View**  | Displays text with proper fonts and sizes  
Displays toolbar  
Notifies Controller when user edits text or clicks toolbar commands |
| **Controller** | Notifies model to change text when user inputs  
Notifies model to perform special commands when toolbar buttons are clicked |
<table>
<thead>
<tr>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>• A <em>subject</em> object is the source of events</td>
</tr>
<tr>
<td>• One or more <em>observer</em> objects want to know when an event occurs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Define an <em>observer</em> interface type</td>
</tr>
<tr>
<td>• The <em>subject</em> maintains collection of <em>observer</em> objects</td>
</tr>
<tr>
<td>• The <em>subject</em> provides methods for attaching <em>observers</em></td>
</tr>
<tr>
<td>• Whenever an event occurs, the subject notifies all <em>observers</em></td>
</tr>
</tbody>
</table>
Observers in MVC

- View observes Model; when Model changes, it notifies View
- Controller observes View; when user manipulates View, it notifies Controller
Pattern: Composite

- Primitive objects can be combined into composite objects
- Clients treat a composite object as a primitive object
- Define an interface type that abstracts primitive objects
- Composite object contains a collection of primitive objects
- Both primitive and composite classes implement interface
- When implementing methods from the interface, composite class applies method to its primitive objects and combines the results
Pattern: Decorator

- You want to enhance the behavior of a component class
- A decorated component can be used in the same way as a plain component
- The component class shouldn’t be responsible for the decoration
- There may be an open-ended set of possible decorations

- Define an interface type that abstracts the component
- Concrete component classes implement this interface
- Decorator classes also implement this interface
- Decorator objects manage the component that it decorates
Pattern: Strategy

• A context class benefits from different variants of an algorithm

• Clients of the context class sometimes want to supply custom versions of the algorithm

• Define an interface type, called a strategy, that abstracts the algorithm

• Each concrete strategy class implements a version of the algorithm

• The client supplies a concrete strategy object to the context class

• Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object
Inheritance

- Describes a relationship between classes in which a **subclass** is a more specific form of a **superclass**
- Declared in Java with the keyword **extends**
Subclasses

- Subclasses often provide additional methods and fields
- or they may *override* the superclass's methods
- Java allows special keyword `super` to refer to superclass
- used to invoke superclass's methods, including constructor
Liskov's Substitution Principle

• Let $q(x)$ be a property provable about objects $x$ of type $T$. Then $q(y)$ should be true for objects $y$ of type $S$ where $S$ is a subtype of $T$. (Liskov)

• You can substitute subclass objects whenever a superclass object is expected

• but not always vice versa (never)
Polymorphism and Inheritance

- Overriding methods can cause some confusion if we're unclear on how inheritance works
- We extended MouseAdapter to make MyMouseListener
  
  ```java
  MouseAdapter ma = new MyMouseListener();
  ma.mouseClicked(); // what happens?
  ```
  
- Actual types of objects, not declared types, determine which methods are called
Encapsulation and Inheritance

- Public and private modifiers apply even to subclasses
- Extending a class doesn't grant you access to its private methods
- Otherwise, implementations would not be interchangeable, since subclasses would depend on private class code
- Subclasses must implement their added functionality using only public interface of superclass
Preconditions and Postconditions

- Subclass methods cannot have stricter preconditions than superclass methods.
- Subclass methods cannot have looser postconditions than superclass methods.
- Because all subclass objects must fit Liskov substitution; they must be viewable as superclass objects.
Inheritance

- Subclasses inherit methods and fields from superclasses
- Analogous to taxonomies
- In Java and most languages, subclasses can only inherit from one superclass
Abstract Classes

• Abstract classes are meant to be extended by various subclasses
• The abstract class can never be instantiated
• but methods and fields can be defined and implemented
• A subclass can only extend one abstract class
Pattern: Template Method

- An algorithm is applicable for multiple types
- The algorithm can be broken down into *primitive operations*. The primitive operations can be different for each type
- The order of the primitive operations in the algorithm doesn't depend on the type
- Define an abstract superclass that has a method for the algorithm and abstract methods for the primitive algorithms
- Implement algorithm to call primitive operations in order
- Leave primitive operations abstract or have basic default
- Each subclass defines primitive operations but not the algorithm
Types

• Programming languages organize variables into types

• Classes are related, but don’t tell the whole story

• Types include primitives and classes

• Java is a *strongly typed* language: many compiler checks to validate type usage
Types in Java

• Types in Java are either
  • A primitive type
  • A class type
  • An interface type
  • An array type
  • The null type
Values in Java

• Values in Java are either
  • A primitive value (int, double, etc.)
  • A reference to an object of a class
  • ” ”
  • A reference to an array
  • null
Reading

- Horstmann Ch. 2.1 - 7.4
- Skim code, focus on concepts