Software Engineering

CSC440/640
Prof. Schweitzer
Week 4

Reuse Concepts

- Reuse is the use of components of one product to facilitate the development of a different product with different functionality
- Two Types of Reuse
  - Opportunistic (accidental) reuse
    - First, the product is built
    - Then, parts are put into the part database for reuse
  - Systematic (deliberate) reuse
    - First, reusable parts are constructed
    - Then, products are built using these parts

Why Reuse?

- 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
- Why do so few organizations employ reuse?

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 the item
  - The cost of defining and implementing a reuse process
- Legal issues (contract software only)
- Lack of source code for COTS components
- The first four impediments can be overcome

Reuse Case Studies

- The first case study took place between 1976 and 1982
- Reuse mechanism used for COBOL design
  - Identical to what we use today for object-oriented application frameworks

Raytheon Missile Systems Division

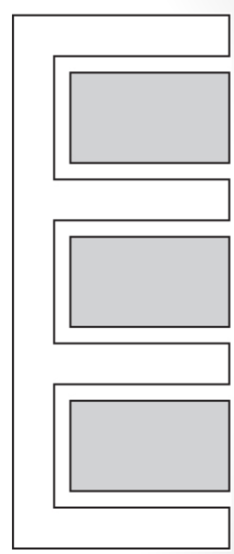
- Reuse rate of 60% was obtained
- Frameworks (“COBOL program logic structures”) were reused
- Paragraphs were filled in by functional modules
- Design and coding were quicker
- By 1983, there was a 50% increase in productivity
  - Logic structures had been reused over 5500 times
  - About 60% of code consisted of functional modules
- Raytheon hoped that maintenance costs would be reduced 60 to 80%
- Unfortunately, the division was closed before the data could be obtained

European Space Agency

- Ariane 5 rocket blew up 37 seconds after lift-off
  - Cost: $500 million
- Reason: An attempt was made to convert a 64-bit integer into a 16-bit unsigned integer
  - The Ada **`exception`** handler was omitted
- The on-board computers crashed, and so did the rocket
- Ten years before, it was mathematically proven that overflow was impossible — on the Ariane 4
- Because of performance constraints, conversions that could not lead to overflow were left unprotected
- The software was used, unchanged and untested, on the Ariane 5
  - However, the assumptions for the Ariane 4 did not hold for the Ariane 5

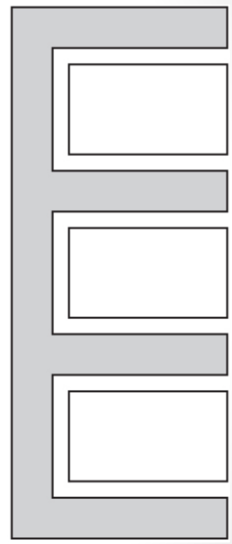
Library or Toolkit

- A set of reusable routines
- Examples:
  - Scientific software
  - GUI class library or toolkit
- The user is responsible for the control logic (white in figure)



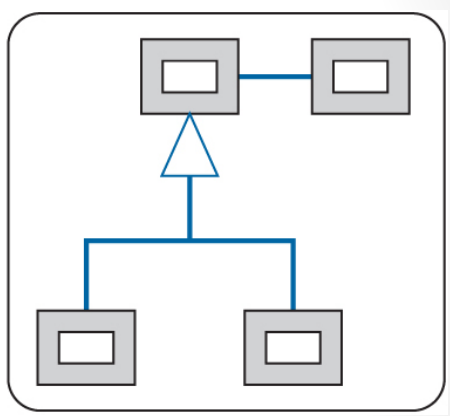
Application Frameworks

- A framework incorporates the control logic of the design
- The user inserts application-specific routines in the "hot spots" (white in figure)
- Faster than reusing a toolkit
  - More of the design is reused
  - The logic is usually harder to design than the operations
- Example:
  - IBM's Websphere
    - Formerly: e-Components, San Francisco
    - Utilizes Enterprise JavaBeans (classes that provide services for clients distributed throughout a network)



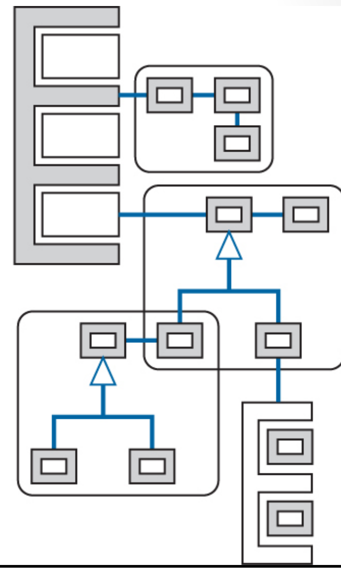
Design Patterns

- A pattern is a solution to a general design problem
  - In the form of a set of interacting classes
- The classes need to be customized (white in figure)



Software Architecture

- Encompasses a wide variety of design issues, including:
  - Organization in terms of components
  - How those components interact
- Example Figure
  - An architecture consisting of
    - A toolkit
    - A framework, and
    - Three design patterns



Architecture Patterns

- Another way of achieving architectural reuse
- Example: The model-view-controller (MVC) architecture pattern
  - Can be viewed as an extension to GUIs of the input–processing–output architecture

MVC component	Description	Corresponds to
Model	Core functionality, data	Processing
View	Displays information	Output
Controller	Handles user input	Input

Wrapper

- Suppose that when class P sends a message to class Q, it passes four parameters
- But Q expects only three parameters from P
- Modifying P or Q will cause widespread incompatibility problems elsewhere
- Instead, construct class A that accepts 4 parameters from P and passes three on to Q
  - Wrapper
- A wrapper is a special case of the *Adapter* design pattern
- *Adapter* solves the more general incompatibility problem
  - The pattern has to be tailored to the specific classes involved (see later)

Adapter Design Pattern

- The *Adapter* design pattern
  - Solves the implementation incompatibilities; but it also
  - Provides a general solution to the problem of permitting communication between two objects with incompatible interfaces; and it also
  - Provides a way for an object to permit access to its internal implementation without coupling clients to the structure of that internal implementation

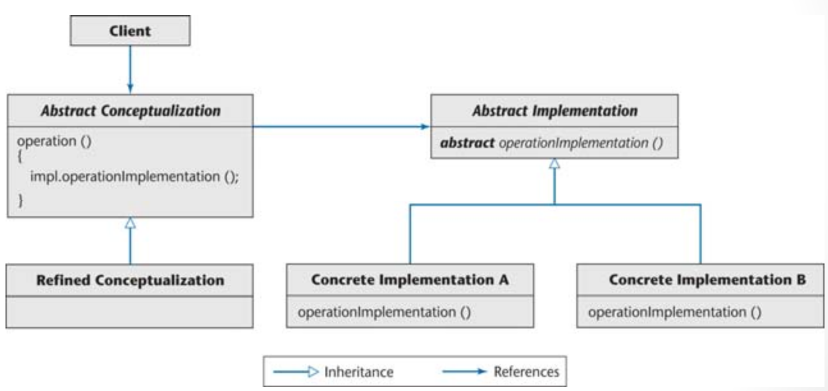
- That is, *Adapter* provides all the advantages of information hiding without having to actually hide the implementation details

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
- Suppose that part of a design is hardware-dependent, but the rest is not
- The design then consists of two pieces
  - The hardware-dependent parts are put on one side of the bridge
  - The hardware-independent parts are put on the other side

Bridge Design Pattern

- The *Bridge* design pattern can support multiple implementations



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 that allows a programmer to traverse the elements of an aggregate object without exposing the implementation of that aggregate

Iterator Design Pattern

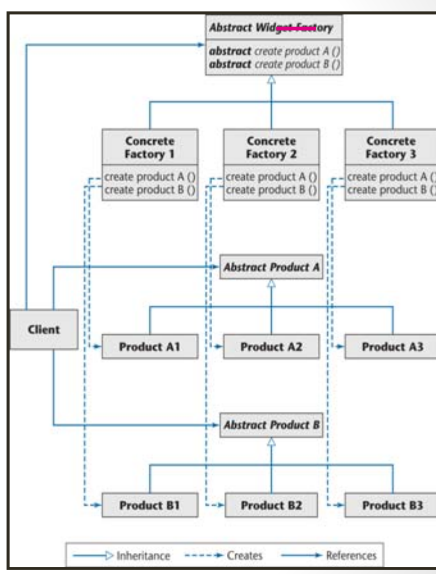
- An aggregate object (or container or collection) is an object that contains other objects grouped together as a unit
  - Examples: linked list, hash table
- An iterator (or cursor) is a programming construct that allows a programmer to traverse the elements of an aggregate object without exposing the 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; and
  - Element traversal, or modifying itself so it points to the next element in the collection

Iterator Design Pattern

- 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
- Instead, we have one uniform interface, namely
  - The four abstract operations *`first`*, *`next`*, *`isDone`*, and *`currentItem`* in ***`Abstract Iterator`***
  - with the specific traversal method(s) implemented in **`Concrete Iterator`**

Abstract Factory Design Pattern

- The Abstract Widget Factory is a special case of the *Abstract Factory* design pattern
- There is a typo in Figure 8.11 in the textbook: The word “Widget” should not appear in the top box



Strengths of Design Patterns

- Design patterns promote reuse by solving a general design problem
- Design patterns provide high-level design documentation, because patterns specify design abstractions
- Design patterns allow us to have a common language to discuss designs between human beings.
- Implementations of many design patterns exist
  - There is no need to code or document those parts of a program
  - They still need to be tested, however
- A maintenance programmer who is familiar with design patterns can easily comprehend a program that incorporates design patterns
  - Even if he or she has never seen that specific program before

Weaknesses of Design Patterns

- 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

Reuse and the World Wide Web

- A vast variety of code of all kinds is available on the Web for reuse
  - Also, smaller quantities of Designs and Patterns
- The Web supports code reuse on a previously unimagined scale
  - Copy and Paste Coding
- All this material is available free of charge
- The quality of the code varies widely
  - Code posted on the Web may or not be correct
  - Reuse of incorrect code is clearly unproductive

Maintenance Rules

Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live.

Portability

- Product P
  - Compiled by compiler C1, then runs on machine M1 under operating system O1
- Need product P', functionally equivalent to P
  - Compiled by compiler C2, then runs on machine M2 under operating system O2
- P is portable if it is cheaper to convert P into P' than to write P' from scratch

Problems with Portability

- Hardware Incompatibilities
  - Storage media incompatibilities
    - Example: Zip vs. DAT
  - Character code incompatibilities
    - Example: EBCDIC vs. ASCII
  - Word size
- Operating System Incompatibilities
  - Job control languages (JCL) can be vastly different
    - Syntactic differences
  - Virtual memory vs. overlays
  - API Differences
    - POSIX Support

More Problems with Portability

- Compiler Incompatibilities
- Language Incompatibilities
  - Java Virtual Machine Implementations
    - Write Once – Test Everywhere

Techniques for Achieving Portability

- Obvious technique
  - Use standard constructs of a popular high-level language
- But how is a portable operating system to be written?
  - 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

Portable Data

- 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
  - This may be nontrivial for complex database models
- Avoid Binary Formats
- Popular Text Based Formats
  - XML
  - JSON
  - CSV
  - INI

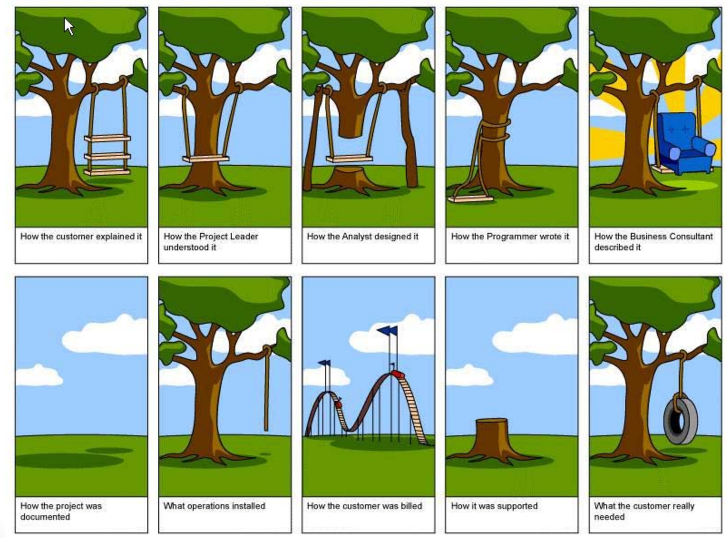
The Aim of the Requirements Workflow

- To answer the question:
  - What must the product be able to do?

Determining What the Client Needs

- Misconception
  - We must determine what the client wants
- "I know you believe you understood what you think I said, but I am not sure you realize that what you heard is not what I meant!"
- We must determine what the client needs

The Communication Problem



Overview of the Requirements Workflow

- First, gain an understanding of the application domain (or domain, for short)
  - The specific environment in which the target product is to operate
- Second, build a business model
  - Model the client's business processes
- Third, use the business model to determine the client's requirements
- Iterate the above steps

Understanding the Domain

- Every member of the development team must become fully familiar with the application domain
  - Correct terminology is essential
- Construct a glossary
  - A list of technical words used in the domain, and their meanings

Business Model

- A business model is a description of the business processes of an organization
- The business model gives an understanding of the client's business as a whole
  - This knowledge is essential for advising the client regarding computerization
- The systems analyst needs to obtain a detailed understanding of the various business processes
  - Different techniques are used, primarily interviewing

Interviewing

- The requirements team meet with the client and users to extract all relevant information
- There are two types of questions
  - *Close-ended* questions require a specific answer
  - *Open-ended* questions are posed to encourage the person being interviewed to speak out
- There are two types of interviews
  - In a *structured* interview, specific preplanned questions are asked, frequently close-ended
  - In an *unstructured* interview, questions are posed in response to the answers received, frequently open-ended

Interviewing

- Interviewing is not easy
  - An interview that is too unstructured will not yield much relevant information
  - The interviewer must be fully familiar with the application domain
  - The interviewer must remain open-minded at all times
- After the interview, the interviewer must prepare a written report
  - It is strongly advisable to give a copy of the report to the person who was interviewed
- A questionnaire is useful when the opinions of hundreds of individuals need to be determined
- Examination of business forms shows how the client currently does business

Other Techniques

- Direct observation (shadowing) of the employees while they perform their duties can be useful
  - Videotape cameras are a modern version of this technique
  - But, it can take a long time to analyze the tapes
  - Employees may view the cameras as an unwarranted invasion of privacy

Use Cases

- A use case models an interaction between the software product itself and the users of that software product (actors)
- Example:

Use Cases

- An actor is a member of the world outside the software product
- It is usually easy to identify an actor
  - An actor is frequently a user of the software product
- In general, an actor plays a role with regard to the software product. This role is
  - As a user; or
  - As an initiator; or
  - As someone who plays a critical part in the use case
- A user of the system can play more than one role
- Example: A customer of the bank can be
  - A **Borrower** or
  - A **Lender**

Use Cases

- Conversely, one actor can be a participant in multiple use cases
- Example: A **Borrower** may be an actor in
  - The `Borrow Money` use case;
  - The `Pay Interest on Loan` use case; and
  - The `Repay Loan Principal` use case
- Also, the actor **Borrower** may stand for many thousands of bank customers

Use Cases

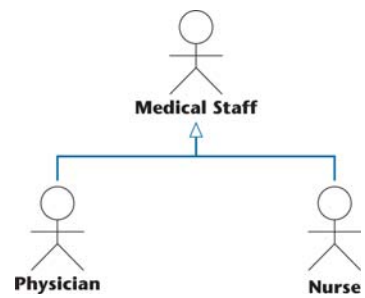
- An actor need not be a human being
- Example: An e-commerce information system has to interact with the credit card company information system
  - The credit card company information system is an actor from the viewpoint of the e-commerce information system
  - The e-commerce information system is an actor from the viewpoint of the credit card company information system

Use Cases

- A potential problem when identifying actors
  - Overlapping actors
- Example: Hospital software product
  - One use case has actor **Nurse**
  - A different use case has actor **Medical Staff**
  - Better:
    - Actors: **Physician** and **Nurse**

Use Cases

- Alternatively:
  - Actor **Medical Staff** with two specializations: **Physician** and **Nurse**
  - Similar to Inheritance



Initial Requirements

- The initial requirements are based on the initial business model
- Then they are refined
- The requirements are dynamic — there are frequent changes
  - Maintain a list of likely requirements, together with use cases of requirements approved by the client

Initial Requirements

- There are two categories of requirements
- A *functional requirement* specifies an action that the software product must be able to perform
  - Often expressed in terms of inputs and outputs
- A *nonfunctional requirement* specifies properties of the software product itself, such as
  - Platform constraints
  - Response times
  - Reliability

Initial Requirements

- Functional requirements are handled as part of the requirements and analysis workflows
- Some nonfunctional requirements have to wait until the design workflow
  - The detailed information for some nonfunctional requirements is not available until the requirements and analysis workflows have been completed

The Classical Requirements Phase

- There is no such thing as "object-oriented requirements"
  - The requirements workflow has nothing to do with how the product is to be built
- However, the approach presented in this chapter is
  - Model oriented, and therefore
  - Object oriented

The Classical Requirements Phase

- The classical approach to requirements
  - Requirements elicitation
  - Requirements analysis
  - Construction of a rapid prototype
  - Client and future users experiment with the rapid prototype

Rapid Prototyping

- Hastily built ("rapid")
  - Imperfections can be ignored
- Exhibits only key functionality
- Emphasis on only what the client sees
  - Error checking, file updating can be ignored
- Aim:
  - To provide the client with an understanding of the product
- A rapid prototype is built for change
  - Languages for rapid prototyping include 4GLs and interpreted languages. Visual Basic became very popular for this.

Human Factors

- The client *and intended users* must interact with the user interface
- Human-computer interface (HCI)
  - Menu, not command line
  - "Point and click"
  - Windows, icons, pull-down menus
- Human factors must be taken into account
  - Avoid a lengthy sequence of menus
  - Allow the expertise level of an interface to be modified
  - Uniformity of appearance is important
  - Advanced psychology vs. common sense?
- Rapid prototype of the HCI of every product is obligatory

Reusing the Rapid Prototype

- Reusing a rapid prototype is essentially code-and-fix
- Changes are made to a working product
  - Expensive
- Maintenance is hard without specification and design documents
- Real-time constraints are hard to meet

Reusing the Rapid Prototype

- One way to ensure that the rapid prototype is discarded
  - Implement it in a different language from that of the target product
- Generated code can be reused
- We can safely retain (parts of) a rapid prototype if
  - This is prearranged
  - Those parts pass SQA inspections
  - However, this is not “classical” rapid prototyping

CASE Tools for the Requirements Workflow

- We need graphical tools for UML diagrams
  - To make it easy to change UML diagrams
  - The documentation is stored in the tool and therefore is always available
- Such tools are sometimes hard to use
- The diagrams may need considerable “tweaking”
- Overall, the strengths outweigh the weaknesses

CASE Tools for the Requirements Workflow

- Graphical CASE environments extended to support UML include
  - System Architect
  - Software through Pictures
- Object-oriented CASE environments include
  - IBM Rational Rose
  - Together
  - ArgoUML (open source)

Challenges of the Requirements Phase

- Employees of the client organization often feel threatened by computerization
- The requirements team members must be able to negotiate
  - The client's needs may have to be scaled down
- Key employees of the client organization may not have the time for essential in-depth discussions
- Flexibility and objectivity are essential

Affinity Diagramming

- Decide on a Problem Statement
- Brainstorm all ideas that you can associated with the problem statement. After you have some, add them to the wall
- As you add ideas to the wall, place common ideas under each other, and re-order them in the steps necessary to solve the problem
- Once all ideas are mapped on the wall, decide on proper categories for the steps, and create category headers.
- Feel free to move *other people's post-it notes around*
- Categories tend to become "Themes for a Release"
- Post-It's become "Epics" or possibly a "User Story" depending on the item

Affinity Diagramming Exercise

- **Problem Statement**: *Order and Receive a Pizza to the Lab*

- Write steps needed to complete the problem statement on a Post-It Note
- *One Step Per Post-It Note*
- Take 10 Minutes to Brain Storm Ideas
- After 5 Minutes, add whatever you have at that point to the wall
- Organize and order the Post-It's as you go
- After brainstorming is complete, we'll finish organizing and categorize the steps.

Next Week

- Reading: Chapters 9 & 12
- Reminder: Requirements Specification Due Next Wednesday at 6PM