CHAPTER 8

REUSABILITY AND PORTABILITY

1

8.1 Reuse Concepts

- Reuse is the use of components of one product to facilitate the development of a different product with different functionality

2

The 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

3

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?

4

8.2 Impediments to Reuse

- Not invented here (NIH) syndrome
- Concerns about faults in potentially reusable routines
- Storage–retrieval issues

5

Impediments to Reuse

- 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

6

8.3 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

7

8.3.1 Raytheon Missile Systems Division

- Data-processing software
- Systematic reuse of
  - Designs
    - 6 code templates
  - COBOL code
    - 3200 reusable modules

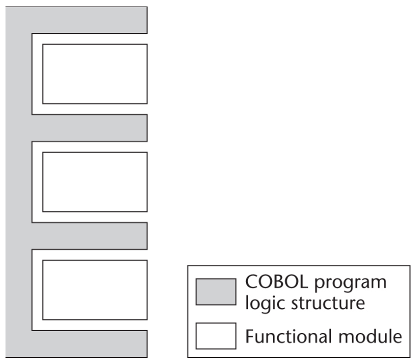


Figure 8.1

8

Raytheon Missile Systems Division (contd)

- 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

9

Raytheon Missile Systems Division (contd)

- 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

10

8.3.2 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

11

The Conversion was Unnecessary

- Computations on the inertial reference system can stop 9 seconds before lift-off
- But if there is a subsequent hold in the countdown, it takes several hours to reset the inertial reference system
- Computations therefore continue 50 seconds into the flight

12

The Cause of the Problem

- 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

13

European Space Agency (contd)

- Lesson:
    - Software developed in one context needs to be retested when integrated into another context

14

8.4 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

15

Objects and Reuse (contd)

- Claim of CS/D:
  - The next best type of module has informational cohesion
  - This is an object (an instance of a class)
- An object comprises both data and action
- This promotes reuse

16

8.5 Reuse During Design and Implementation

- Various types of design reuse can be achieved
  - Some can be carried forward into implementation

17

8.5.1 Design Reuse

- Opportunistic reuse of designs is common when an organization develops software in only one application domain

18

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)

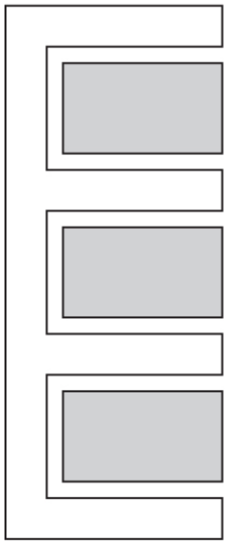


Figure 8.2(a)

19

8.5.2 Application Frameworks

- A framework incorporates the control logic of the design
- The user inserts application-specific routines in the "hot spots" (white in figure)
- Remark: Figure 8.2(b) is identical to Figure 8.1

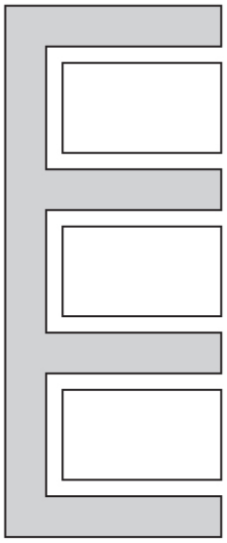


Figure 8.2(b)

20

Application Frameworks (contd)

- 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)

21

8.5.3 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)

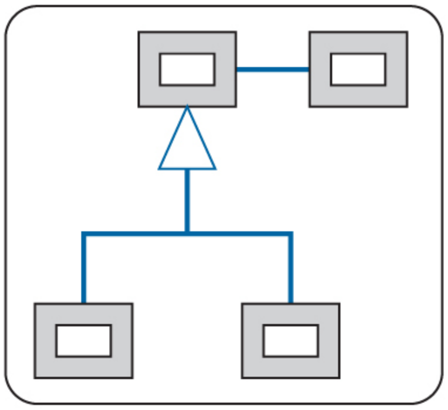


Figure 8.2(c)

22

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

23

Wrapper (contd)

- 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)

24

Design Patterns (contd)

- If a design pattern is reused, then its implementation can also probably be reused

- Patterns can interact with other patterns

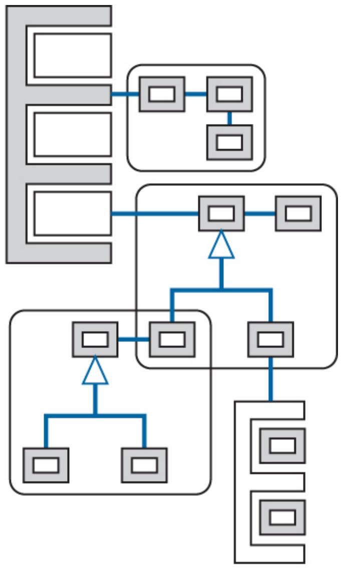


Figure 8.2(d)

25

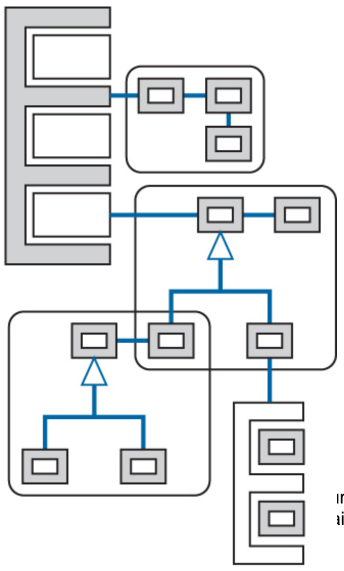
8.5.4 Software Architecture

- Encompasses a wide variety of design issues, including:
  - Organization in terms of components
  - How those components interact

26

Software Architecture

- An architecture consisting of
  - A toolkit
  - A framework, and
  - Three design patterns



27

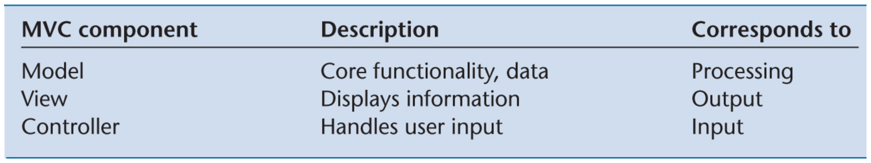
Reuse of Software Architecture

- Architecture reuse can lead to large-scale reuse
- One mechanism:
  - Software product lines
- Case study:
  - Firmware for Hewlett-Packard printers (1995-98)
    - Person–hours to develop firmware decreased by a factor of 4
    - Time to develop firmware decreased by a factor of 3
    - Reuse increased to over 70% of components

28

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

Figure 8.3

29

8.5.5 Component-Based Software Engineering

- Goal: To construct all software out of a standard collection of reusable components
- This emerging technology is outlined in Section 18.3

30

8.6 More on Design Patterns

- Case study that illustrates the *Adapter* design pattern

31

Adapter Design Pattern (contd)

- 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

32

8.6.3 *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
- Often used in *drivers*
  - Example: a printer driver or a video driver

33

8.6.3 *Bridge* Design Pattern

- 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

34

Bridge Design Pattern (contd)

- The abstract operations are uncoupled from the hardware-dependent parts
  - There is a "bridge" between the two 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

35

8.6.4 *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

36

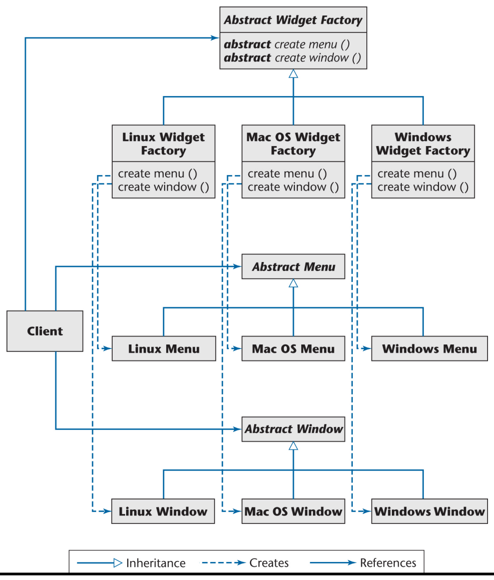
Iterator Design Pattern (contd)

- 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

37

8.6.5 *Abstract Factory* Design Pattern

- We want a widget generator
  - A program that will generate widgets that can run under different operating systems



38

8.8 Strengths and Weaknesses of Design Patterns

- Strengths
  - Design patterns promote reuse by solving a general design problem
  - Design patterns provide high-level design documentation, because patterns specify design abstractions

39

Strengths and Weaknesses of Design Patterns (contd)

- 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

40

Strengths and Weaknesses of Design Patterns (contd)

- 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

41

Strengths and Weaknesses of Design Patterns (contd)

- 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

42

Strengths and Weaknesses of Design Patterns (contd)

- 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

43

8.9 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
- All this material is available free of charge

44

Problems with Reusing Code from the Web

- The quality of the code varies widely
  - Code posted on the Web may or not be correct
  - Reuse of incorrect code is clearly unproductive

45

Problems with Reusing Code from the Web (contd)

- Records are kept of reuse within an organization
  - If a fault is later found in the original code, the reused code can also be fixed
- If a fault is found in a code segment that has been posted on the Web and downloaded many times
  - We cannot determine who downloaded the code, and
  - Whether or not it was actually reused after downloading

46

Problems with Reusing Code from the Web (contd)

- The World Wide Web promotes widespread reuse
- However
  - The quality of the downloaded material may be abysmal, and
  - The consequences of reuse may be severe

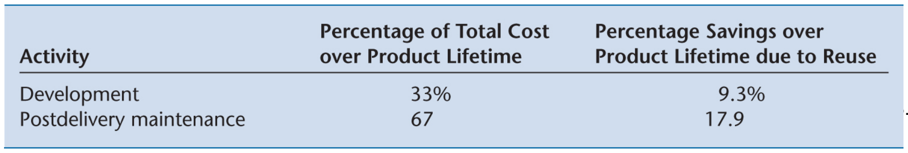
47

8.10 Reuse and Postdelivery Maintenance

- Reuse impacts maintenance more than development
- Assumptions
  - 30% of entire product reused unchanged
  - 10% reused changed

48

Results



Activity	Percentage of Total Cost over Product Lifetime	Percentage Savings over Product Lifetime due to Reuse
Development	33%	9.3%
Postdelivery maintenance	67	17.9

.13

- Savings during maintenance are nearly 18%
- Savings during development are about 9.3%

49

8.11 Portability

- Product P
  - Compiled by compiler C_1, then runs on machine M_1 under operating system O_1
- Need product P', functionally equivalent to P
  - Compiled by compiler C_2, then runs on machine M_2 under operating system O_2
- P is *portable* if it is cheaper to convert P into P' than to write P' from scratch

50

8.11.1 Hardware Incompatibilities

- Storage media incompatibilities
  - Example: Zip vs. DAT
- Character code incompatibilities
  - Example: EBCDIC vs. ASCII
- Word size

51

Hardware Incompatibilities (contd)

- IBM System/360-370 series
  - [One of] The most successful line of computers ever
  - Full upward compatibility

52

8.11.2 Operating System Incompatibilities

- Job control languages (JCL) can be vastly different
  - Syntactic differences
- Virtual memory vs. overlays

53

8.11.3 Numerical Software Incompatibilities

- Differences in word size can affect accuracy
- No problems with
  - Java
  - Ada

54

8.11.4 Compiler Incompatibilities

- FORTRAN standard is not enforced
- COBOL standard permits subsets, supersets
- ANSI C standard (1989)
  - Most compilers use the *pcc* front end
  - The *lint* processor aids portability
- ANSI C++ standard (1998)

55

Language Incompatibilities (contd)

- Ada standard — the only successful language standard
  - First enforced legally (via trademarking)
  - Then by economic forces
- Java is still evolving
  - Sun copyrighted the name to ensure standardization

56

8.12 Why Portability?

- Is there any point in porting software?
  - Incompatibilities
  - One-off software
  - Selling company-specific software may give a competitor a huge advantage

57

Why Portability? (contd)

- 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

58

8.13 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?

59

8.13.1 Portable System Software

- Isolate implementation-dependent pieces
  - Example: UNIX kernel, device-drivers
- Utilize levels of abstraction
  - Example: Graphical display routines

60

8.13.2 Portable Application Software

- Use a popular programming language
- Use a popular operating system
- Adhere strictly to language standards
- Avoid numerical incompatibilities
- **Document meticulously**

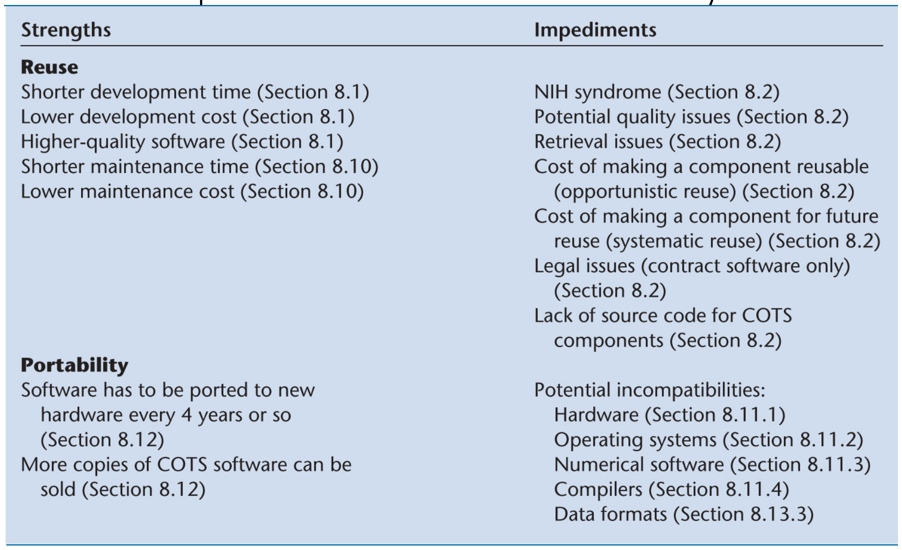
61

8.13.3 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

62

Strengths of and Impediments to Reuse and Portability



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

Figure 8.14

63