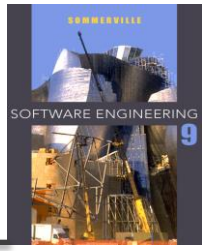
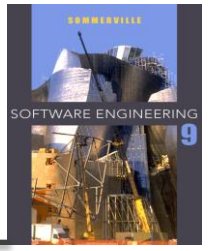
Coding & Refactoring and Design Patterns



Dr. Mohammad Ahmad

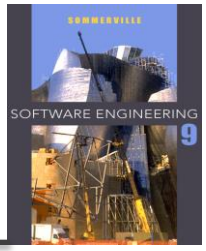
Lecture 11

The Open-Closed Principle



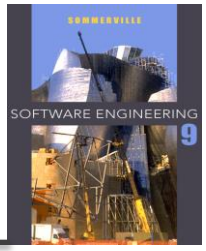
- Software entities should be *open for extension* but *closed for modifications*.
  - Design classes and packages so their functionality can be extended without modifying the source code

Good Signs of OO Thinking



- ✧ Short methods
  - Simple method logic
- ✧ Few instance variables
- ✧ Clear object responsibilities
  - State the purpose of the class in one sentence
  - No super-intelligent objects
  - No manager objects

Some Principles



✧ The Dependency Inversion Principle

- Depend on abstractions, not concrete implementations
  - Write to an interface, not a class

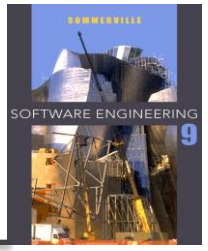
✧ The Interface Segregation Principle

- Many small interfaces are better than one “fat” one

✧ The Acyclic Dependencies Principle

- Dependencies between package must not form cycles.
  - Break cycles by forming new packages

Packages, Modules and other



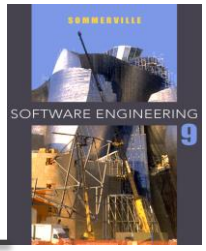
✧ The Common Closure Principle

- Classes that change together, belong together
  - Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed.

✧ The Common Reuse Principle

- Classes that aren’t reused together don’t belong together
  - The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

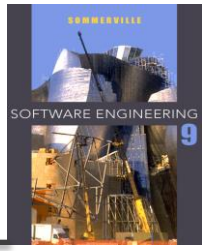
What is Refactoring?



- The process of *changing a software system* in such a way that it *does not alter the external behavior* of the code, yet *improves its internal structure*.

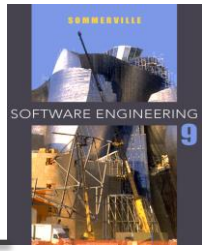
  - Fowler, et al., Refactoring, 1999.

Typical Refactorings



Class Refactorings	Method Refactorings	Attribute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Why Refactor?

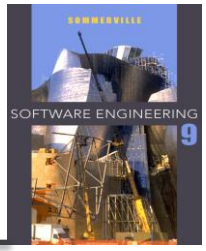


"Grow, don't build software"

- Fred Brooks

- The reality:
  - Extremely difficult to get the design "right" the first time
  - Hard to fully understand the problem domain
  - Hard to understand user requirements, even if the user does!
  - Hard to know how the system will evolve in five years
  - Original design is often inadequate
  - System becomes brittle over time, and more difficult to change
- Refactoring helps you to
  - Manipulate code in a safe environment (behavior preserving)
  - Recreate a situation where evolution is possible
  - Understand existing code

Rename Method — manual steps

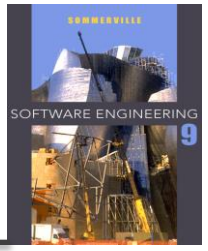


✧ Do it yourself approach:

- Check that no method with the new name already exists in any subclass or superclass.
- Browse all the implementers (method definitions)
- Browse all the senders (method invocations)
- Edit and rename all implementers
- Edit and rename all senders
- Remove all implementers
- Test

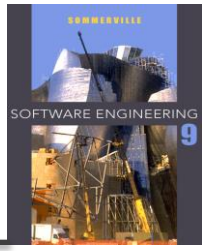
✧ Automated refactoring is better !

Rename Method



- ✧ Rename Method (method, new name)
- ✧ Preconditions
  - No method with the new name already exists in any subclass or superclass.
  - No methods with same signature as method outside the inheritance hierarchy of method
- ✧ PostConditions
  - method has new name
  - relevant methods in the inheritance hierarchy have new name
  - invocations of changed method are updated to new name
- ✧ Other Considerations
  - Typed/Dynamically Typed Languages => Scope of the renaming

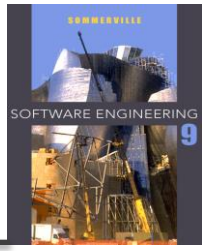
The Law of Demeter



- ***"Do not talk to strangers"***
  - You should only send messages to:
    - an argument passed to you
    - an object you create
    - self, super
    - your class

- Don't send messages to objects returned from other message sends

Code Smells

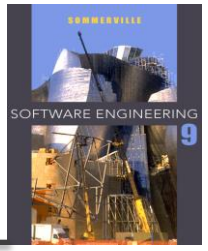


"If it stinks, change it"

- *Grandma Beck*

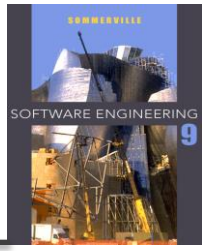
- Duplicated Code
  - Missing inheritance or delegation
- Long Method
  - Inadequate decomposition
- Large Class / God Class
  - —Too many responsibilities
- Long Parameter List
  - —Object is missing
- Type Tests
  - —Missing polymorphism
- Shotgun Surgery
  - —Small changes affect too many objects

Code Smells



- Feature Envy
  - Method needing too much information from another object
- Data Clumps
  - Data always used together (x,y -> point)
- Parallel Inheritance Hierarchies
  - Changes in one hierarchy require change in another hierarchy
- Lazy Class
  - Does too little
- Middle Man
  - Class with too many delegating methods
- Temporary Field
  - Attributes only used partially under certain circumstances
- Data Classes
  - Only accessors

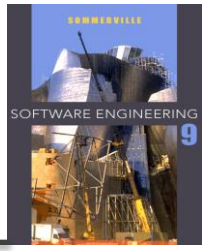
Curing Long Methods



- ✧ Long methods
  - Decompose into smaller methods
  - Self sends should *read like a script*
  - Comments are good delimiters
  - A method is the *smallest unit of overriding*

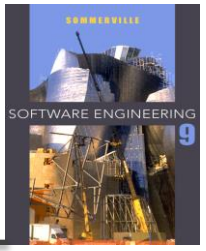
```
self setUp; run; tearDown.
```

Curing Duplicated Code



- In the same class
  - Extract Method
- Between two sibling subclasses
  - Extract Method
  - Push identical methods up to common superclass
  - Form Template Method
- Between unrelated class
  - Create common superclass
  - Move to Component
  - Extract Component (e.g., Strategy)

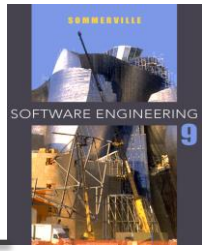
Curing God Class



✧ God Class

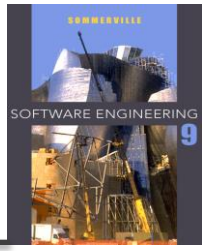
- Incrementally redistribute responsibilities to existing (or extracted) collaborating classes
- Find logical sub-components
  - Set of related working methods/instance variables
- Move methods and instance variables into components
- Extract component
- Extract Subclass
  - If not using all the instance variables

Curing Type Tests



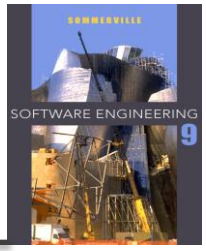
- ✧ Missing Polymorphism
  - Tell, don’t ask!
  - Shift case bodies to (new) methods of object being tested
  - Self type checks:
    - Introduce hook methods and new subclasses
  - Client type checks
    - Introduce “tell” method into client hierarchy
  - Possibly introduce State / Strategy or Null Object Design Patterns

Design patterns



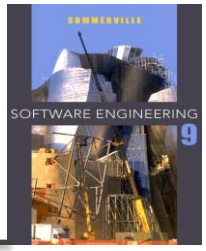
- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

Pattern elements



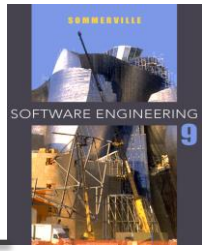
- Name
  - A meaningful pattern identifier.
- Problem description.
- Solution description.
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- Consequences
  - The results and trade-offs of applying the pattern.

Design Patterns



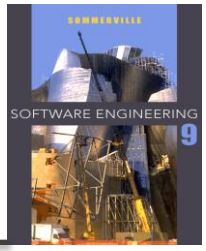
- Design Patterns document *recurrent solutions* to *design problems*
  - They have *names*
    - Composite, Visitor, Observer...
  - They are not components!
  - Design Patterns entail *tradeoffs*
  - Will be implemented in different ways in different contexts

Why Design Patterns?



- ✧ Smart
  - Elegant solutions that a novice would not think of
- ✧ Generic
  - Independent of specific system type, language
- ✧ Well-proven
  - Successfully tested in several systems
- ✧ Simple
  - Combine them for more complex solutions

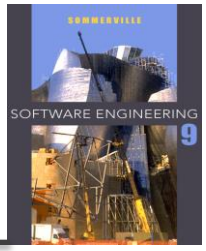
Alert!!! Patterns are invading!



- Design Patterns are not “good” just because they a patterns

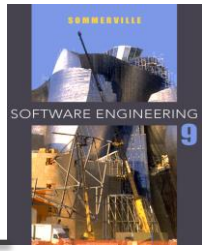
  - It is just as important to understand when *not* to use a Design Pattern
  - Every Design Pattern has tradeoffs
  - Most Design Patterns will make your design *more complicated*
    - More classes, more indirections, more messages
  - Don’t use Design Patterns unless you really need them!

About Pattern Implementation



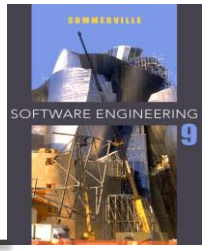
- Do not confuse *structure* and *intent!*
  - Design Patterns document a *possible* implementation
    - Not a definitive one
  - Design Patterns are about *intent* and *tradeoffs*

Common Design Patterns



Pattern	***Intent***
Adapter	Convert the interface of a class into another interface clients expect.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Composite	Compose objects into part-whole hierarchies so that clients can treat individual objects and compositions uniformly.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps so they can be redefined by subclasses.

The Singleton Pattern



Intent:

✧ Ensure that a class has only one instance, and provide a global point of access to it

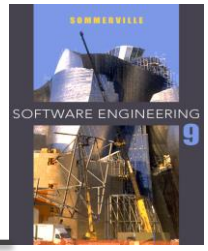
Problem:

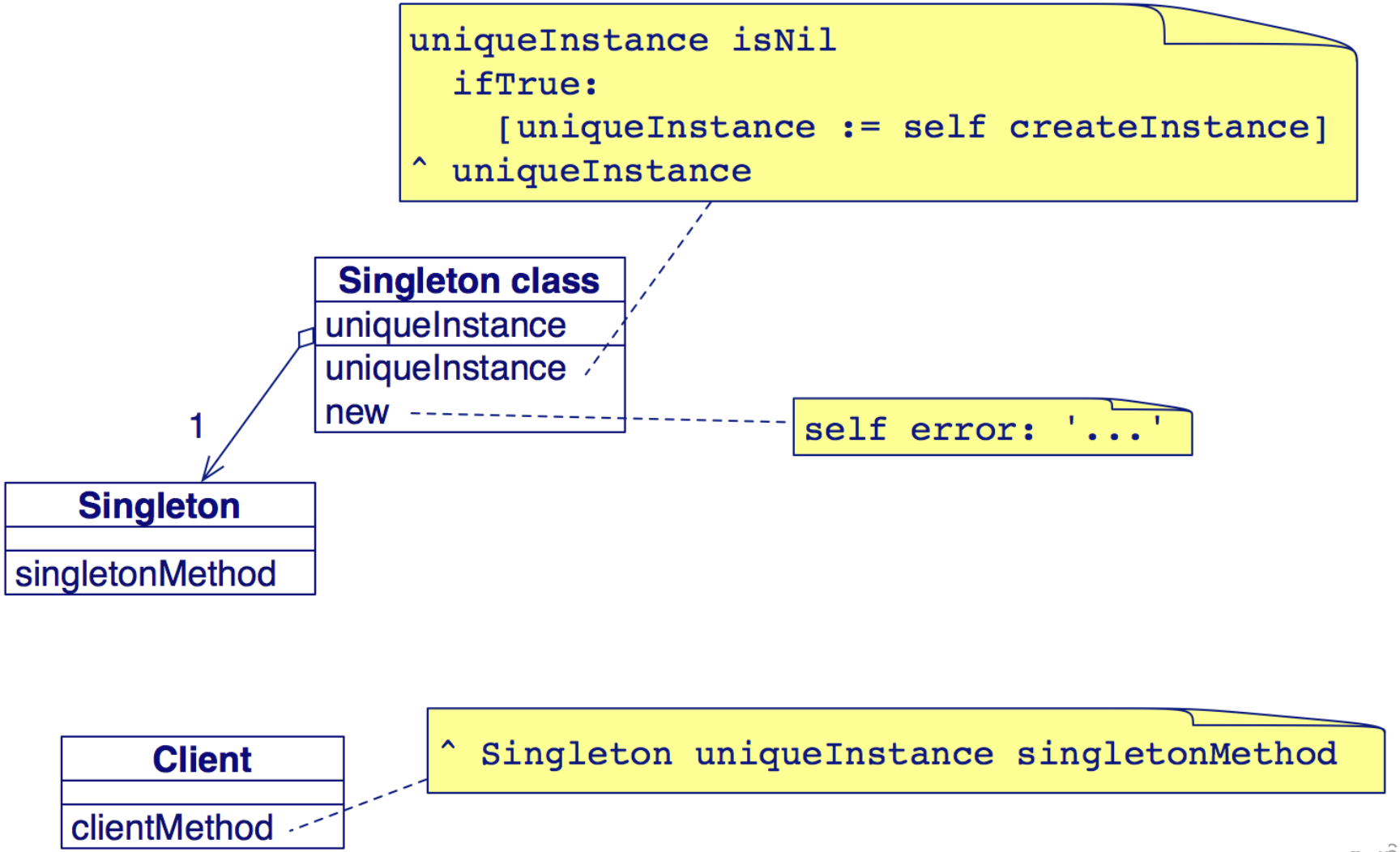
✧ We want a class with a unique instance.

Solution:

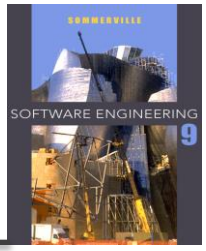
✧ Give the class the responsibility to initialize and provide access to the unique instance. Forbid creation of new instances.

Singleton Structure



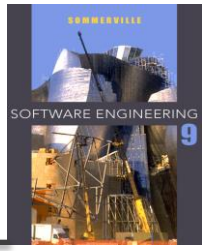


Implementation Issues



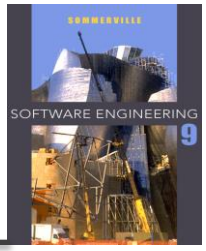
- ✧ Class variable
  - One singleton for a complete hierarchy
- ✧ Class instance variable
  - One singleton per class

Implementation Issues



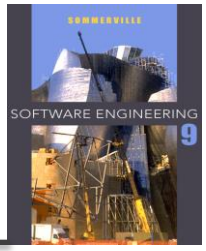
- ✧ Singletons may be accessed via a global variable
- ✧ Global Variable vs. Class Method Access
  - Global Variable Access is dangerous: if we reassign Notifier we lose all references to the current window.
  - Class Method Access is better because it provides a single access point.

The Observer pattern



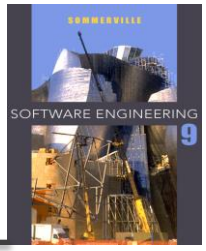
- Name
  - Observer.
- Description
  - Separates the display of object state from the object itself.
- Problem description
  - Used when multiple displays of state are needed.
- Solution description
  - See slide with UML description.
- Consequences
  - Optimisations to enhance display performance are impractical.

The Observer pattern (1)



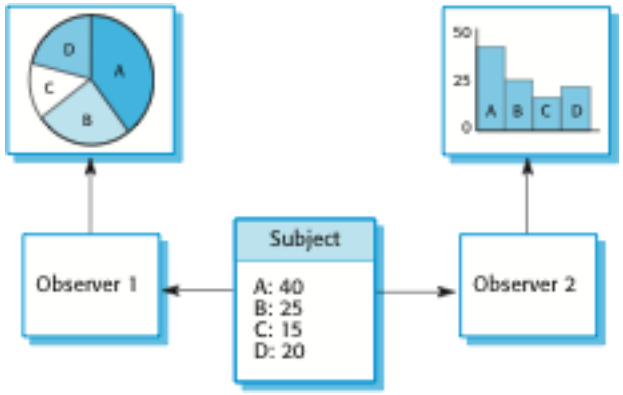
Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

The Observer pattern (2)

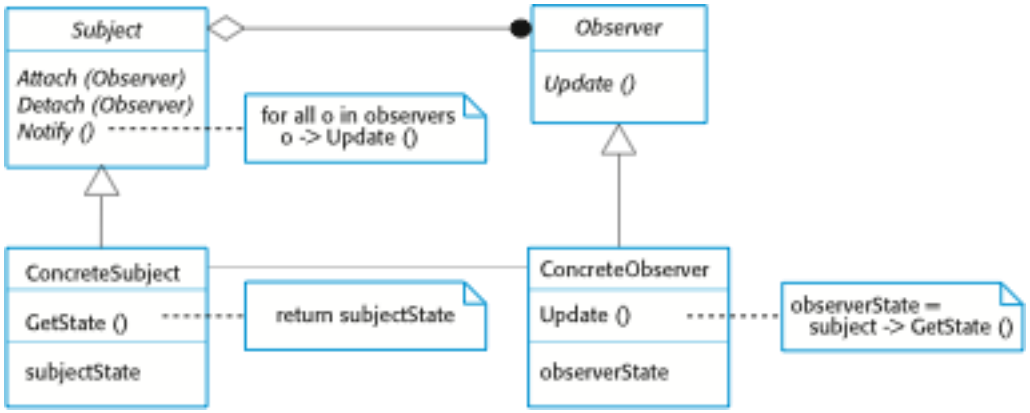


Pattern name	Observer
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

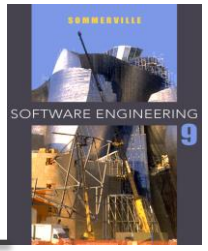
Multiple displays using the Observer pattern



A UML model of the Observer pattern

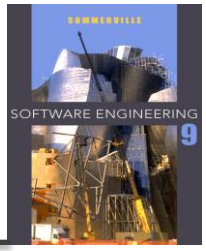


Design problems



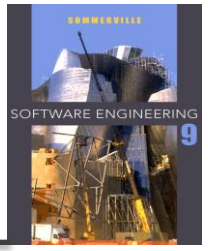
- To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
  - Tell several objects that the state of some other object has changed (Observer pattern).
  - Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).
  - Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).
  - Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

Implementation issues



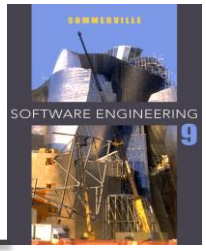
- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - Reuse Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
  - Configuration management During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
  - Host-target development Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

Reuse



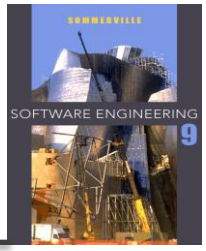
- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

Reuse levels



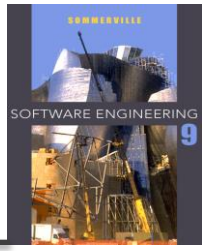
- The abstraction level
  - At this level, you don’t reuse software directly but use knowledge of successful abstractions in the design of your software.
- The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Reuse costs



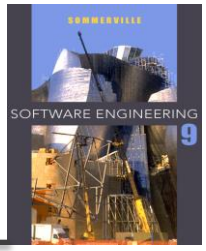
- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Host-target development



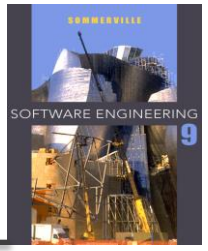
- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Integrated development environments (IDEs)



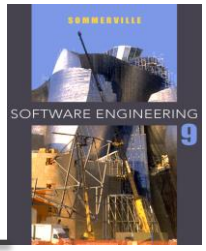
- Software development tools are often grouped to create an integrated development environment (IDE).
- An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

Component/system deployment factors



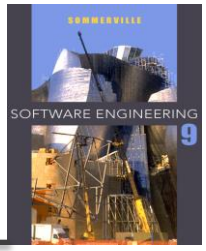
- If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
- If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.

Open source development



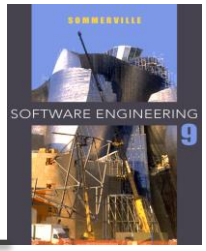
- Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

Open source systems



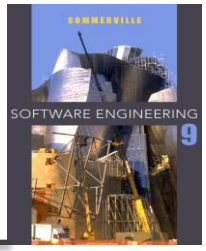
- The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- Other important open source products are Java, the Apache web server and the mySQL database management system.

Open source issues



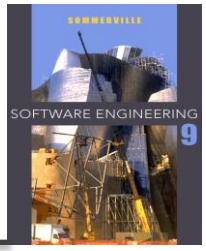
- Should the product that is being developed make use of open source components?
- Should an open source approach be used for the software's development?

Open source business



- More and more product companies are using an open source approach to development.
- Their business model is not reliant on selling a software product but on selling support for that product.
- They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

Key points



- When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.