Announcements

- Design Document
  - Due Friday, September 15 11:58pm
- Thursday sprint reviews
Lecture 07

- Coupling
- Design-related quality
- Software design
- Design principles
- Software architecture
- Architectural patterns
- Design document
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
## Strength attributes

<table>
<thead>
<tr>
<th>Strength</th>
<th>Independence</th>
<th>Error prone</th>
<th>Reusable</th>
<th>Extendable</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 Medium</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>
Coupling

- A measure of the relative interdependence among modules
- Changes in one place may require changes somewhere else
- Difficult to see how components work
Coupling categories

- Data – best
- Stamp
- Control
- External
- Common
- Content – worst
Content coupling

Two modules are content coupled if one directly references the contents of the other

- Encapsulation is broken

Examples

- Module A branches into the local space of module B
- Module A uses data within the local space of module B
Common coupling

- Two components have and use write access privileges to the same **global** data
  - Written by only one routine and read by one or more routines is **not** common coupling

- Singleton pattern provides encapsulated global access to an object
External coupling

- Modules use or pass data and/or control signals to external systems or devices
  - OS dependencies, shared libraries, hardware

- Examples
  - System calls
  - Mac tool box commands
  - Direct I/O routines
Control coupling

- Two components are control coupled if one passes an element of control to the other component
  - Calling module must explicitly control the logic of the called module
- Example: integer “signal” passed to C switch statement. Module A must know internal structure of module B
Stamp coupling

- Components pass data structures (classes) as parameters
  - Not all fields required by called module
- Examples
  - Linked lists
Data coupling

- Two components are data coupled if all parameters (data items) are used by the called routine
  - No data items best; but, usually not possible
- Trade-off between data coupling and stamp coupling
  - Increasing one often decreases the other
### Coupling attributes

<table>
<thead>
<tr>
<th>Coupling</th>
<th>Independence</th>
<th>Error prone</th>
<th>Reusable</th>
<th>Extendable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Stamp</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Control</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>External</td>
<td>Low to Medium</td>
<td>High</td>
<td>Low to Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Common</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Content</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>
Design specification quality

- **Understandable** – by all concerned parties
- **Unambiguous** – one interpretation for each requirement
- **Complete** – nothing overlooked
- **Verifiable** – compliance can be checked. Will know when done
- **Consistent** – no conflicting requirements
- **Modifiable** – change control is in place, and we can change it
- **Traceable** – we can find things easily
Developer quality issues

- Testable – we can test it
- Maintainable – we can repair it or port it
- Enhanceable – we can easily enhance its value
- Correct – it has zero known defects
- Robust – fail-safe in the users operational environment
- **Reliable** – long MTBF and short MTTR. Quick recovery time
- **Installable** – easy to install in user’s environment
- **Liability** – safety and security risks are acceptable
- **Manufacturable** – we can reproduce it
- **Marketable** – we can sell it
User quality issues

- **Affordable** – they are willing to pay for it
- **Valuable** – something they want and/or need
- **Correct** – satisfies expectations, zero defects as used
- **Usable** – easy to use in user’s environment with a low work load
Learnable – effort is worth the time and cost. Good documentation and online help is available

Robust – tolerant of user errors

Safety – causes no harm

Security – protects user’s property rights

Serviceable – quality help available and affordable

Tailorable – can adapt to user’s needs
What is design?

- “...the process of applying various techniques and principles for the purpose of defining a process or a system in sufficient detail to permit its physical realization” – E.S. Taylor
Designs

- **Describe how to implement requirements given constraints imposed by**
  - Quality
  - Platform
  - Process
  - Budget
  - etc
Design issues

- Sub-problems of the overall design
- Often have several alternative solutions (options)
- Design decision chooses among these options
  - The “best” option
  - Trade-off analysis
Making decisions

- Leverage knowledge of...
  - Requirements
  - Design so far
  - Available technology
  - Design principles and “best practices”
  - Past experience
Design space

- Set of possible designs that solve a given problem

![Diagram showing different design options and programming languages](image-url)
Component

- Any piece of software or hardware that has a clear **role**
- May be **isolated**
  - Can swap with another component providing equivalent functionality
- Often designed to be **reusable**
Module

- **Type of component**
- *E.g., methods, classes, and packages in Java*
System

- A logical entity, having a set of definable responsibilities or objectives
  - Consists of hardware, software, or both
- Specification implemented by a collection of components
- Continues to exist even if components are changed or replaced
Subsystem

- System that is part of a larger system
  - Usually with a defined interface
Design

- **Top-down**
  - Start with high level structure
  - Gradually work down to detailed decisions and low-level constructs

- **Bottom-up**
  - Make decisions about reusable low-level components
  - Decide how to put them together to create high-level constructs
Why not both?

- Mix of top-down and bottom-up approaches are normally used
- “Top-down design, bottom-up implementation”
Aspects of design

- Architecture
  - Division into subsystems and components
  - Their connections
  - Interactions
  - Interfaces

- Class design
  - Various features of classes

- User interface design

- Algorithm design
Good Design

- Reduces cost and increases quality
- Conforms to requirements
- Accelerates (helps) development
- Satisfies qualities e.g.,
  - Usability
  - Efficiency
  - Reliability
  - Maintainability
  - Reusability
Design principles

- Divide and conquer
- Increase cohesion
- Reduce coupling
- Maximize abstraction
- Increase reusability
- Reuse other designs and code
- Design for flexibility
Divide and conquer

- It is easier to deal with a series of smaller things than something big all at once
  - Separate people can work on each part
  - Individual software engineers can specialize
  - Individual components are smaller and easier to understand
  - Components can be replaced or modified without impacting other system parts
Dividing

- Distributed system – clients and servers
- Subsystems
- Packages
- Classes
- Methods
Increase cohesion

- Keep things together that are related
- Keep unrelated things out
  - System as a whole is easier to understand
  - Easier to change
- Types of cohesion: functional, communicational or informational, procedural, temporal, logical, coincidental
Reduce coupling

- Coupling occurs when modules have interdependence
  - Changes in one place require changes elsewhere
  - Harder to see how a component works
- Types of coupling: data, stamp, control, external, common, content
Maximize abstraction

- Ensure your designs allow you to hide or defer consideration of details
  - Reduces complexity
- Good abstraction → information hiding
  - Permits one to understand the essence of a subsystem without knowing its details
Increase reusability

- Design components so they can be used again in other contexts
- Generalize
- Simplify
Reuse

- Complementary to design for reusability
- Reusing designs or code allows you to take advantage of the investment you and others have made in reusable components
Flexibility

- Actively anticipate changes that a design may undergo in the future, and prepare for them
  - Reduce coupling, increase cohesion
  - Create abstractions
  - Do not hard-code anything
  - Use reusable code and make code reusable
Design principles cont.

- Anticipate obsolescence
- Portability
- Testability
- Defensive design
Anticipate obsolescence

- Plan for changes in technology and environment so the software can continue to run
  - Avoid using early releases of technology
  - Avoid software libraries that are specific to an environment
  - Avoid undocumented “features”
Avoid software and hardware from companies that are less likely to provide long-term support

Use standard languages and technologies that are supported by multiple vendors
Portability

- Make sure that the software can run on as many platforms as needed
  - Avoid using facilities that are specific to a particular environment
  - E.g., a library only available in Microsoft Windows
Testability

- Take steps to make testing easier
  - Design a program to automatically test the software
    - More later
  - Ensure all functionality can be driven by an external program, bypassing the GUI
- In Java, can create a main() method in each class to exercise other methods
Defensive design

- Never trust how others will use a component that you are designing
  - Handle all cases where other code might attempt to use your component inappropriately
  - Check and validate all inputs to your component
Design by contract

- Defensive design in a systematic way
- Each method has a contract with callers that asserts:
  - What preconditions are true on entry
  - What postconditions are true on exit
  - What invariants exist during execution
Making good design decisions

- List and describe alternatives for a design decision
- List advantages and disadvantages of each
  - Consider your objectives and priorities
- Choose the alternative that best meets your objectives
Software Architecture

- Set of structures that can be used to reason about a system
- Comprises software elements, relations among them, and properties of both

* some slides based on material developed by CS student Joe Koncel
Architectural decisions

- Choices about fundamental system structure – core of design
  - Determines overall efficiency, reusability, and maintainability of system
- Expensive to change once implemented
Importance

- Enables everyone to better understand the system
- Allows people to work on individual pieces in isolation
- Prepares for extension of the system
- Facilitates reuse and reusability
Good architectural models

- Contain a logical breakdown of subsystems
  - Separation of concerns
- Document interfaces between subsystems
- Capture dynamics of component interactions
- Outline shared data
- Are quality-driven
Stability

- Architectural models should be stable
  - Ensure maintainability and reliability
- Stable means features and components can be added or changed without impacting the overall architecture
Developing an architectural model

- Start by sketching an outline of the architecture
  - Based on principal requirements and use cases
- Determine the main components
- Apply architectural patterns when appropriate
Refine the architecture

- Decide how data and functionality will be distributed among the components
- Identify main ways components interact and their interfaces
- Decide if a framework exists that can be re-used
- Consider each use case and adjust the architecture to make it realizable
Architectural patterns

- Like software design patterns, there are software architecture patterns
  - Called architectural patterns or styles
- Each pattern has...
  - A context
  - A problem
  - A solution
Multi-Layer pattern

- Problem – system components need to be built and tested independently
- Solution – define layers (groupings of cohesive modules) and a unidirectional allowed-to-use relation among the layers
  - Often illustrated with stacked boxes representing layers on top of each other
- Separate layer for UI
- Layers below UI provide application functions
  - Determined by use cases
- Bottom layers provide general services
  - Network communication
  - Database access
  - etc
Example

(a) Typical layers in an application program

(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system
What about those design principles?

- Divide and conquer
- Increase cohesion
- Reduce coupling
- Maximize abstraction
- Increase reusability
- Reuse
- Flexibility
- Anticipate obsolescence
- Portability
- Testability
- Defensive design
Client-server architecture

- **Problem** – large number of distributed clients need access to shared resources or services

- **Solution** – client components initiate interactions with server components, invoking services as needed and waiting on results
Example

Client1:  
exchange messages

exchange messages

Client2:  

look up addresses

Server:  

Design principles?

- Divide and conquer
- Increase cohesion
- Reduce coupling
- Maximize abstraction
- Increase reusability
- Reuse
- Flexibility
- Anticipate obsolescence
- Portability
- Testability
- Defensive design
Transaction-processing pattern

- **Problem** – system must read and handle series of inputs that change stored data

- **Solution** – dispatcher component that decides how to handle each transaction (input), calling a procedure or messaging a component
Design principles?

- Divide and conquer
- Increase cohesion
- Reduce coupling
- Maximize abstraction
- Increase reusability
- Reuse
- Flexibility
- Anticipate obsolescence
- Portability
- Testability
- Defensive design
Pipe-and-filter

- Stream of data is passed through a series of processes
  - Each transforms it in some way
  - Data is constantly fed into the pipeline
  - Processes work concurrently
- Architecture is flexible
  - Almost all components could be removed
  - Components are easily replaced
  - New components easily added
  - Easy to reorder
Design principles?

- Divide and conquer
- Increase cohesion
- Reduce coupling
- Maximize abstraction
- Increase reusability
- Reuse
- Flexibility
- Anticipate obsolescence
- Portability
- Testability
- Defensive design
Model-View-Controller (MVC)

- **Problem** – UI needs frequent modification without impacting system’s functionality

- **Solution** – break system into three components – model, view, and controller
  - controller mediates between the model and the view
MVC

- Model contains underlying classes
  - Instances are viewed and manipulated
- View contains objects that render the appearance (UI) of data from the model
- Controller contains objects that control and handle user’s interaction with the view and the model
viewed by actor

View

create and update

notify about changes

Model

Controller

receives actor events

modify
MVC on the WWW

- View component generates HTML
  - Displayed by browser
- Controller interprets HTTP POSTs from the browser
- Model is the underlying system
  - Manages the information
Design principles?

- Divide and conquer
- Increase cohesion
- Reduce coupling
- Maximize abstraction
- Increase reusability
- Reuse
- Flexibility
- Anticipate obsolescence
- Portability
- Testability
- Defensive design
Service-oriented

- **Problem** – service consumers must be able to use/access a number of service providers
  - Without understanding the implementation
- **Solution** – cooperating peers that request service from and provide services to one another across a network
  - Called **web services** on the Internet
Design principles?

- Divide and conquer
- Increase cohesion
- Reduce coupling
- Maximize abstraction
- Increase reusability
- Reuse
- Flexibility
- Anticipate obsolescence
- Portability
- Testability
- Defensive design
Good design documents

- Aid in making better designs
  - Force you to be explicit
  - Consider important issues **before** implementation
  - Allow people to review the design and improve it

- Are a means of communication
  - Between those **implementing** the design
  - Those who need to **modify** the design
  - Those who need to **interface** with the system
Our design document

- **Purpose** – briefly explain the system you are designing

- **Design outline**
  - Outline your design decisions (e.g., client-server model)
  - Identify system components
  - Describe their purpose
  - Describe interactions
  - Include at least one UML diagram showing high-level system structure
Design issues

- Spend a lot of time thinking about design issues
  - One or two is not sufficient for full credit
- Each design issue should have:
  - Descriptive title
  - Potential solutions
  - Justification for your choice
- May be divided into two subsections
  - Functional Issues and Non-Functional Issues
Design details

- Class-level design of the system
  - Read: class diagrams
  - Be as detailed as practical
- Describe classes and interactions between the classes
- Use sequence diagrams to show system activities
- Include activity (or state diagrams)
- Include UI mockups
Avoid

- Documenting information that is readily obvious to a skilled programmer or designer
- Writing details that would make better code comments
- Writing details that can be extracted automatically from code
  - E.g., list of public methods
Questions?