CS 315 - October 5, 2015

From Modules to Objects

- Overview
  - What is a module?
  - Measuring Software
    - Cohesion
    - Coupling
  - Data Encapsulation
  - Abstract Data types
  - Information hiding
  - Objects
  - Inheritance, polymorphism, and dynamic binding
  - The object-oriented paradigm
- What is a module?
  - A lexically contiguous sequence of program statements, bounded by boundary elements, with an aggregate identifier
    - Lexically Contiguous
      - Adjoining in Code
    - Boundary Elements
      - `{ ... }`
      - A class
      - An object
      - A method
      - A package
    - Aggregate Identifier
      - A name for the entire module
- Design
  - Seek
    - Maximal interaction **within** a module (cohesion)
    - Minimal interaction **between** modules (coupling)
  - Module Cohesion
    - Degree of interaction within a module
  - Module Coupling

        - Degree of interaction between modules

- Cohesion
    - The degree of interaction within a module
    - Cohesion implies that a class encapsulates only attributes and operations that are closely related to each other and to the class itself
    - *Single-Mindedness* of a module
    - Placed on a scale from 7 down to 1 (good to bad)
        - Informational
        - Functional
        - Communicational
        - Procedural
        - Temporal
        - Logical
        - Coincidental

    - Coincidental Cohesion
        - A module has coincidental cohesion if it performs multiple completely unrelated actions
        - Parts of a module are grouped arbitrarily. The only relationship between the parts is that they have been grouped together
        - Typical for a utility class
        - Such modules arise from rules like "Every module will consist of between 35 and 50 statements"
        - Why is this bad?
            - It degrades maintainability
            - A module with coincidental cohesion is not reusable
            - The problem is easy to fix
                - Break the module into separate modules, each performing one task

    - Logical Cohesion
        - A module has logical cohesion when it performs a series of related actions, one of which is selected by the calling module
        - Why is this bad?
            - The interface is difficult to understand
            - Code for more than one action may be intertwined
            - Difficult to reuse

    - Temporal Cohesion
        - A module has temporal cohesion when it performs a series of actions related in time
        - Parts of a module are grouped by when they are processed, at a particular time in program execution
        - A function called after catching an exception which closes open files, creates an error

log, and notifies the user
    - Why is this bad?
        - The actions of this module are weakly related to one another, but strongly related to actions in other modules
        - Actions are only linked because they they take place at the same time
        - Not reusable

- Procedural Cohesion
    - A module has procedural cohesion if it performs a series of actions related by the procedure to be followed by the product
        - Why is this bad?
            - The actions are still weakly connected, so the module is not reusable

- Communicational Cohesion
    - A module has communicational cohesion if it performs a series of actions related by the procedure to be followed by the product, but in addition all the actions operate on the same data
    - There is still a lack of reusability

- Functional Cohesion
    - A module with functional cohesion performs exactly one action
    - Benefits
        - Corrective maintenance is easier
            - Fault isolation
            - Fewer regression faults

        - Promotes reuse because the methods are more versatile
        - Easier to extend a product

- Informational Cohesion
    - A module has informational cohesion if it performs a number of actions, each with its **own entry point**, with **independent code** for each action, all performed on the **same data structure**

- Coupling
    - The degree of interaction between modules
        - Five categories or level of coupling (non linear scale)
        - Data coupling (good)
        - Stamp coupling
        - Control coupling
        - Common coupling
        - Content coupling (bad)

    - Content Coupling

        - If one module reference contents of another
        - P modifies q
        - P refers to local data of q
        - P branches into q
        - Why is this bad?
            - Almost any changes to module q, even recompiling with a new compiler or assembler, requires a change to module p
    - Common Coupling
        - If two modules both have write access to global data
        - The ability to read **and change** is important
            - Global constants are okay
        - Why is this bad?
            - It contradicts the spirit of structured programming
                - The resulting code is virtually unreadable
            - Modules have side affects
            - A change during maintenance to the declaration of a global variable in one module necessitates corresponding changes in other modules
            - Common-coupled modules are difficult to reuse
            - A module is exposed to more data than necessary
                - This can lead to computer crime
                - "Does this code have access to this data?"
    - Control Coupling
        - If one module passes an element of control to another
        - An operation code is passed to another module with logical cohesion
        - A control switch passed as an argument
    - Stamp Coupling
        - If a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure
        - Why is this bad?
            - It is not clear which fields have been changed
            - Difficult to understand
            - Unlikely to be reusable
            - More data than necessary is passed
                - Uncontrolled data access can lead to computer crime
                    - You don't need access to a Social Security number to change someone's name
        - Stamp coupling is not bad if the whole data structure is used

    - Data Coupling
        - If all parameters are homogenous data items (simple parameters, or data structures all of whose elements are used by call module)
        - The difficulties of other coupling types are not present
        - Maintenance is easier
        - The Importance of Managing Coupling
            - Changes to one module can require changes to another
            - Good design has high cohesion and low coupling

- Key Definitions
    - Abstract Data Type
    - Abstraction
    - Class
    - Cohesion
    - Coupling
    - Data Encapsulation
    - Information Hiding
    - Object

- Information Hiding
    - Data abstraction
        - The designer thinks at the level of an Abstract Data Type

    - Procedural Abstraction
        - Define a procedure - extend the language by providing new functionality

    - Both are instances of a more general design concept, *information hiding*
        - Design the modules in a way that items likely to change are hidden
        - Future change is localized
        - Changes cannot affect other modules

- Objects
    - First Refinement
        - The product is designed in terms of abstract data types
        - Variables ("objects") are instantiations of abstract data types

    - Second Refinement
        - Class: an abstract data type that supports *inheritance*
        - Objects are instantiations of classes

- Inheritance
    - An object of a class has *attributes*
        - Values are assigned to describe the object

    - A subclass has all the attributes of the parent/super class, plus its own attributes
    - Inheritance is one of the essential features for all object-oriented languages
        - Other two are Polymorphism and Data Encapsulation
    - Not present in classical languages
        - Such as C, COBOL, and FORTRAN
    - Represented by a large open arrow in UML
- Aggregation
    - UML notation is an open diamond
- Association
    - UML notation is a line (optional navigational triangle to indicate flow)
- Inheritance, Polymorphism, and Dynamic Binding
    - Polymorphism and Dynamic Binding
        - Can have a negative impact on maintenance
            - The code is hard to understand if there are multiple possibilities for a specific method
        - A strength and weakness of the object-oriented paradigm
- The Object-Oriented Paradigm
    - Reasons for Success
        - The object-oriented paradigm gives overall equal attention to data and operations
            - At any one time, data or operations may be favored
        - A well-designed object (high-cohesion, low coupling) models all the aspects of one physical entity
        - Implementation details are hidden
    - Weaknesses
        - Development effort and size can be large
        - One's first object oriented project can be larger than expected
            - Even taking the learning curve into account
            - Especially if there is a GUI
        - However, some classes can frequently be reused in the next project
            - Especially if there is a GUI
        - Inheritance can cause problems
            - The fragile base class problem
            - To reduce the ripple effect, all classes need to be carefully designed up front

        - Unless explicitly prevented, a subclass inherits all it's parents attributes
            - Objects lower in the tree can become large
            - "Use inheritance where appropriate"
            - Exclude unneeded inherited attributes
            - The use of polymorphism and dynamic binding can lead to problems
            - It is easy to write bad code in any language
                - It is especially easy to write bad object-oriented code

Design Patterns are generalized solutions to specific problems.

Design Patterns Have a:

- Name
    - Each pattern has a name so it can be easily recognized.
    - Gives us a vocabulary we can use to discuss the design.
- Problem
    - Desgned to address a specific problem
- Solution
    - Each pattern prvoides a solution to a problem
    - Components to that solution are known as participants

Adapter Pattern solves implmentation compatibilities

- Provides a general solution to the problem of permitting communication between to objects with imcompatible interfaces.
- Provides all the advantages of information hiding without having to actually hide the implementation details.

Composite Design Pattern puts objects into tree structures to represent the whole/part hierarchies.

- Component
    - Declares the interface for objects in the composition.
    - Implements the default behavior, as appropriate
    - Declares interfaces for accessing and managing child components.
- Leaf
    - Represents primitive: no children
- Composite
    - Defines behavior for components having children
    - Stores child components
    - Implements child related operations of the component interface.

- Client
  - Manipulates objects in the compositions through the component interface.

Trees, LinkedLists are all examples of the composite design pattern.

Bridge Design Pattern is used to decouple the abstraction from the implementation so that the two can be changed independently of one another.

- Sometimes called a driver.
- Suppose a part of a design is hardware dependent, but the rest is not, then the design consists of two pieces:
  - The hardware dependent parts on one side.
  - The hardware independent parts on the other side.

- Allows for multiple implementations with the same interface.
- The client interacts with the abstract item.

Suppose you have a software that needs to call hardware specific functionality, then you can call the abstraction of that functionality for the specific hardware.

Iterator Design Pattern is used by an aggregate object (container, collection, multiple items) which contains other objects group together as a unit. An iterator allows a programmer to traverse the elements of an aggregate object without exposing the implementation of that aggregate.

- A pointer with two main operations:
  - Element access
  - Element traversal

- Suppose you have a TV Remote Control
  - Up Arrow increases the current channel number.
  - Down Arrow decreases the channel number by 1.
  - **Note**: The keys increase or decrease the channel number without the viewer having to specify or having to know the current channel number.
  - The device implements an element traversal without exposing the implementation of the aggregate.

Categories of Design Paterns

- The 23 Gang of 4 patterns are grouped into three categories:
  - Creational Patterns
    - Abstract Factory
    - Builder
    - Factory Method
    - Prototype

        - Singleton
    - Structural Patterns
        - Adapter
        - Bridge
        - Composite
        - Facade
        - Flyweight
        - Proxy
    - Behavioral Patterns
        - Chain of responsibility
        - Command
        - Interpreter
        - Iterator
        - Mediator
        - Memento
        - Observer
        - State
        - Strategry
        - Template Method
        - Visitor

Strengths and Weaknesses

Strengths

- Promote reuse by solving a general design problem
- Provide highlevel design documentation
- A maintenance programmer who is familar with design patterns can easily comprehend a program that incorporates design patterns.

Weaknesses

- The use of the 23 standard design patterns may be an indication that the language is not powerful enough
- There is no systematic way to determine when and how to apply a design pattern
- Multiple interacting paterns are employed to obtain maximal benefit from design patterns
- It is all but impossible to retrofit patterns to an exisiting product

Luckily, the strengths of these patterns outweigh the weaknesses.

An interesting question in software engineering is: How do we formalize and hence automate design patterns?

Notes

No class on October 26

- Test 2
  - First 4 questions will essentially be the same as Test 1
  - Class Diagram
  - Activity Diagram may or may not be there
  - Sequence Diagram
  - Represent an architecture in UML.
  - Encapsulation, Polymorphism, Dynamic Binding
  - Reuse and Portability
  - Coupling and Cohesion
    - What's good, what's bad, and what's a good scale for both
    - High cohesiveness with small modules, but that tightly couples code. (There's a tradeoff)
  - Reuse and Portability
    - Be able to repurpose code.
    - Designing for reuse.