CS 315 - Oct 14, 2015

Chapter 8: Reusability and Portability

- Reuse concepts
    - Reuse is the use of components of one product to facilitate the development of a different product with different functionality
    - Two Types
        - **Opportunistic** (accidental)
            - First, the product is built
            - Then, the parts are put into the part database for reuse
        - **Systematic** (deliberate)
            - First, reusable parts are constructed
            - Then, products are built using these parts
    - Why?
        - To get products to the market faster
            - There is no need to design, implement, test, and document a reused component
        - On average, only 15% of new code serves an original purpose
            - In principle, 85% could be standardized and reused
            - In practice, reuse rates of no more than 40% are achieved
    - Impediments to Reuse
        - Not invented here (NIH) syndrome
        - Concerns about faults in potentially reusable routines
        - Storage-retrieval issues
        - Cost of reuse
            - The cost of making an item reusable
            - The cost of reusing an item
                - More mature (3 on the CMM scale) start to reuse components
            - The cost of defined and implementing a reuse process
        - Legal issue (contract software only)
        - Lack of source code for COTS components
    - Objects and Reuse
        - Claim of CS/D
            - An ideal module has functional cohesion

            - Problem
                - The data on which the module operates
            - We cannot reuse a module unless the data are identical
        - Claim of CS/D
            - The next best type of module has information cohesion
            - The is an object (an instance of a class)
            - An object comprises both data and action
            - The promotes reuse
    - Reuse during Design and Implementation
        - Various types of design reuse can be achieved
            - Some can be carried forward into implementation
            - Opportunistic reuse of designs is common when organization develops software in only one application domain
        - Library or Toolkit
            - A set of reusable routines
            - The user is responsible for the control logic
        - Application Frameworks
            - A framework incorporates the control logic of the design
            - The user inserts application-specific routines in the "hot spots"
            - Faster than reusing a toolkit
                - More of the design is reused
                - The logic is usually harder to design than the operations
        - Design Patterns
            - A pattern is a solution to a general problem
                - In the form of a set of interacting classes
            - The classes need to be customized
                - Wrapper and Adapter patterns
            - If a design pattern is reused, then its implementation can also probably be reused
            - Patterns can interact with other patterns
        - Software Architecture
            - An architecture consisting of
                - A toolkit
                - A framework
                - Multiple design patterns

    - Reuse of Software Architecture
        - Architecture reuse can lead to large-scale reuse
        - One mechanism
            - Software product lines
        - Architecture Patterns
            - Another way of achieving architectural reuse
            - Example: the model-view-controller (MVC) architecture pattern
                - Can be viewed as an extension to GUIs
                - Input-processing-output architecture
    - Reuse and Post-Delivery Maintenance
        - Reuse impacts maintenance more than development
        - Assumptions
            - 30% of entire product reused unchanged
            - 10% reused changed
- Portability Concepts
    - Have two products, P and P'
        - Functionally equivalent
        - Much easier to convert P into P' than to write P' from scratch
    - Impediments to Portability
        - Hardware
        - OS
        - Numerical/Memory
        - Compiler
        - Language
    - Why?
        - Is there any point in porting software?
            - Incompatibilities
            - One-off software
            - Selling company-specific software may give a competitor a huge advantage
        - On the contrary, portability is **essential**
            - Good software lasts 15 years or more
            - Hardware is changed every 4 years
        - Upwardly compatible hardware works
            - But it may not be cost effective
        - Portability can lead to increased profits

        - Multiple copy software
        - Documentation (especially manuals) must also be portable

    - Techniques for Achieving Portability
        - Obvious technique
            - Use standard constructs of a popular high-level language

        - Isolate implementation-dependent pieces
            - Example: Unix kernel, device drivers

        - Utilize levels of abstraction
            - Example: Graphical display routines

    - Portable Application Software
        - Use a popular programming language
        - Use a popular operating system
        - Adhere strictly to language standards
        - Avoid numerical incompatibilities
        - Document meticulously
        - File formats are often operating system-dependent
        - Porting structured data
            - Construct a sequential (unstructured) file and port it
            - Reconstruct the structured file on the target machine
            - The may be nontrivial for complex database models

- Design Patterns

    - Components
        - Name
            - Each pattern has an assigned name so it can be easily recognized
            - This gives us the vocabulary we can use to discuss design

        - Problem (context)
            - Each pattern is designed to address to a specific problem
            - Some also have conditions before the pattern can be used

        - Solution
            - Each pattern provides a solution to a problem
            - Components of that solution are also known as participants

        - Consequences
            - Costs and Benefits
            - Trade-offs of using design patterns: flexibility, extensibility, portability
            - Evaluate alternative changes

- Adapter Design Pattern
    - Adaptee: existing class with some behavior
    - Target: defines interface expected by the client
    - Adapter: implements the target interface using the functionality of the adaptee
    - Client: works with classes implementing the target interface
    - The *Adapter* Design Pattern:
        - Solves the implementation incompatibilities
        - Provides a general solution to the problem of permitting communication between two objects with incompatible interfaces
        - Provides a way for an object to permit access to its internal implementation without coupling clients to the structure of that internal implementation
    - That is, *Adapter* provides all the advantages of information hiding without having to actually hide the implementation details
- Composite Design Pattern
    - Compose objects into tree structures to represent whole/part hierarchies
        - Allow client to uniformly treat objects and compositions Atomic/primitive objects
        - Composite objects
    - Component
        - Declared the interface for objects in the composition
        - Implements 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 composition through the Component interface
- Bridge Design Pattern
    - Aim of the Bridge Design Pattern
        - To decouple an abstraction from its implementation so that the two can be changed independently of one another
    - Sometimes called a driver
        - Example: a printer driver or a video driver

        - The abstract operation are uncoupled from the hardware-dependent parts
        - If the hardware changes, the modifications to the design and the code are localized to only one side of the bridge
        - The bridge design pattern is a way of achieving information hiding via encapsulation
    - Iterator Design Pattern
        - An aggregate object (or container or collection) is an object that contains other objects grouped together as a unit
            - Examples: linked list, hash table
        - An iterator (or cursor) is a programming construct allows the programmer to traverse the elements of an aggregate object without exposing the implementation of that aggregate
        - An iterator may be viewed as a pointer with two main operations
            - Element access, or referencing a specific element in the collection
            - Element traversal, or modifying so it points to the next element in the collection
        - Implements element traversal without exposing the implementation of the aggregate
        - Implementation details of the elements are hidden from the iterator itself
            - We can use an iterator to process every element in a collection
            - Independently of the implementation of the container of the elements
        - Iterator allows different traversal methods
        - It even allows multiple traversals to be in progress concurrently
            - These traversals can be achieved without having the specific operations listed in the interface
    - Categories of Design Pattern
        - 23 Patterns grouped in 4 categories
            - Creational
            - Structural
                - Adapter
                - Bridge
                - Composite
            - Behavioral
        - Strengths and Weaknesses
            - Strengths
                - Design patterns promote rescue by solving a general design problem
                - Design patterns provide high-level design documentation, because patterns specify design abstractions
                - Implementations of many design patterns exist
                    - There no need to code or document those parts of the program

            - They still need to be tested, however

        - A maintenance programmer who is familiar with design patterns can be easily comprehend a program that incorporates design patterns

- Weaknesses
    - The use of the 23 standard design patterns may be an indication that the language we are using is not powerful enough
    - There is as yet no systematic way to determine when and how to apply design patterns
    - Multiple interacting patterns are employed to obtain maximal benefit from design patterns
        - But we do not yet have a systematic way of knowing when and how to use one pattern, let alone multiple interacting patterns

    - It is all but impossible to retrofit patterns to an existing software product

- The weaknesses of design patterns are outweighed by their strengths
- Research Issue: How do we formalize and hence automate design patterns?
    - This would make patterns much easier to use than at present