Announcements

- Code repository, project name, and product backlog
  - Due Today 11:59pm
- Check your documents into your repository
  - /docs
- Last day to discuss product backlog is Wednesday, February 1

Lecture 08

- Design patterns
- Software design

Design patterns

- Recurring aspects of designs are called design patterns
  - A pattern in this context is a general, reusable solution to a commonly occurring problem
  - Many have been systematically documented for developers to use

“Good” patterns

- Are as general as possible
- Contain a solution that effectively solves the problem in the indicated context
- Studying patterns is one way of learning from the experience of others

What makes a design pattern

- Context: the general situation in which the pattern applies
- Problem: short description of the main difficulty tackled
- Forces: issues or concerns to consider when solving the problem
- Solution: recommended way to solve the problem
...and maybe
- Antipatterns: solutions that are inferior or do not work in the given context
- Related patterns: similar patterns
- References: who developed the pattern

Architectural patterns
- Similar to software design patterns but with broader scope

Abstraction-occurrence pattern
- Context: domain models often contain sets of related objects (occurrences)
- Members of such set share a subset of information
- Problem: finding the best way to represent such occurrences in a class diagram
- Forces: want to avoid duplicating common information

Solution
- Create an <<Abstraction>> class that contains data common to all members of the set of occurrences
- Create an <<Occurrence>> class representing the occurrences of this abstraction
- Connect the classes with a one-to-many association

General hierarchy pattern
- Context: objects may form a hierarchy with superiors and subordinates
- Some objects cannot have subordinates
- Problem: representing a hierarchy of objects with some prohibited from having subordinates
- Forces: object have many common properties and operations
**Solution**

- Create an abstract `<Node>` class containing the features possessed by all objects in the hierarchy
  - E.g., each node has a superior
- Create a `<SuperiorNode>` and link it via `<subordinates>` to the superclass
- Create `<NonSuperiorNode>`
  - Cannot be linked with `<subordinates>`

---

**Player-role pattern**

- Context: an object has a particular set of properties - a role
- Object may play different roles in different contexts
- Problem: modeling players and roles so that a player can change roles
  - Or possess multiple roles

---

**Solution**

- Forces: want to improve encapsulation
- Avoid multiple inheritance
- Instances cannot change class

- Create `<Player>` class for the object
- Create an association to an `<AbstractRole>` class
  - Subclasses encapsulate features of each different role
Singleton pattern

- **Context:** classes for which only one instances should exist
- **Problem:** ensure that it is impossible to create more than one instance
- **Forces:** public constructor cannot guarantee this
  - Singleton instance must be accessible to all classes that require it

**Solution**

- Private class variable, theInstance, stores the single instance
- Public method, getInstance(), instantiates on first invocation
  - Subsequent calls return theInstance
- Private constructor ensures no other class can create another instance
Delegation pattern

- Context: two classes, one provides the required service, the other desires it
- Problem: how to make use of a method that exists in another class
- Forces: inheritance is not appropriate
- Other methods are unneeded
- Want to minimize development cost through reuse

Solution

- Create a <<Delegator>> class with a method that calls another method in <<Delegate>>

Immutable pattern

- Context: object that contains a state that never changes after creation
- Problem: creating immutable instances of a class
- Forces: no loopholes permitting modification of object

Solution

- Ensure values for the instance variables are only set/modified in the constructor
- Accessor methods must not have side effects
Software design concepts

- What is a module?
  - "A lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier"

Module terms

Modularity

Abstraction

- High: general terms from problem environment
- Low: stated in direct implementation terms

Information hiding

- Moving the details of the software implementation to as low a level as possible
- Separates module definition from actual implementation
- Allows implementation to change without affecting calling routines

Independence

- Module or class independence can be measured using two qualitative criteria: cohesion and coupling
Cohesion

- A measure of the relative functional strength of a module
- Highly cohesive systems have increased readability and reusability
- Complexity is well managed
- In object-oriented programming, classes are cohesive if the methods are similar in many aspects

Cohesion “categories”

- Functional – best
- Communicational/Informational – almost as good
- Procedural
- Temporal
- Logical
- Coincidental – worst

Coincidental

- Difficult to describe the module’s function(s) or activities
- Module performs series of unrelated activities
- Examples
  - Initialization routines
  - Main program – might fit this category if we include the unrelated calls
  - “Utilities”

Logical

- Modules which perform logically related activities as directed by the calling program or an internal control variable
- Examples
  - Module which handles all input or output functions
  - Menu drive activities

Temporal

- Modules which perform activities related by time, must be done together
- Functions are weakly related to each other
- Examples
  - Exception handling cleanup
  - Read a series of control sensors every second

Procedural

- Performs more than one function, which are problem related and carried out in a time related sequential order
- Still weakly related
- Examples
  - Get new client, update database, print report
  - Read control variable, check if in tolerance, update status board
Communicational or informational

- Performs more than one function on the same data
- Related sequentially (time) and procedurally
- Examples
  - Read file into list, sort, print, write to file, delete
  - Build matrix, invert, solve

Functional

- Modules that perform exactly one function, a single well-defined task
- Examples
  - Compute square root
  - Print a binary tree
  - Invert a matrix

Strength attributes

<table>
<thead>
<tr>
<th>Strength</th>
<th>Independence</th>
<th>Error prone</th>
<th>Reusable</th>
<th>Extensible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Informational</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Communicational</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Procedural</td>
<td>Medium</td>
<td>Low</td>
<td>Low to</td>
<td>Medium</td>
</tr>
<tr>
<td>Temporal</td>
<td>Low</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Logical</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Coincidental</td>
<td>Low</td>
<td>Very High</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

Questions?