Lecture 16: Program design

Lecture goals

- Distinguish between heavyweight and lightweight design processes
- Document static and dynamic designs using UML diagrams
- Leverage design patterns to reuse solutions to common problems

Program design models

Program design

- Goal: represent software architecture in form that can be implemented as one or more executable programs
- Specifies:
  - Programs, components, packages, classes, class hierarchies
  - Interfaces, protocols
  - Algorithms, data structures, security mechanisms, operational procedures
- Historically (e.g. aerospace), program design done by domain engineers, implementation done by programmers

Heavyweight design

- Program design and coding are separate
  - Use models to specify program in detail, before beginning to code
  - UML provides modeling notation

Lightweight design

- Program design and coding are interwoven
  - Development is iterative
  - Assisted by integrating multiple development tools (IDEs)
- Fine line between “lightweight” and “sloppy”

Mixed approach

- Use models to specify outline design
- Work out details iteratively during coding

UML models for design

- Diagrams give general overview
  - Principal elements
  - Relationships between elements
- Specifications provide details about each element
  In a heavyweight process, specifications should have sufficient detail so that corresponding code can be written unambiguously. Ideally, specification is complete before coding begins.

UML model choices

- Requirements
  - Use case diagram: use cases, actors, and relationships
- Architecture
Component diagram: interfaces and dependencies between components
Deployment diagram: configuration of processing nodes and the components that execute on them

Program design
- Class diagram (structural): classes, interfaces, collaborations, and relationships
- Sequence diagram (dynamic): set of objects and their relationships

Structural (static) modeling

Class diagram
- Class: Set of objects with the same attributes, operations, relationships, and semantics
- "Operation" in UML = "method" in Java

<table>
<thead>
<tr>
<th>Window</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin size</td>
<td>attributes [local, instance, and class (static) variables]</td>
</tr>
<tr>
<td>open() close() move() display()</td>
<td>methods</td>
</tr>
<tr>
<td></td>
<td>responsibilities [optional text]</td>
</tr>
</tbody>
</table>

Example: Hello World applet

```java
import java.applet.Applet;
import java.awt.Graphics;

class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello!", 10, 20);
    }
}
```

**class**

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>paint()</td>
</tr>
</tbody>
</table>
Annotations

- **class**
  - **name**: HelloWorld
  - **methods**: paint()

**optional annotation**
g.drawString("Hello!", 10, 20)

Relationships

- **Association**: show multiplicity of links between instances of classes
  - Analogous to relations in entity-relation diagrams
  - Bidirectional – doesn’t imply ownership or composition
  - Solid line with multiplicity at each end, optional label
  - See Sommerville, Figure 5.9
- **Dependency**
  - A change to one class may affect the semantics of another
  - Dashed arrow with stick head, pointing to the dependency
- **Generalization (inheritance)**
  - Objects of a specialized (child) class are substitutable for objects of a generalized (parent) class
  - Solid arrow with enclosed head pointing from child to parent
- **Realization (interfaces)**
  - A class is guaranteed to fulfil a contract specified by another class
  - Dashed arrow with enclosed head
- **Aggregation**
  - An instance of one class (the whole) is composed of objects of other classes (the parts)
  - To reduce coupling, prefer composition over inheritance
  - See Sommerville, Figure 5.13
Example: Hello World applet relationships

```
«interface»
MenuContainer

Applet

HelloWorld

part()

Graphics
```

Tools for UML-based design
- Rational Rose (and derivatives)

Lightweight design
- Less detail
  - Only show "interesting" behaviors and attributes with ownership significance
- Less permanent
  - May only exist on whiteboard during design brainstorming
  - Reduces maintenance of keeping documents in-sync with code
- Less sequential
  - Only design what you need for current task
  - Use lessons from implementation to iterate on designs
- Leverage tooling and modern languages
  - Generate diagrams from source code
  - Generate specifications from comments
  - IDEs highlight attributes and methods

Class design
Given a real-life system, how do you decide which classes to use?
- Given a real-life system, how do you decide which classes to use?
Step 1: Identify set of candidate classes
  - What terms do users and implementers use to describe the system?
  - Is each candidate class crisply defined?
  - What are the candidate classes' responsibilities? Are they balanced?
  - What attributes and methods does each class need to carry out its responsibilities?

Step 2: Refine list of classes
  - Improve clarity of design
  - Increase coherence within classes, reduce coupling between classes

Application and solution classes
  - Application classes represent application concepts.
    - Use Noun Identification to generate candidate application classes
  - Solution classes represent system concepts
    - User interface objects, databases, etc.

Example
Noun identification

*The library contains books and journals. It may have several copies of a given book. Some of the books are reserved for short-term loans only. All others may be borrowed by any library member for three weeks.*

*Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.*

*The system must keep track of when books and journals are borrowed and returned, and enforce the rules.*
Candidate classes

<table>
<thead>
<tr>
<th>Noun</th>
<th>Comments</th>
<th>Candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library</td>
<td>the name of the system</td>
<td>no</td>
</tr>
<tr>
<td>Book</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>Journal</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>Copy</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>ShortTermLoan</td>
<td>event</td>
<td>no (?)</td>
</tr>
<tr>
<td>LibraryMember</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>Week</td>
<td>measure</td>
<td>no</td>
</tr>
<tr>
<td>MemberOfLibrary</td>
<td>repeat of LibraryMember</td>
<td>no</td>
</tr>
<tr>
<td>Item</td>
<td>book or journal</td>
<td>yes (?)</td>
</tr>
<tr>
<td>Time</td>
<td>abstract term</td>
<td>no</td>
</tr>
<tr>
<td>MemberOfStaff</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>System</td>
<td>general term</td>
<td>no</td>
</tr>
<tr>
<td>Rule</td>
<td>general term</td>
<td>no</td>
</tr>
</tbody>
</table>

Candidate relations

- Book is an Item
- Journal is an Item
- Copy is a copy of a Book
- LibraryMember is a copy of a Book
- Item
- MemberOfStaff is a LibraryMember

Candidate methods

- LibraryMember borrows Copy
- LibraryMember returns Copy
- MemberOfStaff borrows Journal
- MemberOfStaff returns Journal
Candidate class diagram

Moving towards final design
- Reuse: Wherever possible use existing components, or class libraries
  - They may need extensions.
- Restructuring: Change the design to improve understandability, maintainability
  - Merge similar classes, split complex classes
- Optimization: Ensure that the system meets anticipated performance requirements
  - Change algorithms, more restructuring
- Completion: Fill all gaps, specify interfaces, etc.
- Design is iterative
  - As the process moves from preliminary design to specification, implementation, and testing it is common to find weaknesses in the program design. Be prepared to make major modifications.

Class design advice
- Classes should be easy to use correctly and hard to use incorrectly
  - See Effective C++, Third Edition
- Avoid cyclic dependencies (tight coupling)
  - While allowed, can lead to awkwardness in build procedure, limit portability

Dynamic modeling
- Interaction diagrams: show a set of objects and their relationships
  - Includes messages sent between objects
- Sequence diagrams: time ordering of messages
Object notation

**Classes**

- **AnyClass**
  - attribute1
  - attribute2
  - method1()
  - method2()

**Objects**

- anObject:AnyClass
- :AnyClass
- anObject

The names of objects are underlined.

Message notation

**call**

- returnCopy(c)
- okToBorrow() \(\text{local}\)

**return**

- status

**send**

- notifyReturn(b) \(\text{asynchronous signal}\)

**create object**

- <<create>>

**destroy object**

- <<destroy>> \(\text{stereotypes}\)
Design patterns

Reusable design patterns

- Design templates that solve recurring problems in a variety of different systems
- Popularized by "Gang of Four"
  - E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994
- Avoid reinventing the wheel; adopt proven solutions with known tradeoffs
- When developers are familiar with design patterns, they can be used to quickly communicate complex relationships between classes

Properties of patterns

- Meaningful name
- Description of the problem setting
  - Explains where pattern may be applied
- Description of solution
  - Not a library, but a "design template"; can be instantiated in different ways
  - Often expressed graphically
- Statement of consequences
  - Results and tradeoffs of applying the pattern in the problem setting

Implementation

- Design patterns make extensive use of inheritance and abstract classes/interfaces
  - Classes that provide concrete implementations for abstract methods can participate in the pattern
Observer pattern

- **Setting:** A variety of entities (for example, different graphical views) need to be updated whenever the state of an object changes
- **Solution**
  - Observers: notified when Subject state changes; should update (i.e. display) accordingly
  - Subject: notifies Observers when its state changes
- **Consequences**
  - Subject not coupled to concrete Observers
  - Lack of coupling may impede performance optimizations
  - Redundant updates may be triggered
  - Control flow for Observers is inverted, can be hard to trace

See Sommerville, Figure 7.12

**Examples**

- Swing JButton (subject) and ActionListener (observer)