CS 307: Software Engineering

Lecture 6: Design Patterns
Modeling Interactions & Behavior
Modules & Cohesion

Prof. Jeff Turkstra
Announcements

- Code repository, project name, and product backlog
  - Due Tomorrow, September 8 11:58pm
- Check your documents into your repository
  - ~/docs
Lecture 06

- UML wrap up
- Design patterns
- Modeling interactions and behavior
- Modules and cohesion
System domain model vs. system model

- The system domain model omits many classes needed for a complete system
  - Could contain less than half of them
  - Developed and used independently of
    - UI classes
    - Architectural classes

- Complete system model includes
  - System domain model
  - UI Classes
  - Architectural classes
  - Utility classes
Suggested sequence

- Identify a **first set of candidate classes**
- Add **associations and attributes**
- Find **generalizations**
- List the **main responsibilities of each class**
- Decide on **specific operations**
- Iterate over the entire process until model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities, operations, etc
  - Identify interfaces
  - Apply design patterns (later)
Identifying classes

- Developing a domain model leads to discovery of classes
- When you work on the UI or architecture, one tends to invent classes to solve a design problem
- Reuse should always be kept in mind
Discovering domain classes

- Look at source material such as description of requirements
- Extract nouns and noun phrases
- Eliminate nouns that
  - ...are redundant
  - ...represent instances
  - ...are vague or highly general
  - ...not needed in the application
- Pay attention to classes in a domain model that represents types of users or other actors
Identifying associations and attributes

- Start with classes you think are most central and important
- Decide on clear and obvious data each must contain in its relationships to other classes
- Work outwards towards classes that are less important
- Avoid adding many associations and attributes to a class
  - Systems are simpler if they manipulate less information
Attributes

- Look for information about each class that must be maintained
- Nouns rejected as classes may make good attributes
- Attribute should generally contain a simple value
  - String, number, etc
Tips

- Avoid duplicates if possible
- If a subset can be formed, create a distinct class

Bad, due to a plural attribute
Bad, due to too many attributes, and the inability to add more addresses
Good solution. The type indicates whether it is a home address, business address etc.
Example
Attributes & associations
Allocating responsibilities

- A responsibility is something that the system is required to do
- Each **functional requirement** should be attributed to a class
  - All responsibilities for a given class should be clearly related
  - Too many responsibilities? Consider splitting into two or more classes
  - No responsibilities? Likely useless
  - Trouble attributing a responsibility to a class? Create a new class
Responsibilities

- Perform **use case analysis**
  - Look at user stories
- **Look for verbs and nouns describing actions in the system description**
Example

- Creating a new regular flight
- Searching for a flight
- Modifying attributes of a flight
- Creating a specific flight
- Booking a passenger
- Canceling a booking
Prototyping class diagrams

- Write names on small cards as you identify classes
- List attributes and responsibilities as they become known
  - If you cannot fit everything on one card, maybe it should be split
- Move cards around on a whiteboard to create a class diagram
- Draw lines among the cards for associations and generalizations
Identifying operations

- Operations are needed to realize the responsibilities of each class
  - May be several operations for one responsibility
  - Main operations are normally public
  - Other methods that collaborate to perform the responsibility should be private
Class diagrams in Java

- Attributes are instance variables
- Generalizations → extends
- Interfaces → implements
- Associations are normally implemented using instance variables
  - Divide two-way association into two one-way associations (each class has an instance variable)
  - Multiplicity one or optional, declare a variable of that class (a reference)
  - Multiplicity of many, use a collection class implementing List (e.g., vector)
Design patterns
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**: objects 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>>**
Aggregation

- Aggregations are special associations representing "part-whole" relationships
  - “Whole” side is often referred to as an assembly or aggregate
  - Diamond is shorthand notation for isPartOf
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 posses multiple roles
 Forces: want to improve encapsulation
 Avoid multiple inheritance
 Instances cannot change class
Solution

- 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 instance 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
```java
if (theCompany == null)
    theCompany = new Company();

return theCompany;
```
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>>
```java
public class Delegator
{
    public void delegatingMethod()
    {
        delegateMethod();
    }
}
```

```java
public class Delegate
{
    public void method()
    {
        push();
    }
}
```

```
public class Stack
{
    public void push()
    {
        list.addFirst();
    }
    public void pop()
    {
    }
    public boolean isEmpty()
    {
    }
}
```

```
public class LinkedList
{
    public void addFirst()
    {
    }
    public void addLast()
    {
    }
    public void addAfter()
    {
    }
    public void removeFirst()
    {
    }
    public void removeLast()
    {
    }
    public void delete()
    {
    }
    public boolean isEmpty()
    {
    }
}
```
```
flightNumber()
{
    return specificFlight.flightNumber();
}
```

```
flightNumber()
{
    return regularFlight.flightNumber();
}
```
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
Modeling interactions and behavior
Dynamic (behavioral) diagrams

- Model the dynamic aspects of a software system
  - Sequence diagrams
  - State machine diagrams
  - Activity diagrams
Interactions and messages

- Sometimes built from a use case (user story) and a class diagram
  - Show how a set of objects accomplish the required interactions with an actor
- Steps taken together is an interaction
- Can show several types of communication
  - Method calls, network messages, etc
  - All referred to as messages
Diagram elements

- Class instances
  - Boxes with class and object identifier underlined

- Actors
  - Stick-person symbol (use case diagrams)

- Messages
  - Arrows from actor to object or object to object
Creating

- Develop a class diagram and use case model before creating an interaction diagram
Sequence diagrams

Course
  getPrerequisite()

CourseSection
  requestToRegister()
  addToRegistrationList()

Registration
  create
  addToSchedule()

Student
  addToSchedule()
  hasPassedCourse()
Sequence diagrams

- Shows the sequence of messages exchanged by a set of objects performing a specific task
  - Objects are arranged horizontally
  - Actor is often on the left
  - Vertical dimension represents time
  - Vertical line, lifeline, attached to each object or actor
  - Messages are arrows between sender and receiver
    - Can have argument list and return value
Deletion
State diagrams

- Describe the behavior of a system, part of a system, or an individual object
- Directed graph
  - Nodes = states
  - Arcs = transitions
- System or object is always in some state
  - Will behave in a specific way in response to an event or input
- Events cause the system to change state
  - Each state behaves a certain way
Tic-tac-toe
States

- System is always in a single state
- Event causes state to transition
- States are rounded rectangles
  - Contain the name
- Special states
  - Black circle = start state
  - Circle with a ring around it = end state
    - Can be multiple end states
Transitions

- Change of state in response to an event
- Label on each transition is the event
Time-outs and conditions

(a) GreenLight
   after(25s)
   YellowLight
   after(30s)
   after(5s)
   RedLight

(b) GreenLightNoTrigger
   vehicleWaitingToTurn
   GreenLightChangeTriggered
   after(25s since exit from state RedLight)
   YellowLight
   after(5s)
   RedLight
Actions

- **Closed**
  - enter/stop motor
  - closingCompleted

- **Opening**
  - enter/run motor forwards
  - openingCompleted

- **Closing**
  - enter/run motor in reverse
  - pressButton

- **Open**
  - enter/stop motor
  - pressButton
A diagram illustrating a state transition model:

- **Recording**
  - exit/stop

- **Rewinding**
  - endOfTape
  - startOfTape/stop

- **Wait**
  - endOfProgram

The diagram shows the flow from Recording to Rewinding to Wait, with transitions marked by the events mentioned.
Activity diagrams

- Similar to a state diagram except most transitions caused by internal events (e.g., completion of computation)
- Help understand object or component work flow
- Visualize interrelation and interaction between different use cases
- Most often associated with several classes
- Can represent concurrency
Concurrency

- **Fork** – one incoming transition, multiple outgoing transitions

- **Join** – multiple incoming transitions, one outgoing transition
  - All incoming transitions must occur before proceeding

- **Rendezvous** – multiple incoming and multiple outgoing transitions
  - All incoming transitions must occur first
Swimlanes

- Partition activities among classes
Software design concepts

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

![Graph showing the relationship between total cost, cost per module, and number of modules, with a minimum cost region highlighted.](image-url)
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
Questions?