CHAPTER 08
REUSABILITY AND PORTABILITY

Reuse Concepts

- Reuse — Using components of one product to facilitate development of a different product with different functionality
- E.g., a module, code fragment, a design, a part of a manual, test data, duration and cost estimates, etc.
- Two types of reuse
  - Opportunistic (accidental) reuse — Developers realize that a component can be reused
  - Systematic (deliberate) reuse — Software components constructed specifically for possible future reuse. Components more likely to be easy and safe to reuse
- Application programming interface (API) — A set of operating system calls that facilitate programming
  - E.g., Win32 for Microsoft Windows XP, Cocoa for Max OS X

Reference

Reuse Concepts

- On average, only about 15% of any software product serves an original purpose.
- The other 85% in theory can be standardized and reused in future products.
- Reuse rates
  - Theoretical upper limit of 85%.
  - Reuse rates of 40% can be achieved in practice.
- A product is portable if it is significantly easier to modify it to run on another system configuration (compiler, hardware, operating system) than recode it from scratch.

Impediments to Reuse

- Software professionals would rather work from scratch than reuse:
  - Known as not invented here (NIH) syndrome.
  - Management can address this issue: Financial incentives to promote reuse.
- Many developers would be willing to reuse provided they could be sure no faults would be introduced;
  - Exhaustive testing of potentially reusable routines.
- How to effectively store and retrieve components to reuse:
  - Database solutions.
  - E.g., choosing an appropriate sort routine.
- Reuse can be expensive:
  - Three costs: Cost of making something reusable, cost of using, and cost of defining and implementing a reuse process.
  - Costs increase reported from 11% to 480%.
- Legal issues can arise with contract software:
  - Using a component of a client’s product in a new product for a different client.
  - Client owning software that is reused for another client.
  - Copyright violation of the first client.
  - Not a problem with internal software.
- Reuse of commercial off-the-shelf (COTS) components:
  - Developers are not given the source code.
  - Software that reuses COTS components has limited extensibility and modifiability.

Objects and Reuse

- A well-designed object is the fundamental building block of software.
- It models all aspects of a particular real-world entity (conceptual independence or encapsulation).
- It conceals implementation of both its data and operations (physical independence or information hiding).
- Object-oriented paradigm utilized correctly:
  - Resulting modules (objects) have informational cohesion.
  - This promotes reuse.
Reuse during the Design and Implementation

- Different types of reuse are possible during the design: Vary from one to two artifacts to the architecture of complete software product
  
  **Design Reuse**
  - During design, a module or class from an earlier design can be reused
  - Particularly common when developing software in one specific application domain
  - Advantages
    - Faster and higher quality design since tested designs are reused
    - If design is reused: Likely implementation can be reused
  - Library reuse
    - A set of related reusable routines
    - E.g., LAPACK++ scientific library (matrix inversions, Eigen values)

reuse during the Design and Implementation

- Toolkit reuse
  - A set of classes that can handle every aspect of a domain
  - E.g., Java Abstract Windowing Toolkit (GUI class)

  **Application Frameworks**
  - Application framework — Incorporates control logic of a design
  - Developers have to design application-specific operations
  - Hot spots — Places where application specific operations are inserted
  - E.g., a set of classes for design of a compiler
  - Results in faster development than toolkit reuse
    - Less to design from scratch when framework is reused
    - Portion reused is more difficult to design

Reuse during the Design and Implementation

- E.g., IBM’s WebSphere
  - Formerly known as e-Components
  - A framework for building online information systems in Java
  - Utilizes Enterprise JavaBeans (classes that provide services for clients distributed throughout a network)
- E.g., Borland Visual Component Library (VCL)
- An object-oriented set of frameworks
- For building GUIs in Windows-based applications
- Can perform standard windowing operations (e.g., moving and resizing windows, processing input via dialog boxes, handling events like mouse clicks or menu selections)

Reuse during the Design and Implementation

- Design pattern — A solution to a general design problem in form of a set of interacting classes that have to be customized to create a specific design
  - Patterns can interact with other patterns
  - If a design pattern is reused: Implementation of that pattern probably also can be reused

  **Software Architecture**
  - Software architecture encompasses a variety of design issues
    - Organization of product in terms of its components
    - Product-level control structures
    - Communication and synchronization
    - Databases and data access
    - Physical distribution of components
Reuse during the Design and Implementation

- Performance
- Choice of design alternatives

- Software architecture is a considerably more wide-ranging concept than design patterns
- Software architecture involves description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns

One way that reuse of architecture is achieved in practice is with software product lines

- Software product line — Set of software products in same application domain built by using core assets (common software artifacts)
  - A software architecture common to a number of software products
  - Instantiate this architecture when developing a new product
- E.g., Hewlett-Packard firmware architecture for printers
  - Instantiated for each new printer model
  - Person-hours to develop firmware for a new printer decreased by a factor of 4
  - Time to develop firmware decreased by a factor of 3
  - For recent printers over 70% of the firmware components are reused, almost unchanged

Architecture patterns — Another way of achieving architectural reuse (model-view-controller and three-tier)

- Model-View-Controller (MVC)
- Popular architecture pattern
- Extension of input-processing-output architecture to GUI domain
- Decomposition of architecture into model, view, and controller
- Components can be changed independently of the other two
- Enhancing reusability

<table>
<thead>
<tr>
<th>MVC component</th>
<th>Description</th>
<th>Corresponds to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Core functionality, data</td>
<td>Processing</td>
</tr>
<tr>
<td>View</td>
<td>Displays information</td>
<td>Output</td>
</tr>
<tr>
<td>Controller</td>
<td>Handles user input</td>
<td>Input</td>
</tr>
</tbody>
</table>
Reuse during the Design and Implementation

- Three-Tier Architecture
- Another popular architecture pattern
- Presentation logic tier — Accepts user input and generates output
- Business logic tier — Incorporates processing of business rules
- Data access logic tier — Communicates with underlying database
- Components can be changed independently of the other two
- Promotes reuse

**Component-Based Software Engineering**

- Goal of component-based software engineering — To construct a standard collection of reusable components

More on Design Patterns

- Design patterns are important in object-oriented software engineering

**Adapter Design Pattern**

- Permitting communication between two objects with incompatible interfaces
- Methods to send messages as requests
- Figure 8.6

**Bridge Design Pattern**

- Aim to decouple an abstraction from its implementation: Two be changed independently
- Sometimes called a driver (e.g., printer driver)
- Design in two pieces
  - Part of design hardware-dependent and rest is not
  - Hardware-dependent pieces on one side
  - Hardware-independent pieces on the other side
  - Operations decoupled
  - A bridge between the two parts
  - Hardware changes: Modifications localized to one side of bridge
- Also for decoupling operating system-dependent pieces
- Also for decoupling compiler-dependent pieces

More on Design Patterns

- Figure 8.7
More on Design Patterns

**Iterator Design Pattern**
- Aggregate object — Contains other objects grouped together as a unit
- Iterator — Programming construct allowing a programmer to traverse elements of an aggregate object without exposing implementation of aggregate
  - Referred to as a cursor within a database context
- Iterator — A pointer with two main operations of element access and element traversal
- Example of iterator: Television remote control
  - Up and down channel buttons
  - Without viewer specifying current channel number
  - Element traversal without exposing implementation
- Use an iterator to process every element in a collection, independently of implementation

**Abstract Factory Design Pattern**
- Design in such a way that the application program is uncoupled from the specific operating system
  - E.g., Linux, Mac OS, Windows
- Each class contains specific methods for a given operating system

**More on Design Patterns**

- Allows different traversal methods
- Figure 8.9

**Figure 8.9**

- Abstract Aggregate
  - abstract createrator(): Iterator

- Concrete Aggregate
  - createrator()
  - return new concretetator(this);

- Abstract Iterator
  - abstract first(): Item
  - abstract next(): Boolean
  - abstract isDone(): Boolean
  - abstract currentItem(): Item

- Concrete Iterator
  - first()
  - next()
  - isDone()
  - currentItem()

**More on Design Patterns**

- Figure 8.11

**Figure 8.11**

- Abstract Widget Factory
  - abstract create product A()
  - abstract create product B()
Categories of Design Patterns

- 23 standard design patterns
- Divided into three categories
- Creational patterns — Solve design problems by creating objects
  - E.g., abstract factory pattern
- Structural patterns — Solve design problems by identifying a simple way to realize relationships between entities
  - E.g., Adapter pattern and bridge pattern
- Behavioral patterns — Solve design problems by identifying communicational patterns between objects
  - E.g., iterator pattern
- Other lists and categories proposed: Not widely accepted

Strengths and Weaknesses of Design Patterns

- Strengths
  - Promote reuse by solving a general design problem
  - Provide high-level documentation of design
  - Implementations exist: No need to code or document those parts
  - If a maintenance programmer is familiar: Easier to comprehend the program
  - Research into automated detection of design patterns
- Weaknesses
  - Use of design patterns may be an indication of the language not being powerful enough
  - No systematic way to determine when and how to apply design patterns

Strengths and Weaknesses of Design Patterns

- For maximal benefits, multiple interacting patterns are employed
- Impossible to retrofit patterns to an existing software product
- Overall
- Weaknesses of design patterns are outweighed by their strengths
- Once research to formalize and automate design patterns succeeds: Patterns will be easier to use
Reuse and the World Wide Web

- Programmers place code on Web
  - From students to professional programmers
  - Wide variety of programming languages
  - Broad range of applications
- Design and patterns also available to a much lesser extent
- The Web supports code reuse
  - Anyone can download code
  - Source should be acknowledged
- Drawbacks of reusing code from the Web
  - Quality of code varies widely: No guarantee for quality
  - Code can be reused and fixed by an organization, and original author will be unaware: No way to know who downloaded and who reused

Reuse and Postdelivery Maintenance

- Reuse has a greater impact on postdelivery maintenance than on development
- Reused components generally are
  - Well designed
  - Thoroughly tested
  - Comprehensively documented
- Average cost saving assumptions:
  - 40% of a new product consists of reused components
  - 3/4 of which are reused unchanged
- Figure 8.13

<table>
<thead>
<tr>
<th>Activity</th>
<th>Percentage of Total Cost over Product Lifetime</th>
<th>Percentage Savings over Product Lifetime due to Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>33%</td>
<td>9.3%</td>
</tr>
<tr>
<td>Postdelivery maintenance</td>
<td>67%</td>
<td>17.9%</td>
</tr>
</tbody>
</table>

Portability

- Portable product — Significantly easier (less expensive) to modify the product to run on another system configuration (compiler, hardware, operating system) than recode it from scratch
  - From source computer
  - To target computer
- Advantages
  - Costs can be recouped by selling versions
  - Client organization purchases new hardware
- Incompatibilities
  - Hardware incompatibilities
  - Operating system incompatibilities
  - Numerical software incompatibilities (arithmetic types)
  - Compiler incompatibilities

Why Portability

- Potential partial recoup of cost by selling for different platforms
- Life of a software product generally is longer than hardware
  - Good software products can have a life of 15 years or more
  - Hardware frequently is changed every 4 years
Techniques for Achieving Portability

Portable System Software
- Isolation of implementation-dependent pieces of software
- E.g., UNIX operating system
  - 9000 lines of operating system implemented in C
  - 1000 lines of kernel in assembler: Reimplemented each time
  - 1000 lines of C code consisted of device drivers: Reimplemented each time
  - 8000 lines of C code remained largely unchanged
- Use of levels of abstraction
  - E.g., graphics display routines
    - High-level language for user’s commands
    - Low-level code artifact to execute commands
    - When ported: No change to user’s code, no change to upper level of graphical display routines
    - Only low-level code artifacts reimplemented

Portable Application Software
- Product implemented in a high-level language
- Language selection on basis of cost-benefit analysis (if there is a choice)
- Decisions that result in a more portable product (e.g., not using a case-sensitive compiler)
- Operating system should be a popular one (if there is a choice)
  - UNIX has been implemented on a wide range of hardware
  - For Personal computers: To be determined whether Linux will overtake windows
- Portable Operating System Interface for Computer Environments (POSIX) — To facilitate moving of software from one UNIX-based system to another

Portable Data
- Problem of portability of data
  - Differences in the format of files
  - File headers are operating system specific
- Use of unstructured (sequential) file
- Summary of strengths of and impediments to reuse and portability

Model-Driven Architecture
- Model-Driven Architecture (MDA) — Entirely decoupling the functionality of a software product from its implementation
- Emerging technology that achieves portability
# Techniques for Achieving Portability

**Figure 8.14**

<table>
<thead>
<tr>
<th>Strengths</th>
<th>Impediments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reuse</strong></td>
<td></td>
</tr>
<tr>
<td>Shorter development time (Section 8.1)</td>
<td>NIH syndrome (Section 8.2)</td>
</tr>
<tr>
<td>Lower development cost (Section 8.1)</td>
<td>Potential quality issues (Section 8.2)</td>
</tr>
<tr>
<td>Higher-quality software (Section 8.1)</td>
<td>Retrieval issues (Section 8.2)</td>
</tr>
<tr>
<td>Shorter maintenance time (Section 8.10)</td>
<td>Cost of making a component reusable (opportunistic reuse) (Section 8.2)</td>
</tr>
<tr>
<td>Lower maintenance cost (Section 8.10)</td>
<td>Cost of making a component for future reuse (systematic reuse) (Section 8.2)</td>
</tr>
<tr>
<td>Legal issues (contract software only) (Section 8.2)</td>
<td>Lack of source code for COTS components (Section 8.2)</td>
</tr>
<tr>
<td><strong>Portability</strong></td>
<td></td>
</tr>
<tr>
<td>Software has to be ported to new hardware every 4 years or so (Section 8.12)</td>
<td>Potential incompatibilities:</td>
</tr>
<tr>
<td>More copies of COTS software can be sold (Section 8.12)</td>
<td>Hardware (Section 8.11.1)</td>
</tr>
<tr>
<td></td>
<td>Operating systems (Section 8.11.2)</td>
</tr>
<tr>
<td></td>
<td>Numerical software (Section 8.11.3)</td>
</tr>
<tr>
<td></td>
<td>Compilers (Section 8.11.4)</td>
</tr>
<tr>
<td></td>
<td>Data formats (Section 8.13.3)</td>
</tr>
</tbody>
</table>