

Object-Oriented and Classical Software Engineering

Eighth Edition, WCB/McGraw-Hill, 2011

Stephen R. Schach

REUSABILITY AND PORTABILITY

- ❑ Reuse concepts
- ❑ Impediments to reuse
- ❑ Reuse case studies
- ❑ Objects and reuse
- ❑ Reuse during design and implementation
- ❑ More on design patterns
- ❑ Categories of design patterns
- ❑ Strengths and weaknesses of design patterns

- Reuse and the world wide web
- Reuse and postdelivery maintenance
- Portability
- Why portability?
- Techniques for achieving portability

8.1 Reuse Concepts

Slide 8.5

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

The Two Types of Reuse

Slide 8.6

- 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?

Slide 8.7

- ❑ 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?

8.2 Impediments to Reuse

Slide 8.8

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

Impediments to Reuse (contd)

Slide 8.9

- 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

8.3 Reuse Case Studies

Slide 8.10

- 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

8.3.1 Raytheon Missile Systems Division

Slide 8.11

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

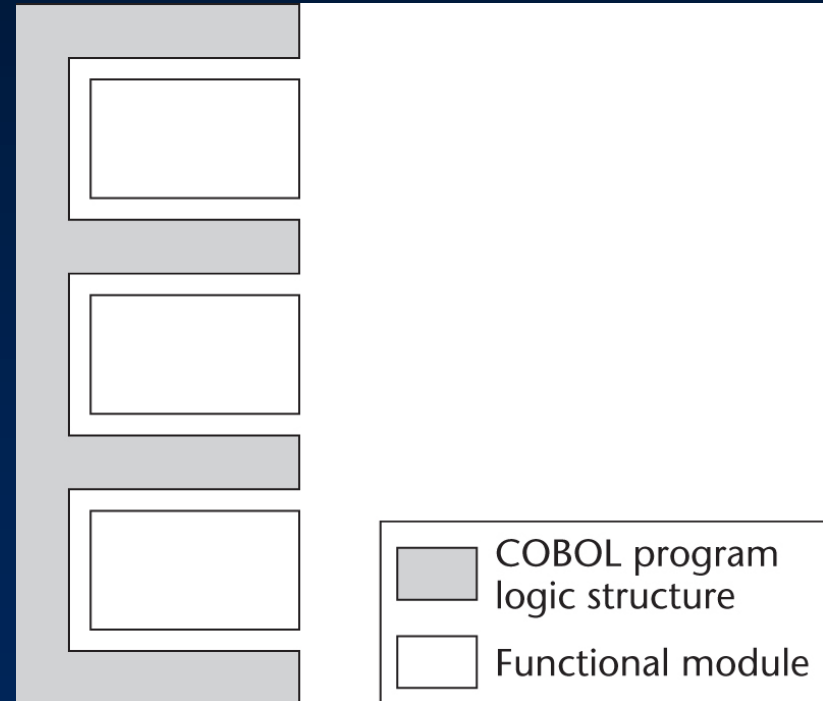


Figure 8.1

- ❑ 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

Raytheon Missile Systems Division (contd)

Slide 8.13

- 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

8.3.2 European Space Agency

Slide 8.14

- ❑ 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

The Conversion was Unnecessary

Slide 8.15

- ❑ 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

The Cause of the Problem

Slide 8.16

- ❑ 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

□ Lesson:

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

8.4 Objects and Reuse

Slide 8.18

- 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 informational cohesion
 - This is an object (an instance of a class)
- An object comprises both data and action
- This promotes reuse

8.5 Reuse During Design and Implementation

Slide 8.20

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

8.5.1 Design Reuse

Slide 8.21

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

- 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)

8.5.2 Application Frameworks

Slide 8.23

- 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)

- 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)

8.5.3 Design Patterns

Slide 8.25

- 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)

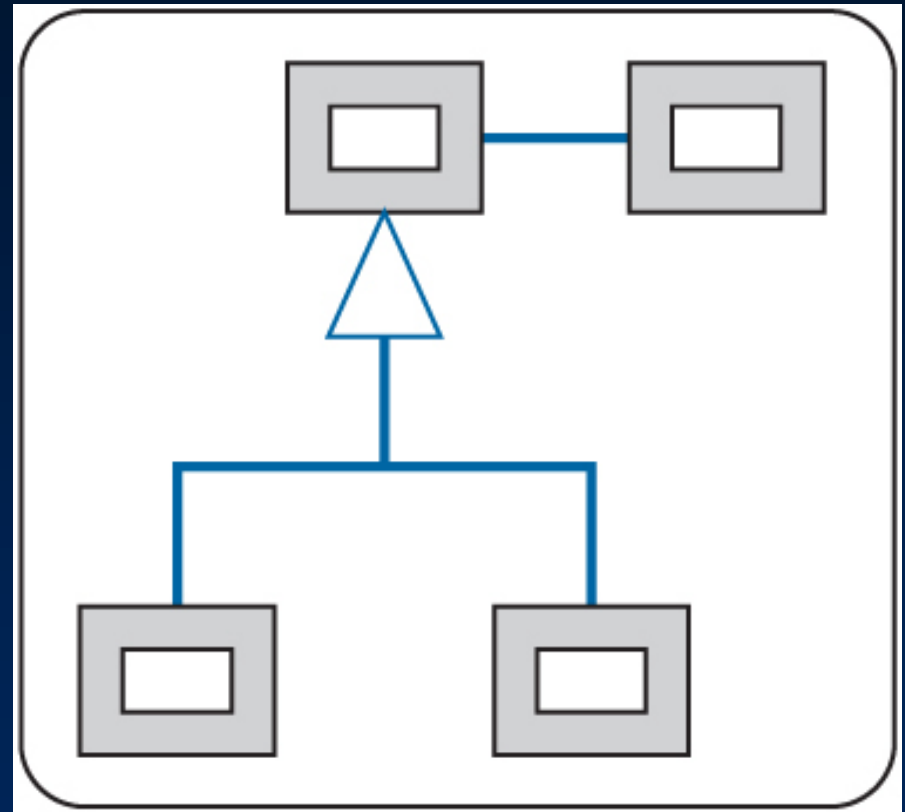


Figure 8.2(c)

- 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)

Design Patterns (contd)

Slide 8.28

- If a design pattern is reused, then its implementation can also probably be reused
- Patterns can interact with other patterns

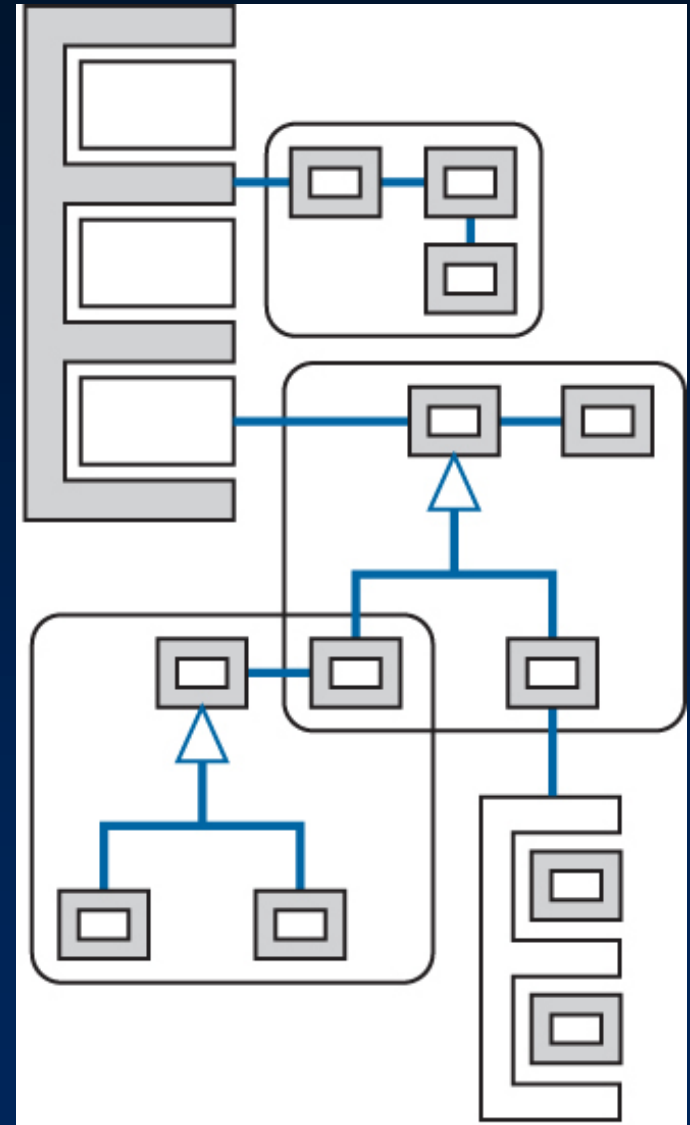


Figure 8.2(d)

8.5.4 Software Architecture

Slide 8.29

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

Software Architecture (contd)

Slide 8.30

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

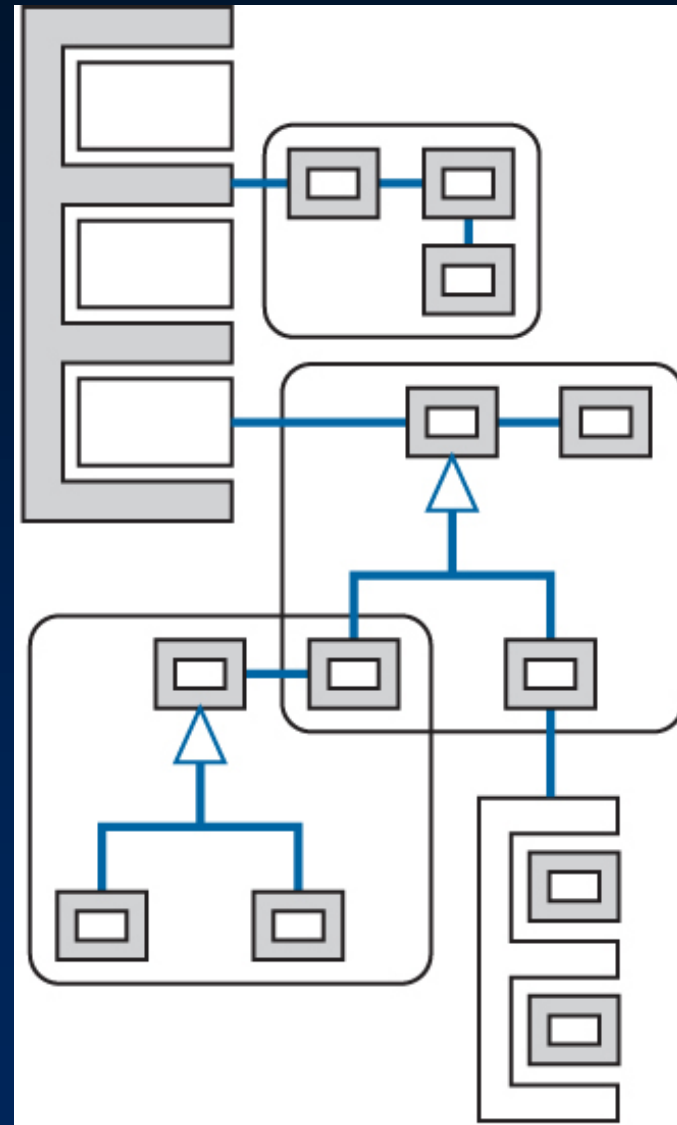


Figure 8.2(d)
(again)

- 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

- 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

8.5.5 Component-Based Software Engineering

Slide 8.33

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

8.6 More on Design Patterns

Slide 8.34

- Case study that illustrates the *Adapter* design pattern

8.6.1 FLIC Mini Case Study

Slide 8.35

- Until recently, premiums at Flintstock Life Insurance Company (FLIC) depended on both the age and the gender of the applicant
- FLIC has now decided that certain policies will be gender-neutral
 - The premium will depend solely on the age of the applicant

FLIC Mini Case Study (contd)

Slide 8.36

- Currently, premiums are computed by sending a message to method `computePremium` of class `Applicant`
 - Passing the `age` and `gender` of the applicant
- Now a different computation has to be made
 - Based solely on the applicant's `gender`
- A new class is written — `Neutral Applicant`
- Premiums are computed by sending a message to method `computeNeutralPremium` in that class

FLIC Mini Case Study (contd)

Slide 8.37

- However, there has not been enough time to change the whole system
 - The situation is therefore is as shown

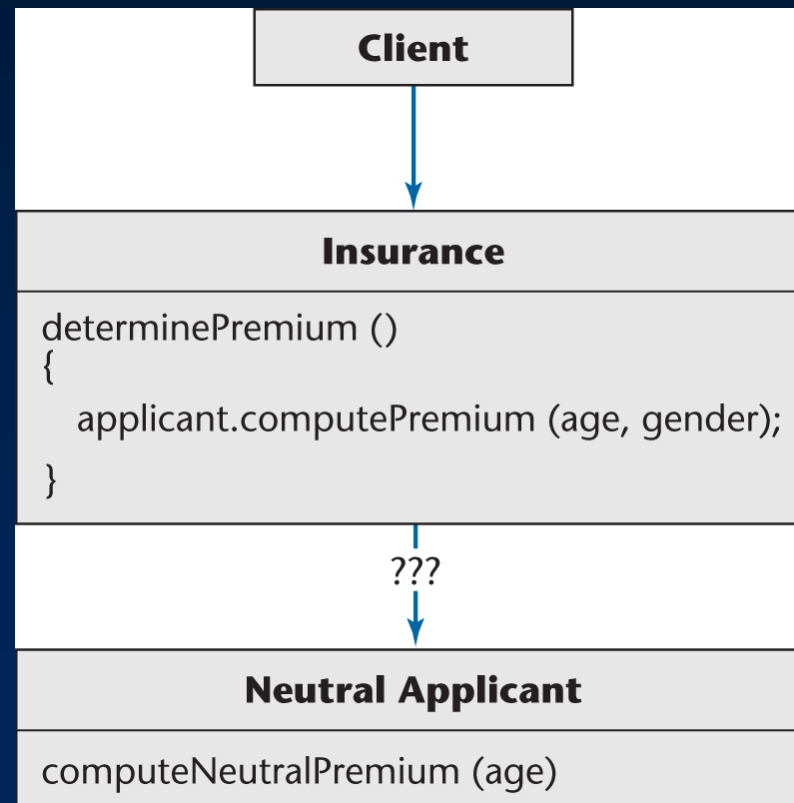


Figure 8.4

- There are three serious interfacing problems:
 - An `Insurance` object passes a message to an object of type (class) `Applicant`, instead of `Neutral Applicant`
 - The message is sent to method `computePremium` instead of method `computeNeutralPremium`
 - Parameters `age` and `gender` are passed, instead of just `age`
- The three question marks on the lower arrow represent these three interfacing problems

FLIC Mini Case Study (contd)

Slide 8.39

- To solve these problems
 - Interpose class **Wrapper**

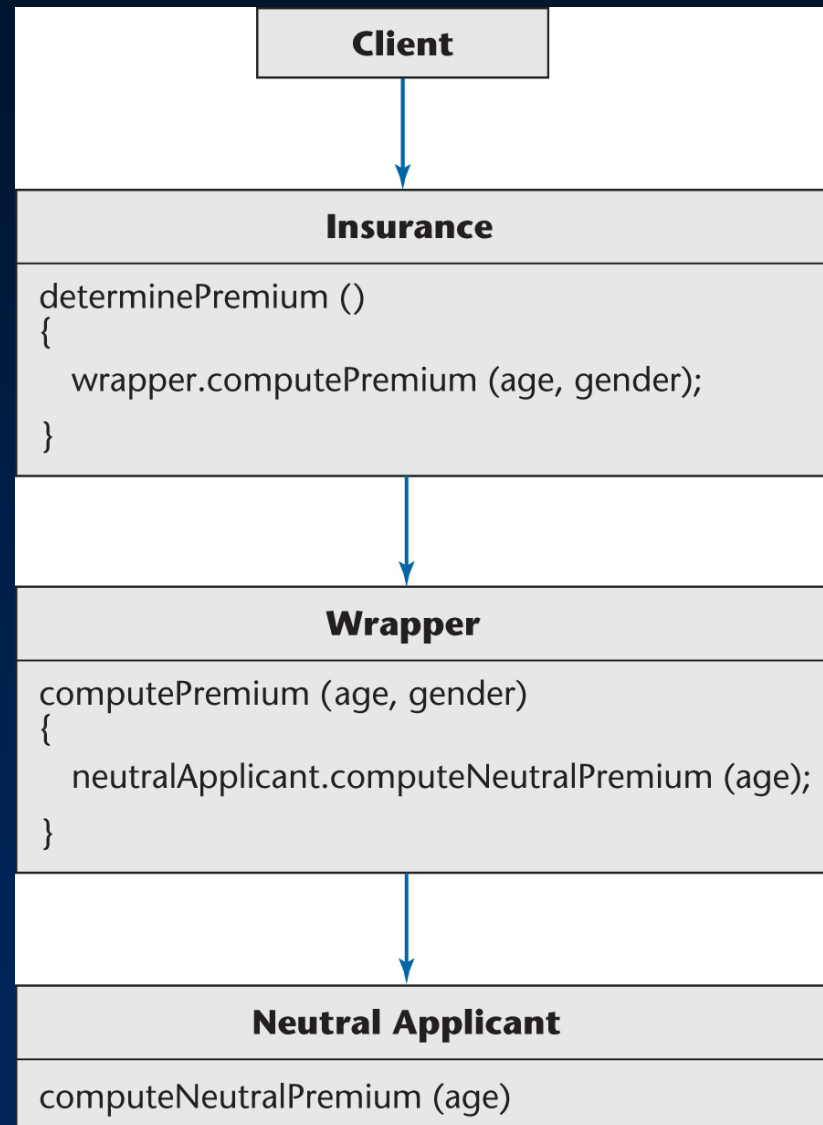


Figure 8.5

FLIC Mini Case Study (contd)

Slide 8.40

- An object of class **Insurance**
 - Sends the same message `computePremium`
 - Passing the same two parameters (`age` and `gender`)
- Now the message is sent to an object of type **Wrapper**
- This object then sends message `computeNeutralPremium` to an object of class **NeutralApplicant**
 - Passing only `age` as parameter
- The three interfacing problems have been solved

8.6.2 *Adapter* Design Pattern

Slide 8.41

- Generalizing the solution to the FLIC mini case study leads to the *Adapter* design pattern

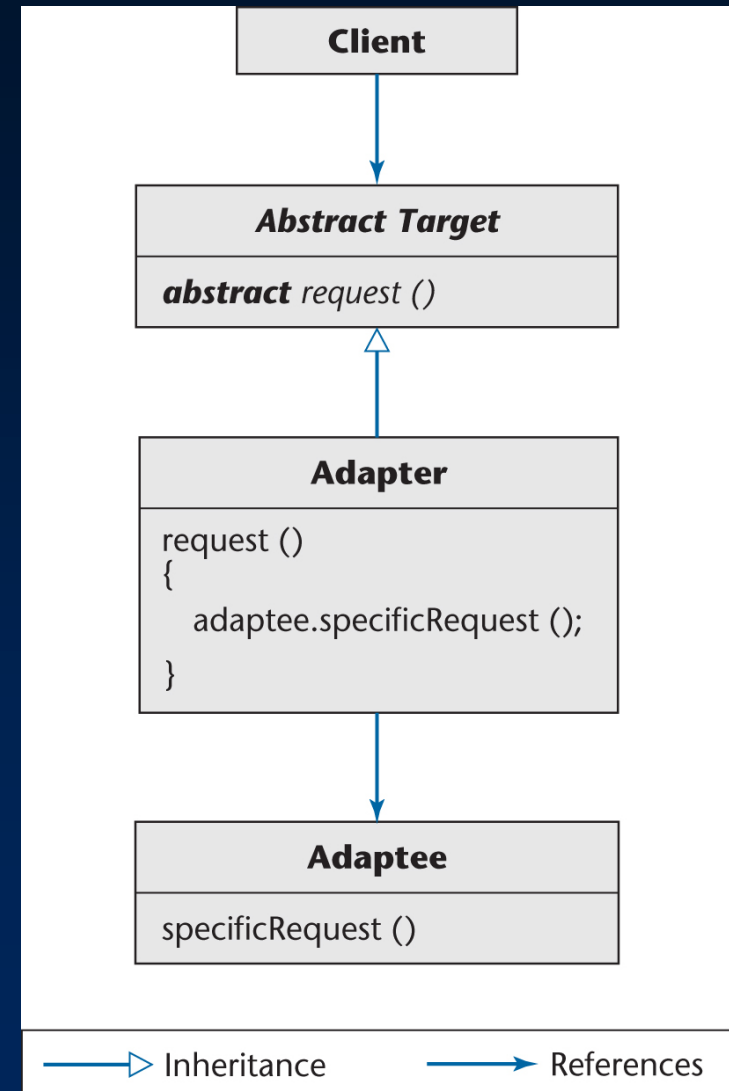


Figure 8.6

Adapter Design Pattern (contd)

Slide 8.42

- The names of abstract classes and their abstract (virtual) methods are in *Courier italics*
- Method *request* is defined as an abstract method of class *Abstract Target*
 - It is implemented in (concrete) class **Adapter** to send message *specificRequest* to an object of class **Adaptee**
- Class **Adapter** is a concrete subclass of abstract class *Abstract Target*
 - The open arrow denotes inheritance

- 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

8.6.3 *Bridge* Design Pattern

Slide 8.44

- 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

8.6.3 *Bridge* Design Pattern

Slide 8.45

- 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

- 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

Bridge Design Pattern (contd)

Slide 8.47

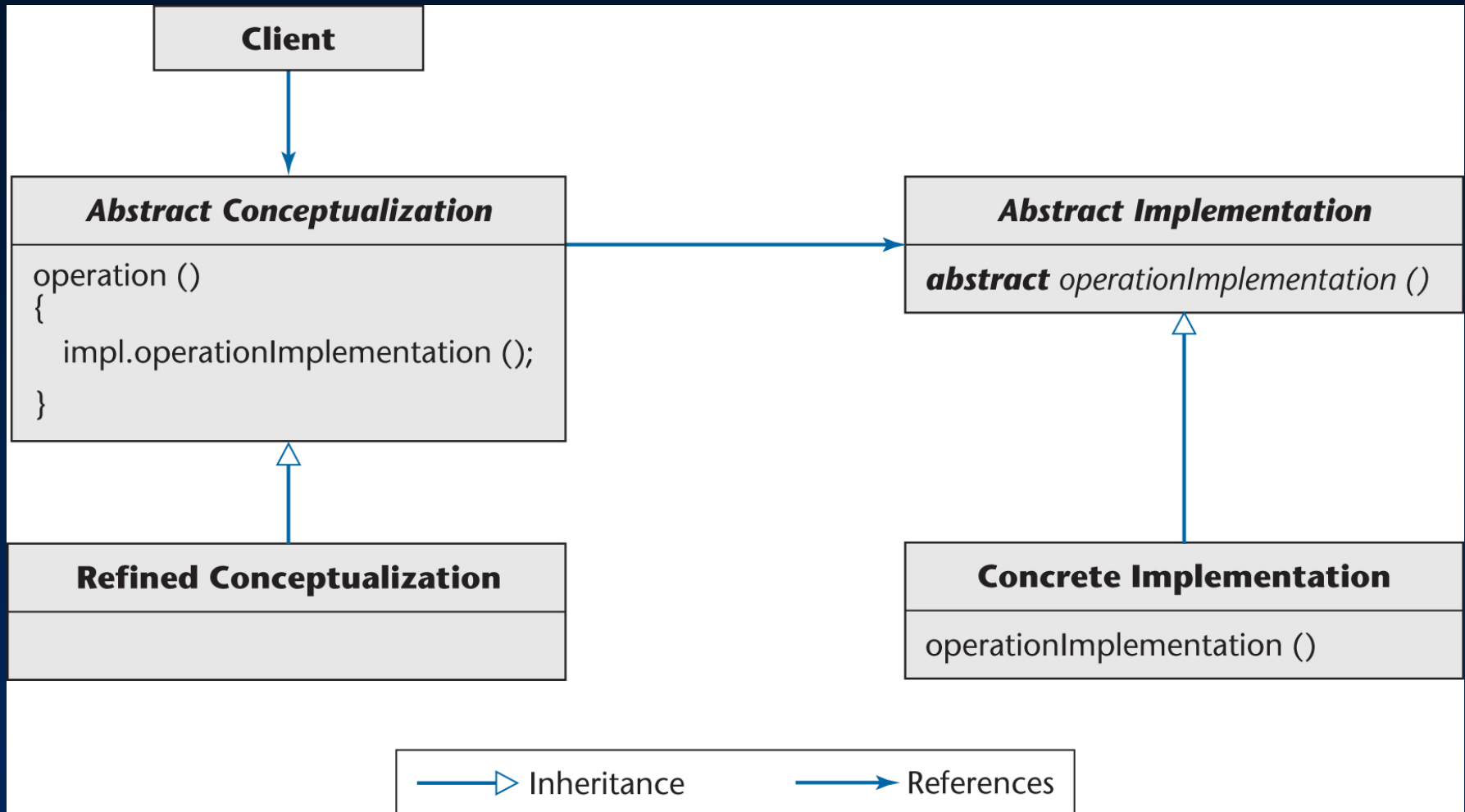


Figure 8.7

- The implementation-independent piece is in classes *Abstract Conceptualization* and *Refined Conceptualization*
- The implementation-dependent piece is in classes *Abstract Implementation* and *Concrete Implementation*

Bridge Design Pattern (contd)

Slide 8.49

- The *Bridge* design pattern can support multiple implementations

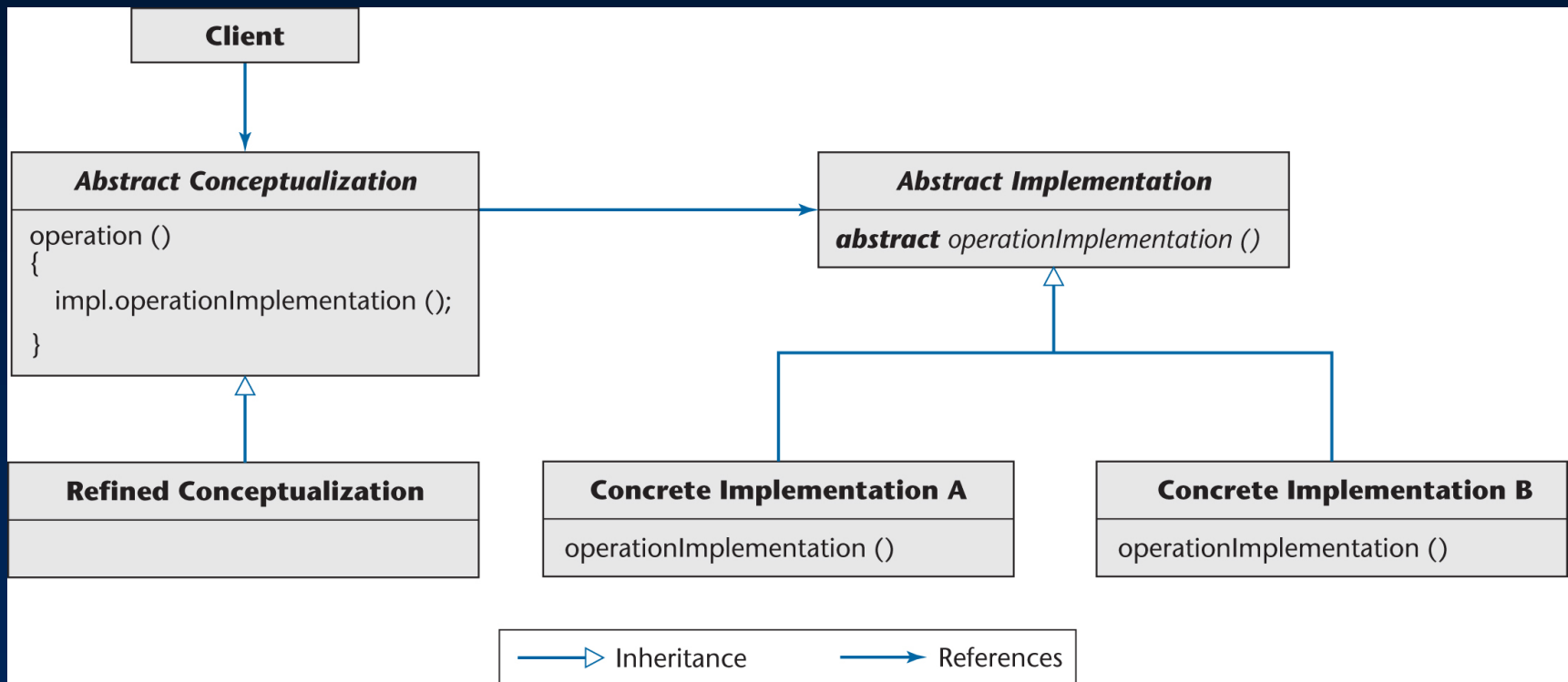


Figure 8.8

8.6.4 *Iterator* Design Pattern

Slide 8.50

- 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 (contd)

Slide 8.52

- ❑ Example of an iterator: television remote control
 - Key (Up or ▲) increases the channel number by one
 - Key (Down or ▼) decreases the channel number by one

- ❑ Keys increase or decrease the channel number without the viewer having to specify (or even having to know) the current channel number
 - Let alone the program being carried on that channel

- ❑ The device implements element traversal without exposing the implementation of the aggregate

Iterator Design Pattern (contd)

Slide 8.53

□ Iterator design pattern

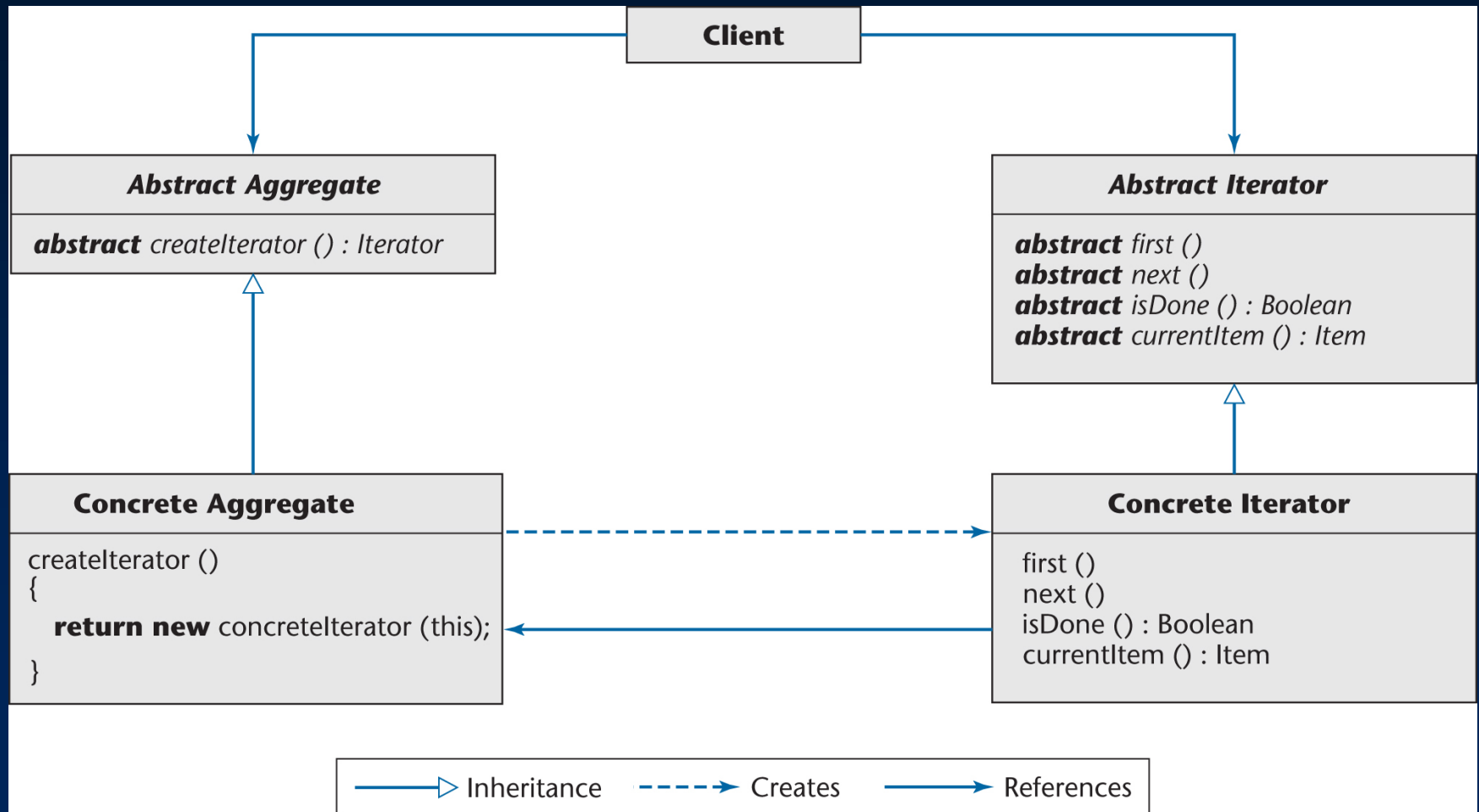


Figure 8.9

Iterator Design Pattern (contd)

Slide 8.54

- A `client` object deals with only
 - *Abstract Aggregate* and *Abstract Iterator* (essentially interfaces)
- The `client` object asks the *Abstract Aggregate* object to create an iterator for the `Concrete Aggregate` object
- It utilizes the returned `Concrete Iterator` to traverse the contents of the aggregate

Iterator Design Pattern (contd)

Slide 8.55

- The **Abstract Aggregate** object has an abstract method `createIterator` that returns an iterator to the `client` object
- The **Abstract Iterator** interface defines the basic four abstract traversal operations
 - `first`, `next`, `isDone`, `currentItem`
- Implementation of these five methods is achieved at the next level of abstraction, in
 - **Concrete Aggregate** (`createIterator`), and in
 - **Concrete Iterator** (`first`, `next`, `isDone`, `currentItem`)

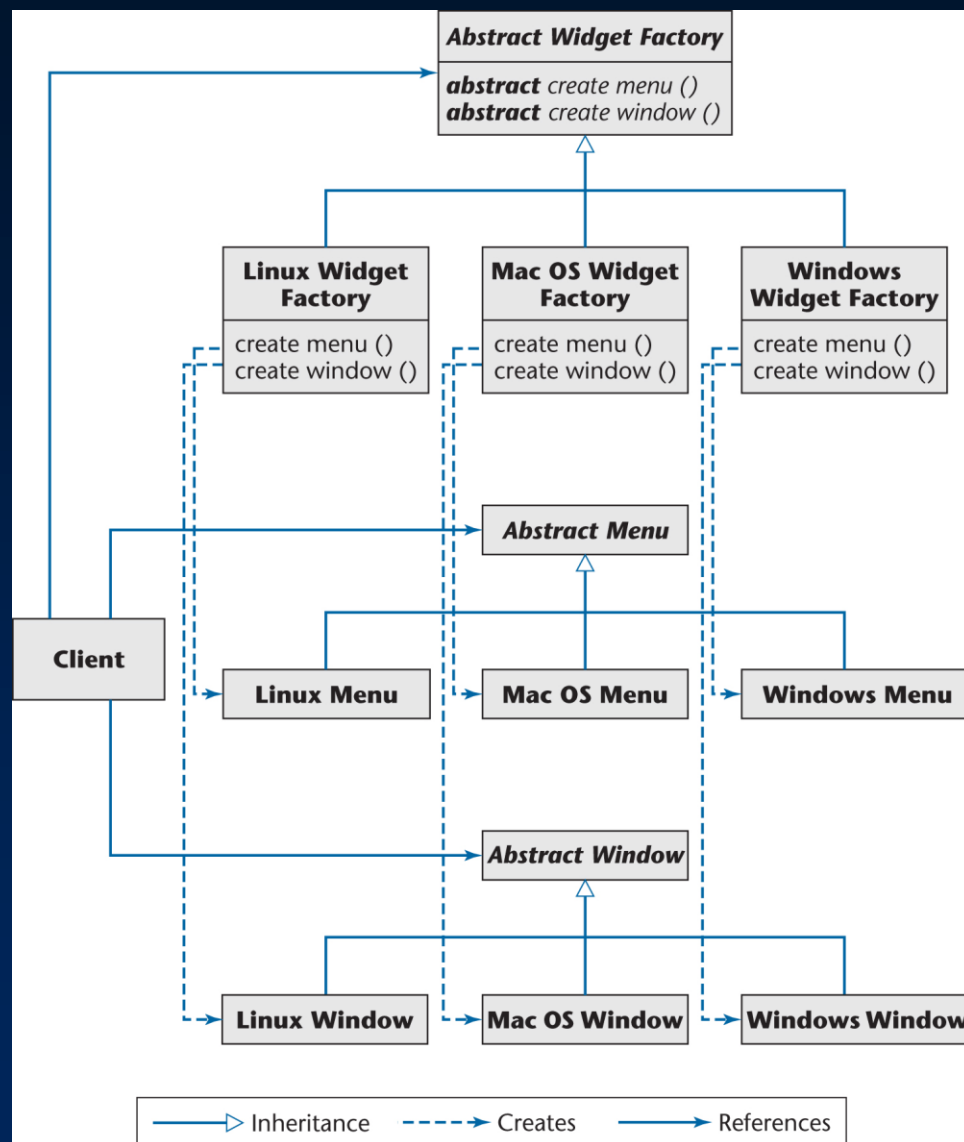
- Implementation details of the elements are hidden from the iterator itself
 - We can use an iterator to process every element in a collection,
 - Independently of the implementation of the container of the elements

- *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**

8.6.5 Abstract Factory Design Pattern

Slide 8.58

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



- The three interfaces between the `client` and the generator are abstract
- Each concrete widget factory generates widgets to be utilized by the application program
 - Those widgets will run under the operating system specific to that concrete widget factory
- The application program is hence uncoupled from the specific operating system

Abstract Factory Design Pattern

Slide 8.60

- 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

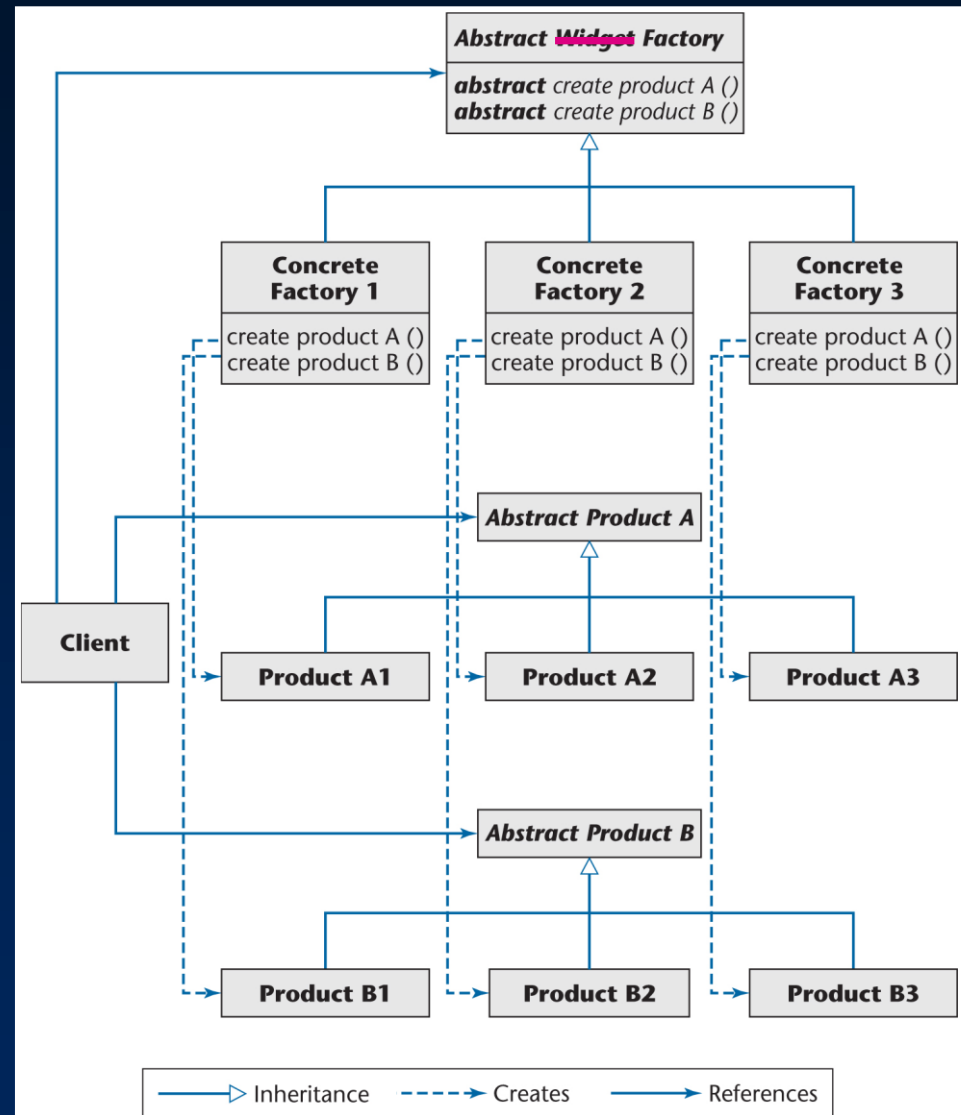


Figure 8.11

8.7 Categories of Design Patterns

Slide 8.61

- The 23 “Gang of Four” design patterns are grouped into three categories:

Creational patterns

<i>Abstract factory</i>	Creates an instance of several families of classes (Section 8.6.5)
<i>Builder</i>	Allows the same construction process to create different representations
<i>Factory method</i>	Creates an instance of several possible derived classes
<i>Prototype</i>	A class to be cloned
<i>Singleton</i>	Restricts instantiation of a class to a single instance

Figure 8.12(a)

- Creational design patterns solve design problems by creating objects
 - Example: *Abstract Factory* pattern

Categories of Design Patterns (contd)

Slide 8.62

Structural patterns

<i>Adapter</i>	Matches interfaces of different classes (Section 8.6.2)
<i>Bridge</i>	Decouples an abstraction from its implementation (Section 8.6.3)
<i>Composite</i>	A class that is a composition of similar classes
<i>Decorator</i>	Allows additional behavior to be dynamically added to a class
<i>Façade</i>	A single class that provides a simplified interface
<i>Flyweight</i>	Uses sharing to support large numbers of fine-grained classes efficiently
<i>Proxy</i>	A class functioning as an interface

Figure 8.12(b)

- Structural design patterns solve design problems by identifying a simple way to realize relationships between entities
 - Examples: *Adapter* pattern, *Bridge* pattern

Categories of Design Patterns (contd)

Slide 8.63

Behavioral patterns

<i>Chain-of-responsibility</i>	A way of processing a request by a chain of classes
<i>Command</i>	Encapsulates an action within a class
<i>Interpreter</i>	A way to implement specialized language elements
<i>Iterator</i>	Sequentially access the elements of a collection (Section 8.6.4)
<i>Mediator</i>	Provides a unified interface to a set of interfaces
<i>Memento</i>	Captures and restores an object's internal state
<i>Observer</i>	Allows the observation of the state of an object at run time
<i>State</i>	Allows an object to partially change its type at run time
<i>Strategy</i>	Allows an algorithm to be dynamically selected at run time
<i>Template method</i>	Defers implementations of an algorithm to its subclasses
<i>Visitor</i>	Adds new operations to a class without changing it

Figure 8.12(c)

- Behavioral design patterns solve design problems by identifying common communication patterns
 - Example: *Iterator* pattern

8.8 Strengths and Weaknesses of Design Patterns

Slide 8.64

□ Strengths

- Design patterns promote reuse 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 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

- 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

8.9 Reuse and the World Wide Web

Slide 8.69

- 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

Problems with Reusing Code from the Web

Slide 8.70

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

- 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

- 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

8.10 Reuse and Postdelivery Maintenance

Slide 8.73

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

Results

Slide 8.74

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

Figure 8.13

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

- 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

8.11.1 Hardware Incompatibilities

Slide 8.76

- Storage media incompatibilities
 - Example: Zip vs. DAT

- Character code incompatibilities
 - Example: EBCDIC vs. ASCII

- Word size

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

8.11.2 Operating System Incompatibilities

Slide 8.78

- Job control languages (JCL) can be vastly different
 - Syntactic differences

- Virtual memory vs. overlays

8.11.3 Numerical Software Incompatibilities

Slide 8.79

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

8.11.4 Compiler Incompatibilities

Slide 8.80

- 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)

- 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

8.12 Why Portability?

Slide 8.82

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

Why Portability? (contd)

Slide 8.83

- ❑ 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

8.13 Techniques for Achieving Portability

Slide 8.84

- Obvious technique
 - Use standard constructs of a popular high-level language

- But how is a portable operating system to be written?

8.13.1 Portable System Software

Slide 8.85

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

- Utilize levels of abstraction
 - Example: Graphical display routines

8.13.2 Portable Application Software

Slide 8.86

- 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
 - This may be nontrivial for complex database models

Strengths of and Impediments to Reuse and Portability

Slide 8.88

Strengths	Impediments
Reuse Shorter development time (Section 8.1) Lower development cost (Section 8.1) Higher-quality software (Section 8.1) Shorter maintenance time (Section 8.10) Lower maintenance cost (Section 8.10)	NIH syndrome (Section 8.2) Potential quality issues (Section 8.2) Retrieval issues (Section 8.2) Cost of making a component reusable (opportunistic reuse) (Section 8.2) 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) More copies of COTS software can be sold (Section 8.12)	Potential incompatibilities: 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