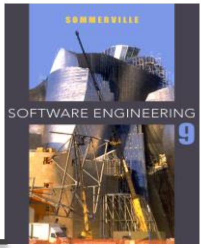
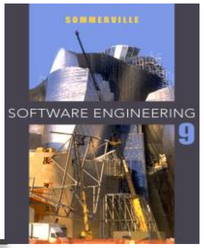


Software Design

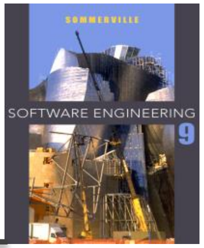
K.Sridhar Patnaik

Dept of CSE,BIT Mesra

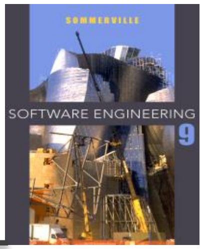
Software Design

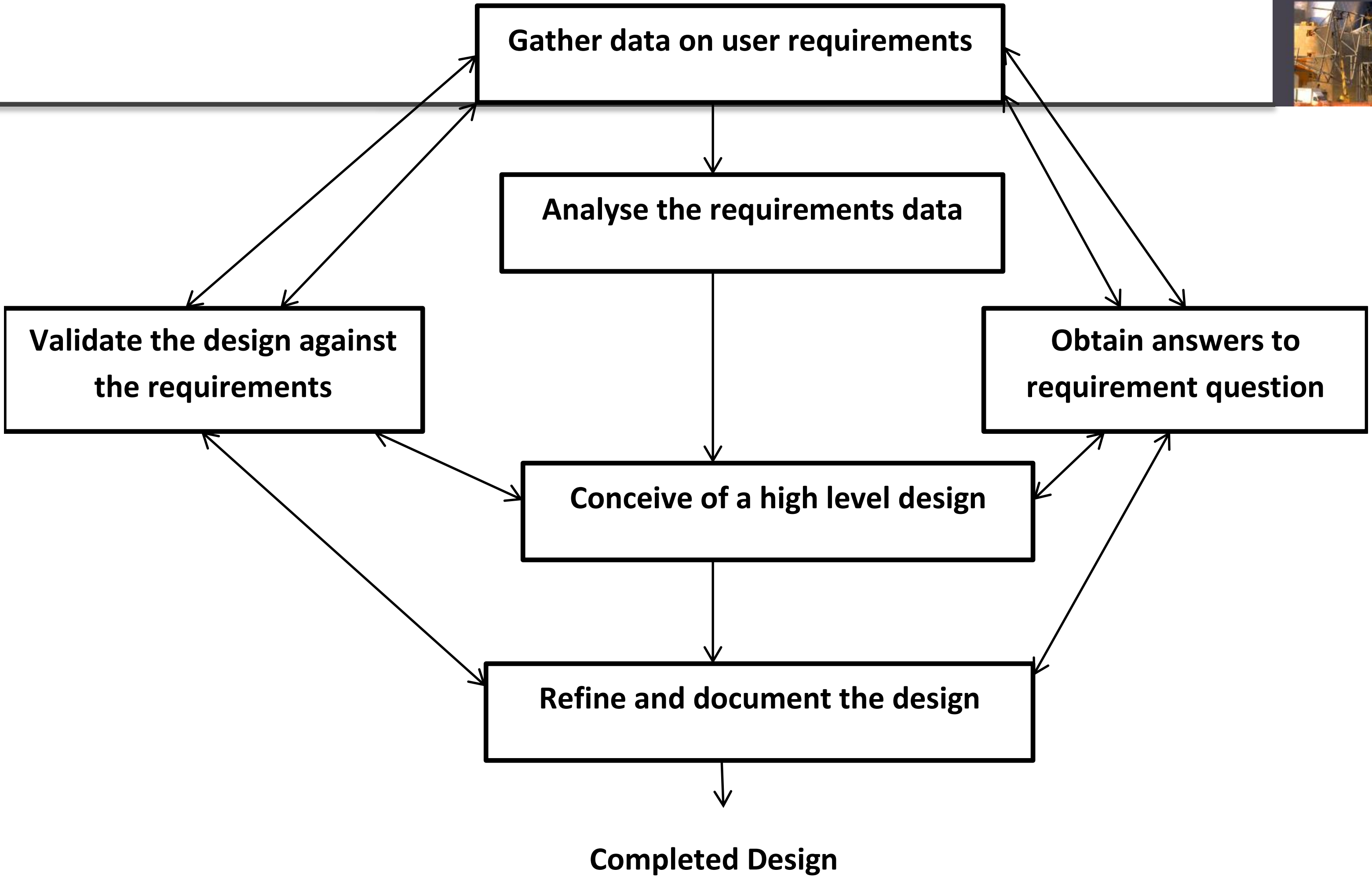


- ✧ Bridge between specification and coding
- ✧ More creative process than analysis, because it deals with the development of the actual mechanism for a new workable system.
- ✧ Definition: The highly significant phase in the s/w development where the designer plans “how” a s/w system should be produced in order to make it functional, reliable and reasonably easy to understand ,modify and maintain.
- ✧ SRS tells “what” a system does which is the input to the design process,which tells us “how” a s/w system works.



✧ The purpose of the design process is to produce a solution to a problem given in SRS document.

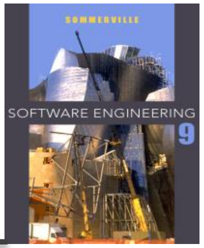




Gather data on user requirements
Analyse the requirements data
Validate the design against the requirements
Obtain answers to requirement question
Conceive of a high level design
Refine and document the design
Completed Design

DESIGN FRAMEWORK

Design Concepts



Abstraction

Architecture

Refinement

Modularity

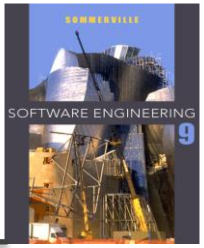
Patterns

Information Hiding

Functional independence

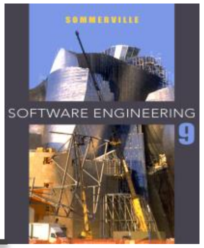
Refactoring

Design Classes



✧ Abstraction:(Data and Procedure)

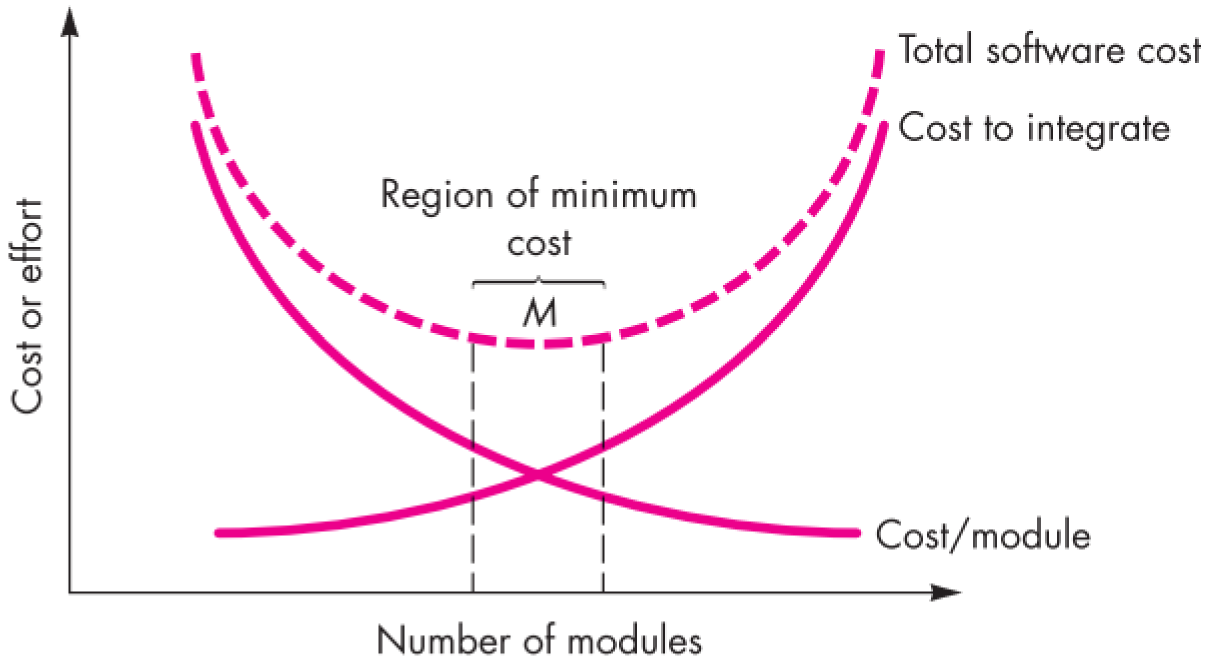
When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

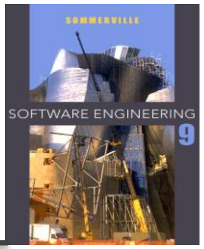


- **Architecture:**
- Architectural design can be represented by one or more of a number of models:
- Structural models, Framework models, Dynamic models,

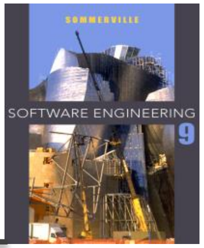
  Process models and Functional models.

✧ Modularity:


Total software cost
Cost to integrate
Region of minimum cost
M
Cost or effort
Cost/module
Number of modules



- ✧ **Functional Independence(FI)**: achieved by developing modules with “single minded” function and an “aversion” to excessive interaction with other modules. We wanted to design s/w so that each module addresses a specific subfunction of requirements and has simple interface when viewed from other parts of the program structure.
- ✧ Why independence is imp?
- ✧ s/w with effective modularity,i.e independent modules ,is easier to develop because function may be compartmentalized and interfaces are simplified.
- ✧ Independent modules are easier to maintain(and test) because secondary effects caused by design or code modification are limited,error propagation is reduced, and reusable modules are possible.
- ✧ FI is key to good design and design is the key to s/w quality.

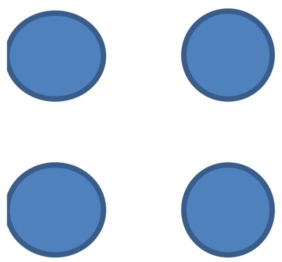


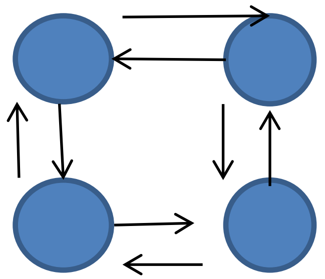
✧ FI is measured using two qualitative criteria:

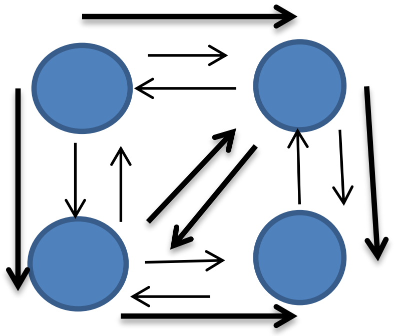
Cohesion and Coupling

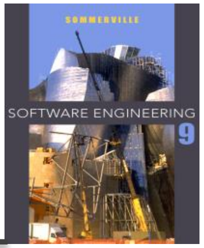
Coupling: *measure of the relative interdependence among modules.*

Cohesion: *measure of the relative functional strength of a module.*


No Coupling


Loosely coupled


Highly coupled



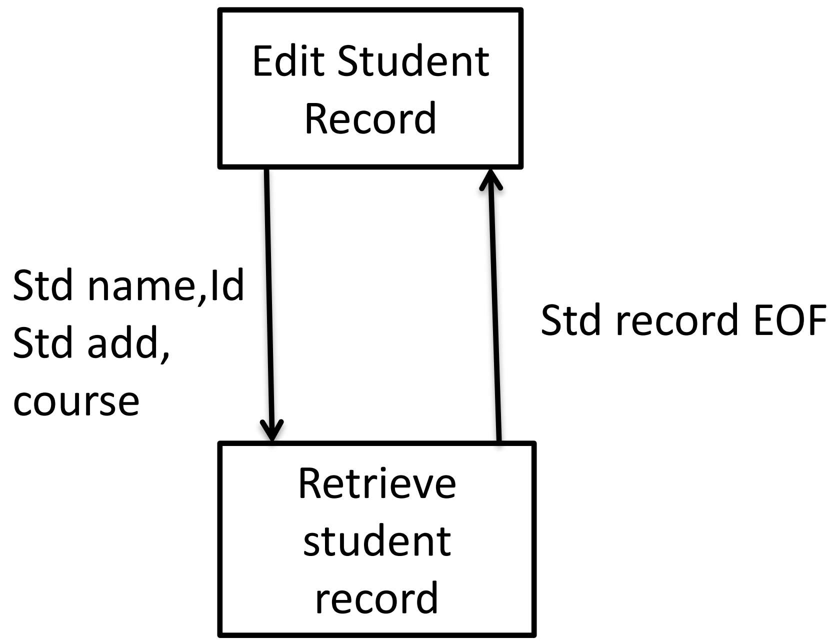
- A good design will have low coupling and high cohesion. Coupling increases as the number of calls between modules increases, or the amount of shared data increases
- Design with high coupling will have more errors.
- Loose coupling minimizes the interdependence amongst the modules and can be achieved by :

A)Controlling the number of parameters passed amongst modules.

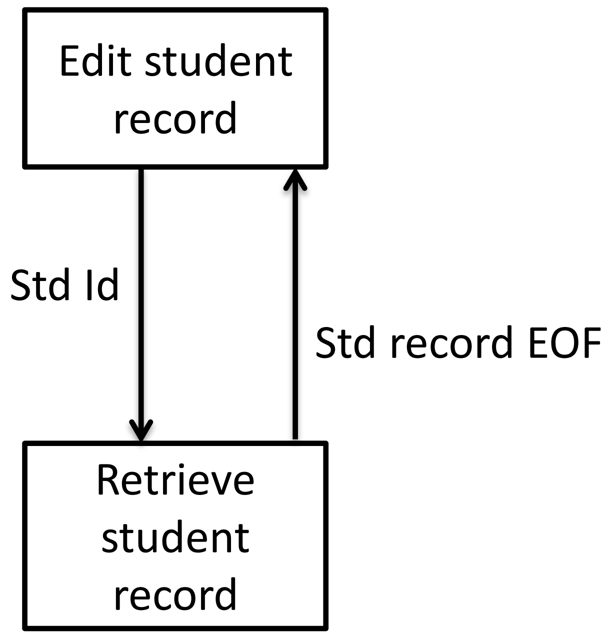
B)Avoid passing undesired data to calling module.

C)Maintain parent/child relationship between calling and called modules

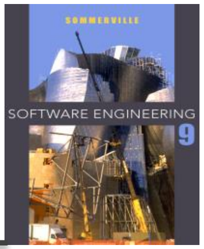
D)Pass data, not the control information



Poor Design:Tight coupling



Good Design: Loose coupling



Types of Coupling

Data Coupling

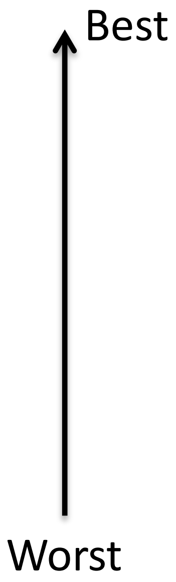
Stamp Coupling

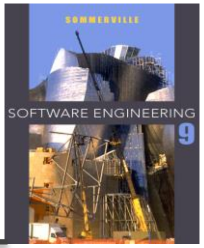
Control Coupling

External Coupling

Common Coupling

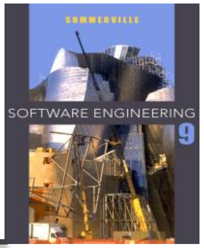
Content Coupling





- **Data Coupling**:The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicate by only passing of data. Other than communicating through data, the two modules are independent. A good strategy is to ensure that no module communication contains “tramp data”(std name, add, course are examples of tramp data)

- **Stamp Coupling:** Stamp coupling occurs between module A and B when complete data structure is passed from one module to another. Since not all data making up the structure are usually necessary in communication between the modules, stamp coupling typically involves **tramp data**. If one procedure only needs a part of a data structure, calling module should pass just that part ,not the complete data structure.

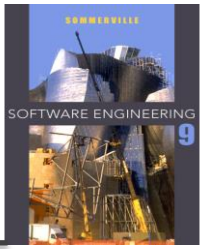


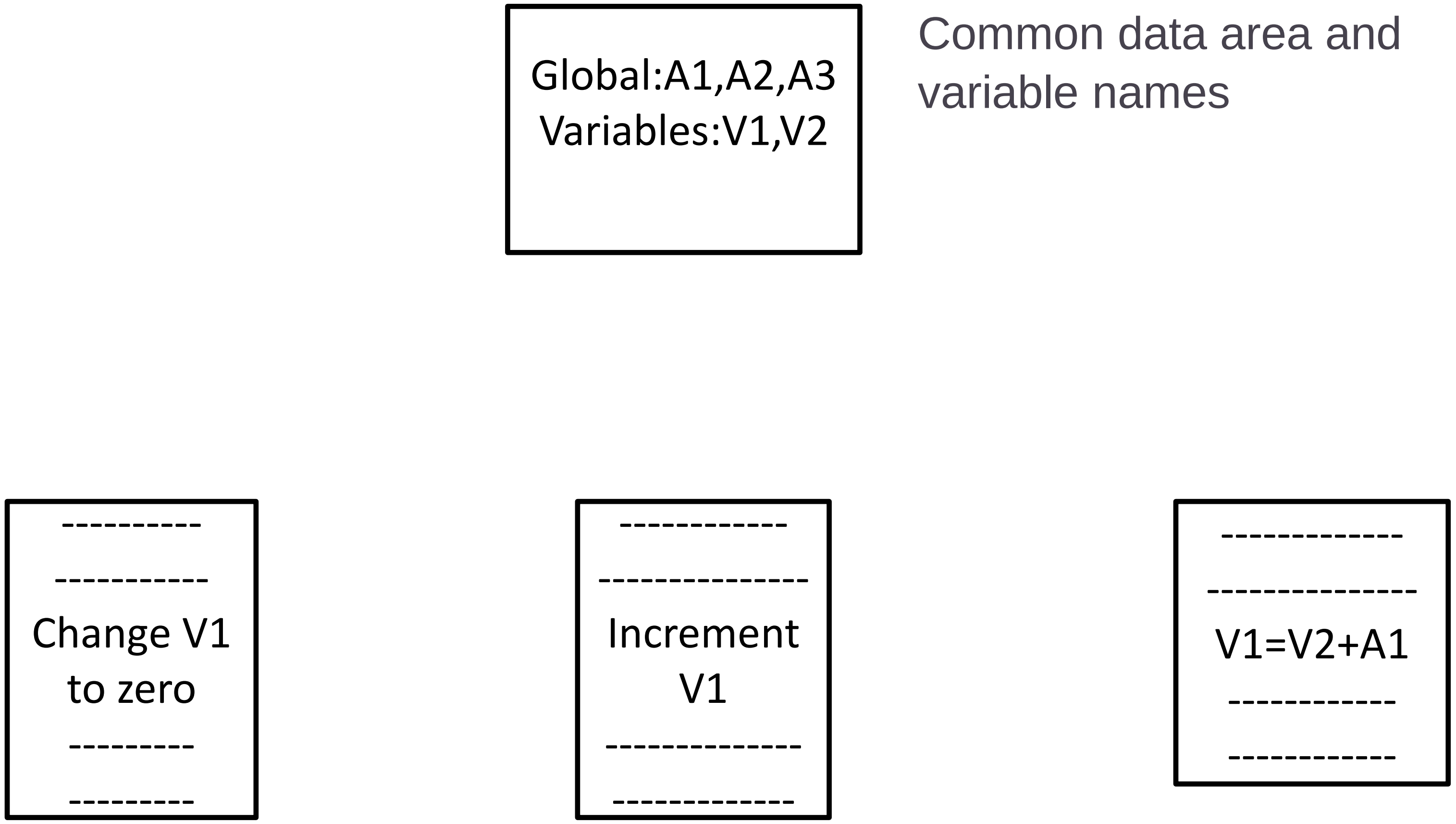
Control Coupling: Module A and B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.

External Coupling: A form of coupling in which a module has a dependency to other module, external to the software being developed or to a particular type of h/w. This is basically related to the communication to external tools and devices.

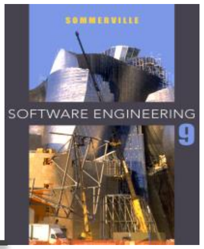
Common Coupling: With common coupling module A and B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of change.

With common coupling;it can be difficult to determine which module is responsible for having set a variable to a particular value.





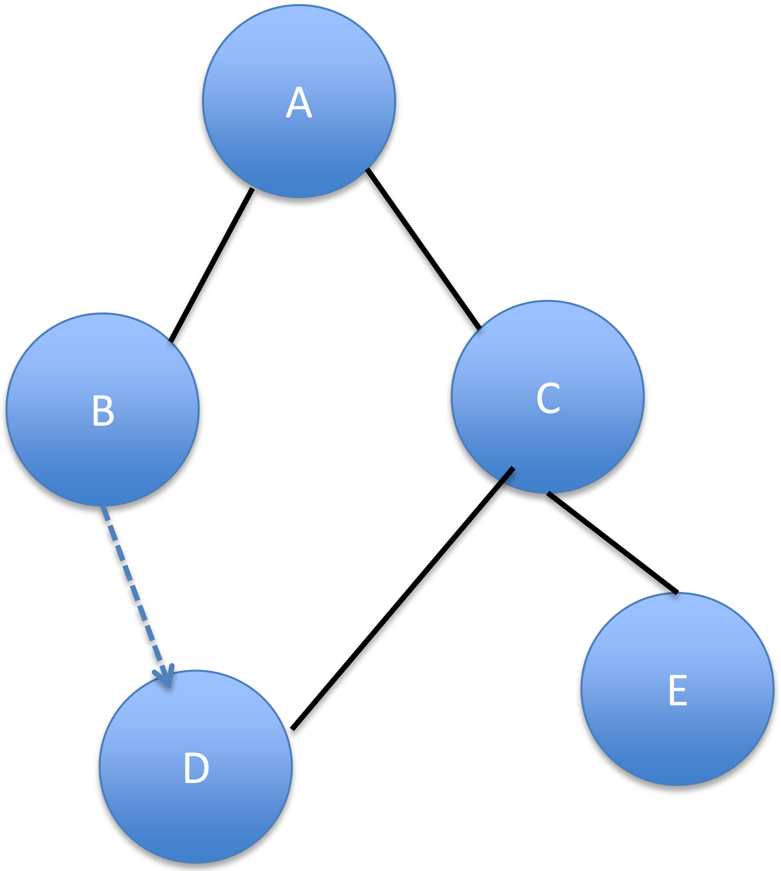
Ex of common coupling



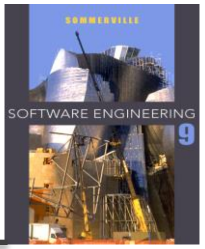
✧ **Content Coupling:** Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another.

```
Module B
-------
-------
Go to temp
```

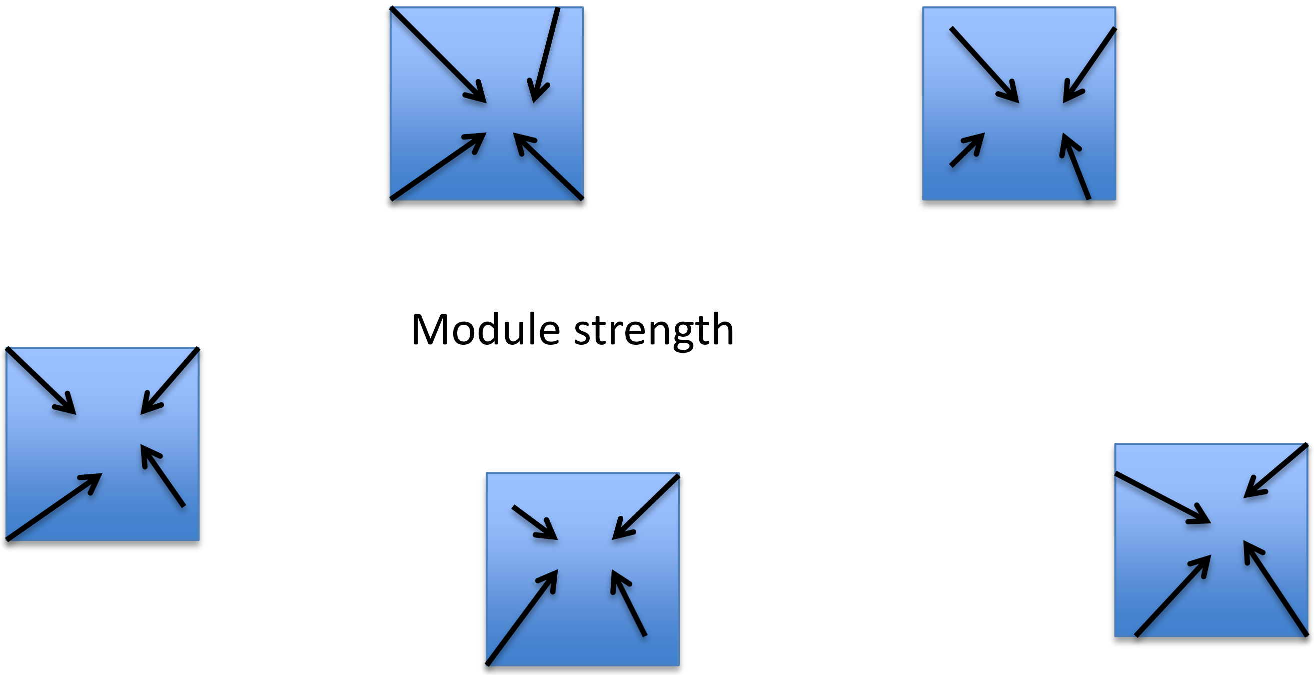
```
Module D
-------
---------
Go to temp
---------
Temp:-------
--------
```



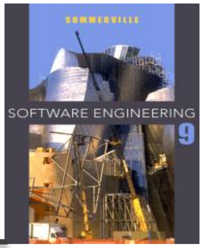
Module Cohesion



- Cohesion is a measure of the degree to which the elements of a module are functionally related..



Types of Cohesion



Functional Cohesion

Sequential Cohesion

Communicational Cohesion

Procedural Cohesion

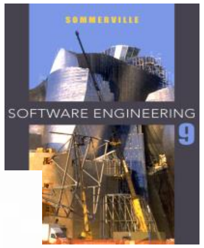
Temporal Cohesion

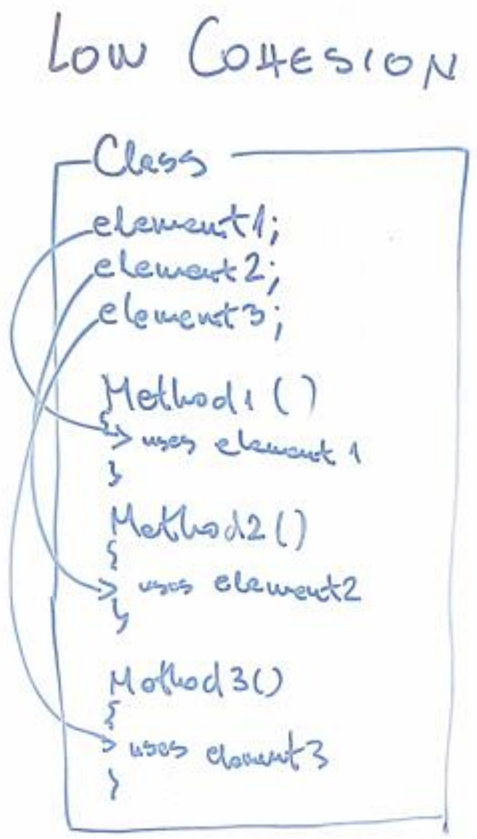
Logical Cohesion

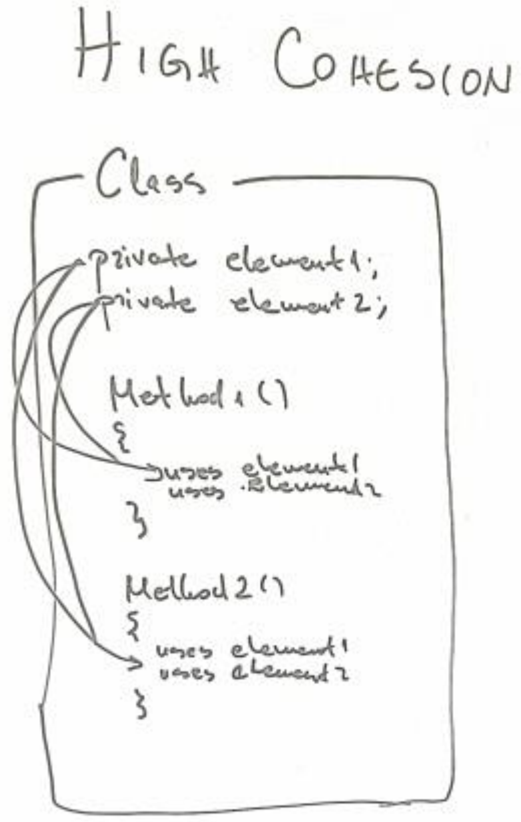
Coincidental Cohesion

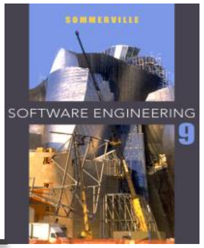
Best(high)

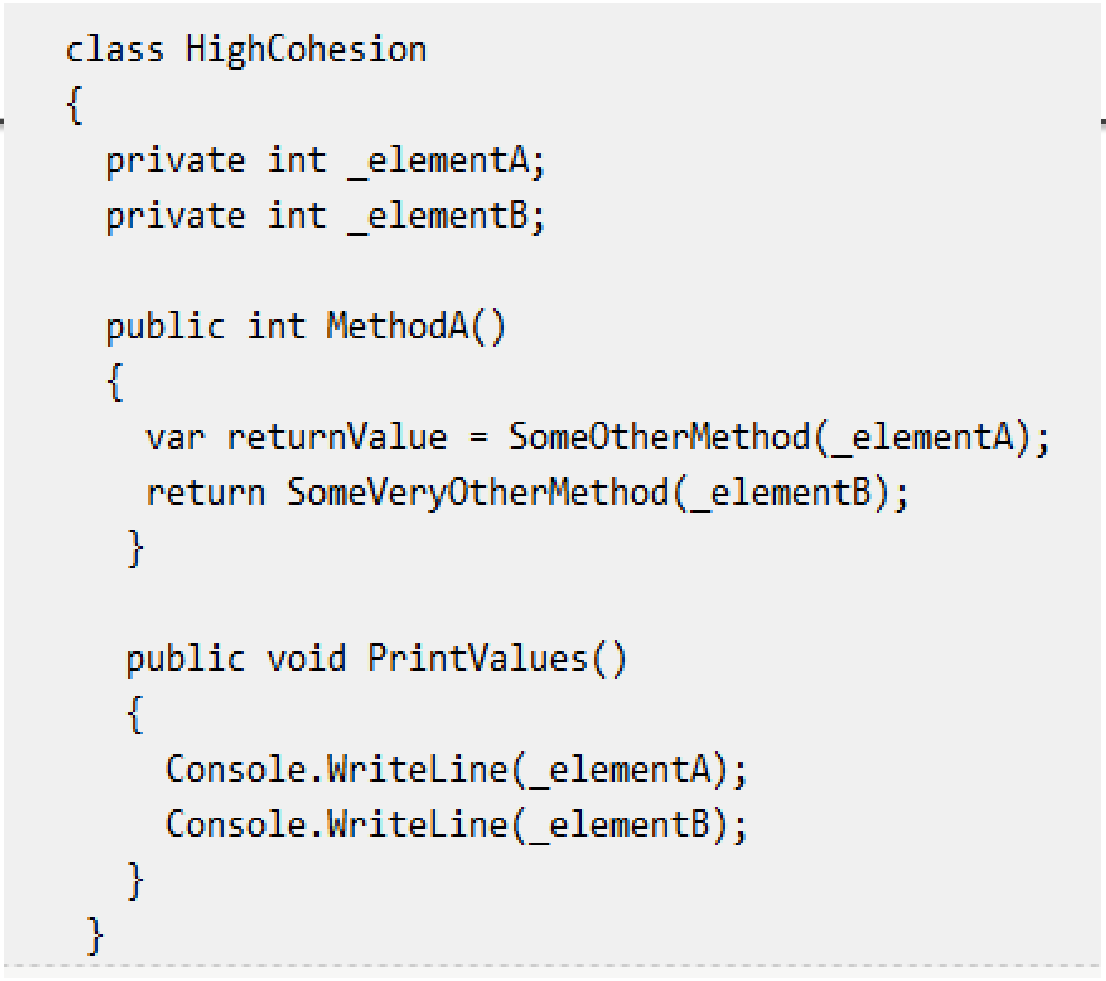
Worst(low)


SOMMERVILLE
SOFTWARE ENGINEERING
9

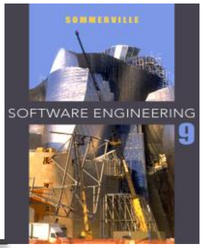

Low Cohesion
Class
element1;
element2;
element3;
Method1 ()
{
uses element 1
}
Method2 ()
{
uses element2
}
Method3()
{
uses element3
}

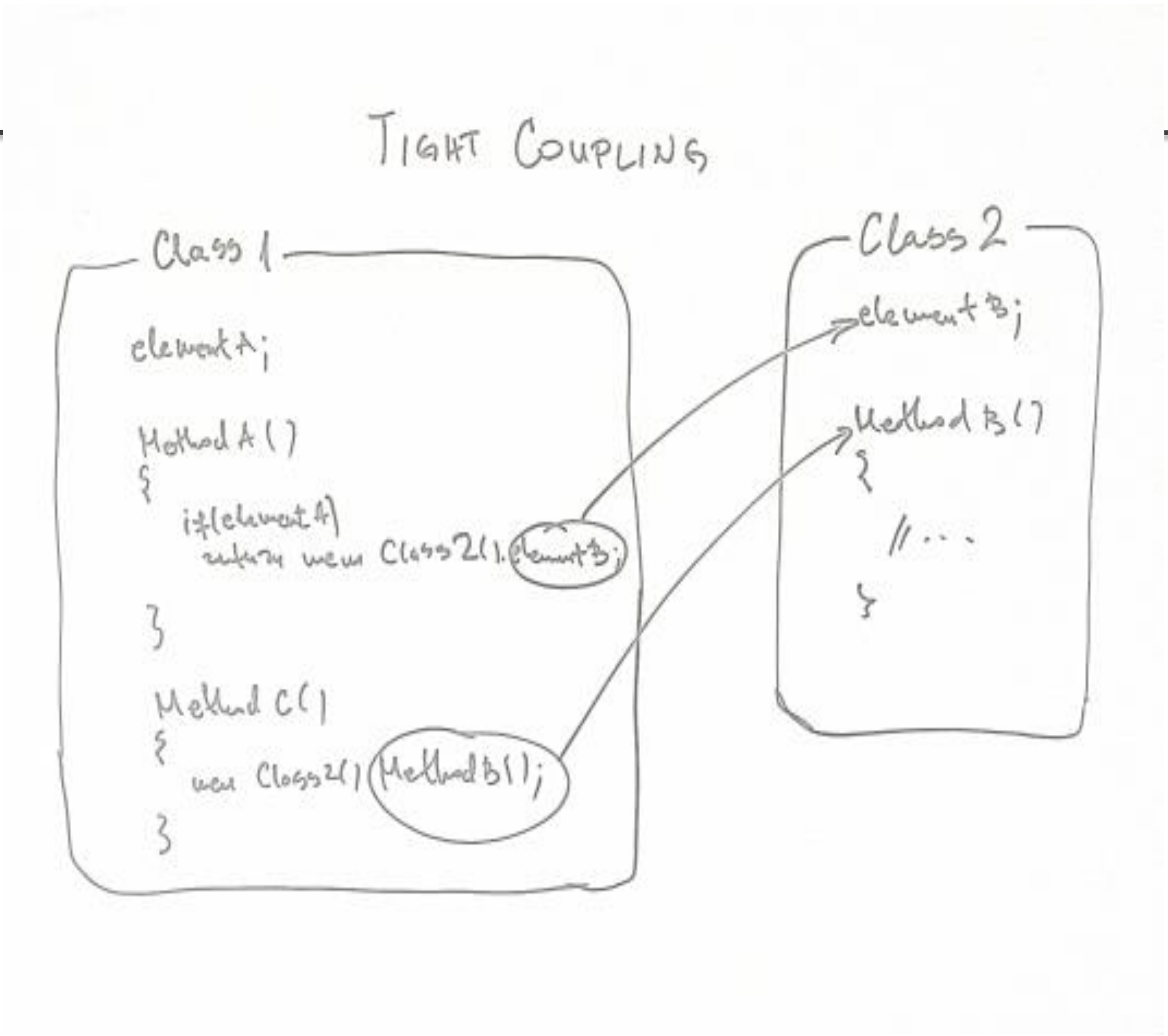

High Cohesion
Class
private element1;
private element 2;
Method1 ()
{
uses element1
uses element2
}
Method2 ()
{
uses element1
uses element2
}

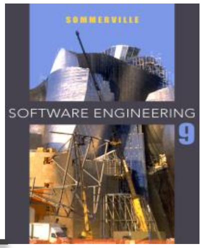


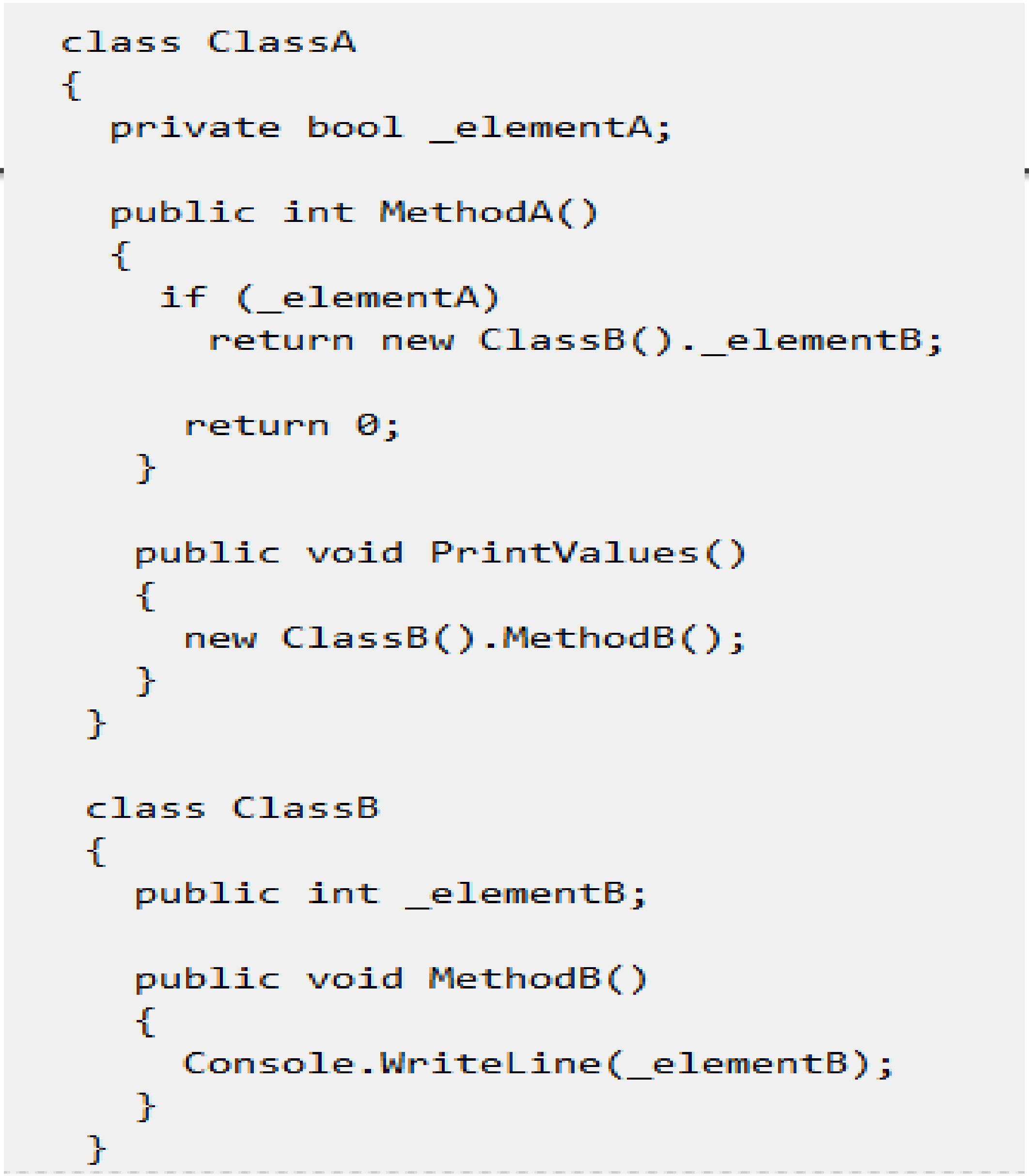


```
class HighCohesion
{
  private int _elementA;
  private int _elementB;

  public int MethodA()
  {
    var returnValue = SomeOtherMethod(_elementA);
    return SomeVeryOtherMethod(_elementB);
   }

   public void PrintValues()
   {
     Console.WriteLine(_elementA);
     Console.WriteLine(_elementB);
   }
 }
```

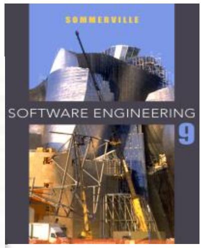


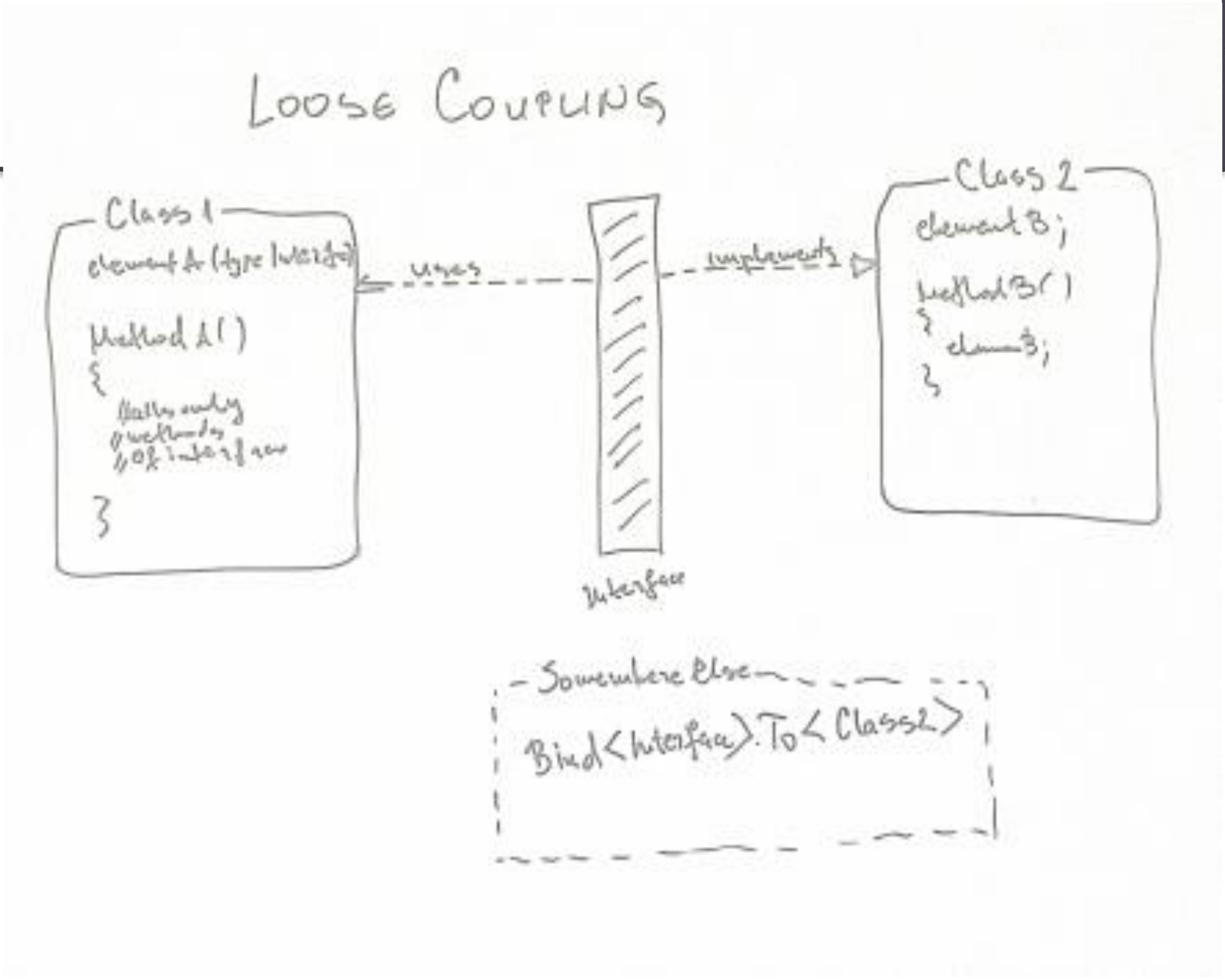

TIGHT COUPLING
Class 1
elementA;
MethodA()
{
if(elementA)
return new Class2().elementB;
}
MethodC()
{
new Class2().MethodB();
}
Class 2
elementB;
MethodB()
{
//...
}





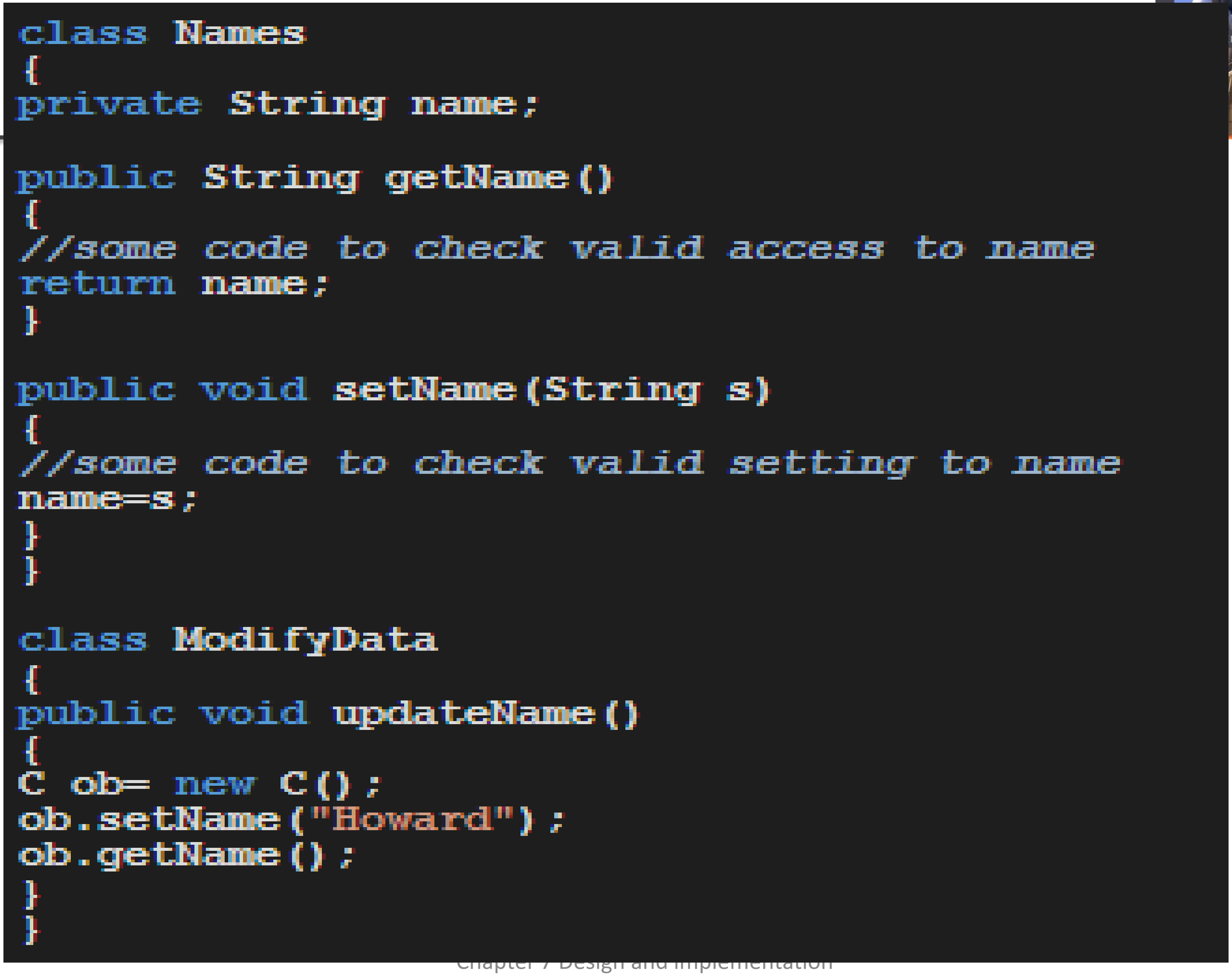
```
class ClassA
{
  private bool _elementA;

  public int MethodA()
  {
    if (_elementA)
      return new ClassB()._elementB;

     return 0;
   }

   public void PrintValues()
   {
     new ClassB().MethodB();
   }
 }

 class ClassB
 {
   public int _elementB;

   public void MethodB()
   {
     Console.WriteLine(_elementB);
   }
 }
```


SOMMERVILLE
SOFTWARE ENGINEERING
9


Loose Coupling
Class 1
element A (type Interface)
Method A( )
{
//calls only
// methods
// of interface
}
uses
implements
Interface
Class 2
element B;
Method B( )
{
elem B;
}
Somewhere else
Bind<Interface>.To<Class2>

Loose Coupling

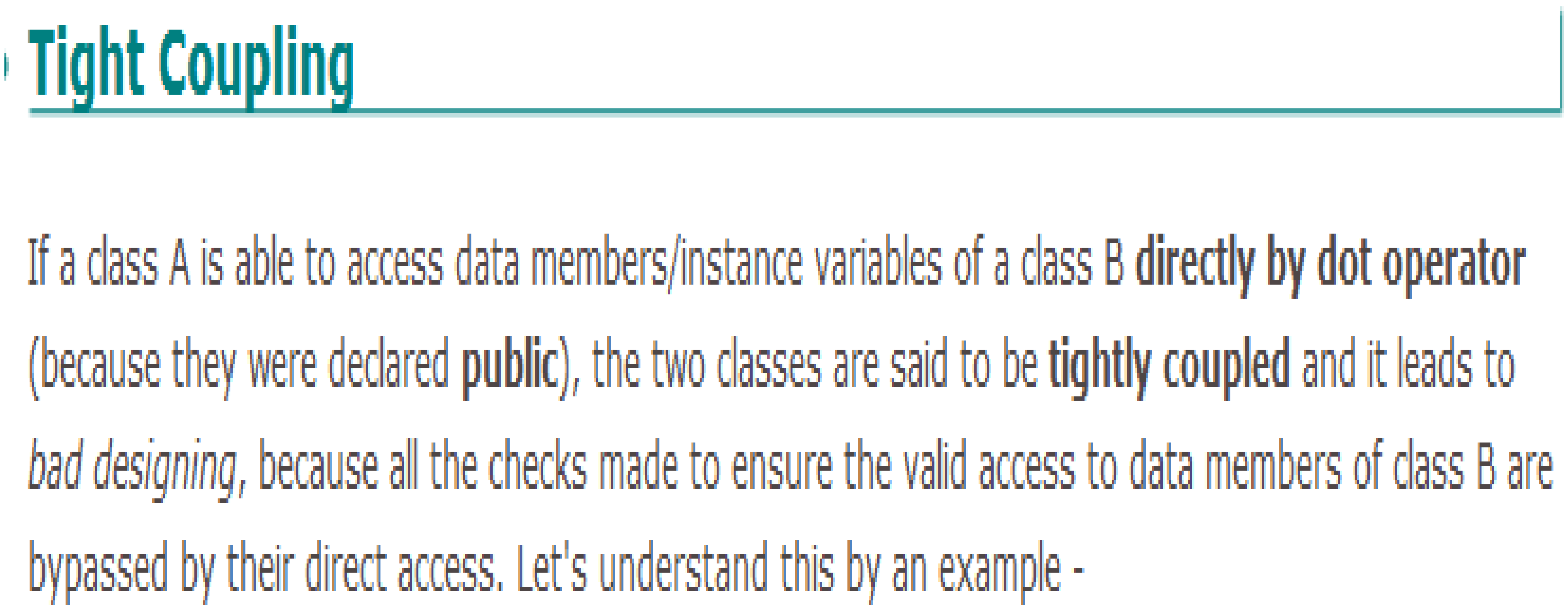
A good application designing is creating an application with loosely coupled classes by following proper **encapsulation**, i.e. by declaring data members of a class with **private** access and forcing other classes to access them only through **public getter, setter methods.** Let's understand this by an example -



```
class Names
{
private String name;

public String getName()
{
//some code to check valid access to name
return name;
}

public void setName(String s)
{
//some code to check valid setting to name
name=s;
}
}

class ModifyData
{
public void updateName()
{
C ob= new C();
ob.setName("Howard");
ob.getName();
}
}
```

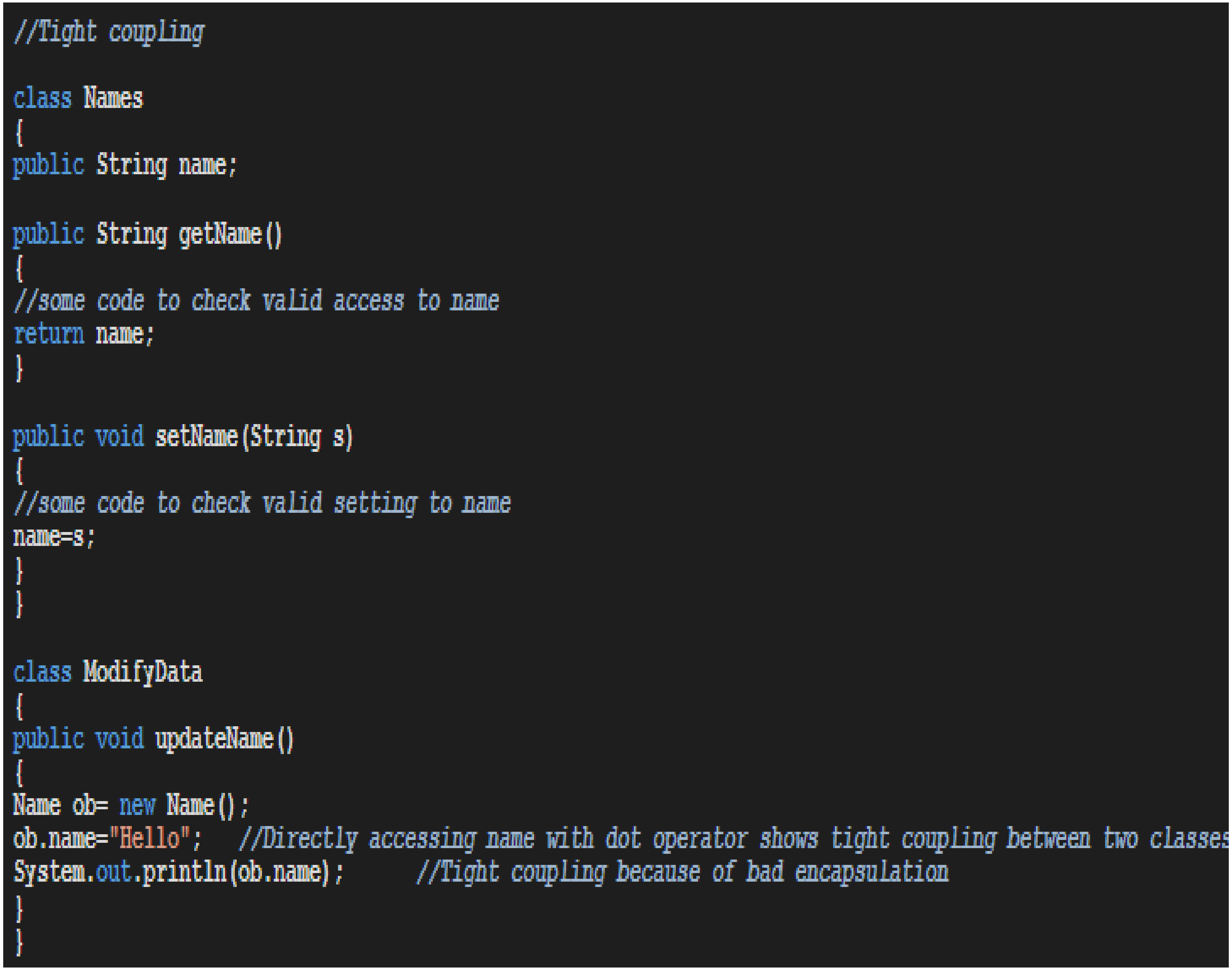
Program Analysis

- Class **Name** has an instance variable, name, which is declared **private.**
- Class **Name** has two **public** getter, setter methods which check for the valid access to name before *getting and setting the name.*
- Class **ModifyData** has a method **updateName( )**, which creates an object of Name class and using it, calls the **getName( )** and **setName( )** method of class Name.
- Hence, **getName( )** and **setName( )** methods of class Name are called and their *checks are run* before instance member, name, of class Name is get or set. It shows class **Name** is *loosely coupled* to class **ModifyData**, which is a *good designing*.

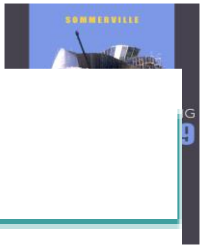


Tight Coupling

If a class A is able to access data members/instance variables of a class B **directly by dot operator** (because they were declared **public**), the two classes are said to be **tightly coupled** and it leads to *bad designing*, because all the checks made to ensure the valid access to data members of class B are bypassed by their direct access. Let's understand this by an example -



```
//Tight coupling

class Names
{
public String name;

public String getName()
{
//some code to check valid access to name
return name;
}

public void setName(String s)
{
//some code to check valid setting to name
name=s;
}
}

class ModifyData
{
public void updateName()
{
Name ob= new Name();
ob.name="Hello";   //Directly accessing name with dot operator shows tight coupling between two classes
System.out.println(ob.name);      //Tight coupling because of bad encapsulation
}
}
```



Program Analysis

- Class **Name** has an instance variable, name, which is declared **public**.
- Class **Name** has two **public** getter, setter methods which check for the valid access to name before getting and setting the name.
- Class **ModifyData** has a method **updateName()**, which creates an object of **Name** class and accesses its instance variable **directly by dot operator**, because they were declared **public**.
- Hence, **getName()** and **setName()** methods of class Name are never called and their security checks are *bypassed*. It shows class **Name** is *tighly coupled* to class **ModifyData**, which is a *bad designing*.

• Low Cohesion

When a class is designed to do many different tasks rather than focussing on a **single specialized task**, this class is said to be a "*low cohesive*" class. Low cohesive class are said to be *badly designed* leading to a hard time at creating, maintaining and updating them. Let's understand this by an example -

```
class PlayerDatabase
{
public void connectDatabase();
public void printAllPlayersInfo();
public void printSinglePlayerInfo();
public void printRankings()
public void closeDatabase();
}
```

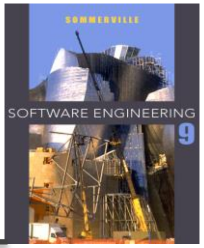
Program Analysis

Here, we have a class PlayerDatabase which is performing many different tasks like connecting to a database, printing the information of all the players, printing an information of a singe player, printing all the players, print players rankings and finally closing all open database connections. Now such a class is not easy to create, maintain and update and as it is involved in doing so many different tasks, a programming design to avoid.

High Cohesion

A good application design is creating an application with *high cohesive* classes, which are targeted towards a *specific specialized task* and such class is easy not only easy to create, but also easy to maintain and update.

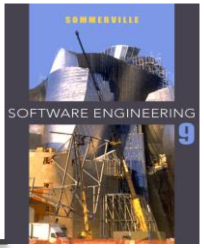
```
class PlayerDatabase
{
ConnectDatabase connectD= new connectDatabase();
PrintAllPlayersInfo allPlayer= new PrintAllPlayersInfo();
PrintRankings rankings = new PrintRankings();
CloseDatabase closeD= new CloseDatabase();
PrintSinglePlayerInfo singlePlayer = PrintSinglePlayerInfo();
}



class ConnectDatabase
{
//connecting to database.
}


class CloseDatabase
{
//closing the database connection.
}


class PrintRankings
{
//printing the players current rankings.
}


class PrintAllPlayersInfo
{
//printing all the players information.
}


class PrintSinglePlayerInfo
{
//printing a single player information.
}
```



Program Anaysis

Unlike the previous program where we had a single class performing many different tasks, here we have created severaldifferent classes, each class performing a specific specialized task, leading to an easy creation, maintanence and updation of these classes . Classes created by following this programming design are said to performing a cohesive role and are high cohesion classes, which is an appropriate programming design while creating an application.

Architectural Design(AD)



Large systems are always decomposed into subsystems that provides some related set of services. The initial design process of identifying these subsystems and establishing a framework for subsystems control and communication is called Architectural Design.

Output of this design process is a description of the s/w architecture.AD is the first stage in the design process and represents a critical link between the design and requirements engineering process.

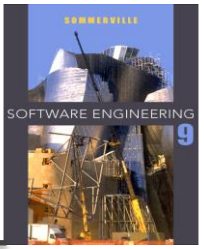
AD process is concerned with establishing a basic structural framework that identifies the major components of a system and the communication between these components.

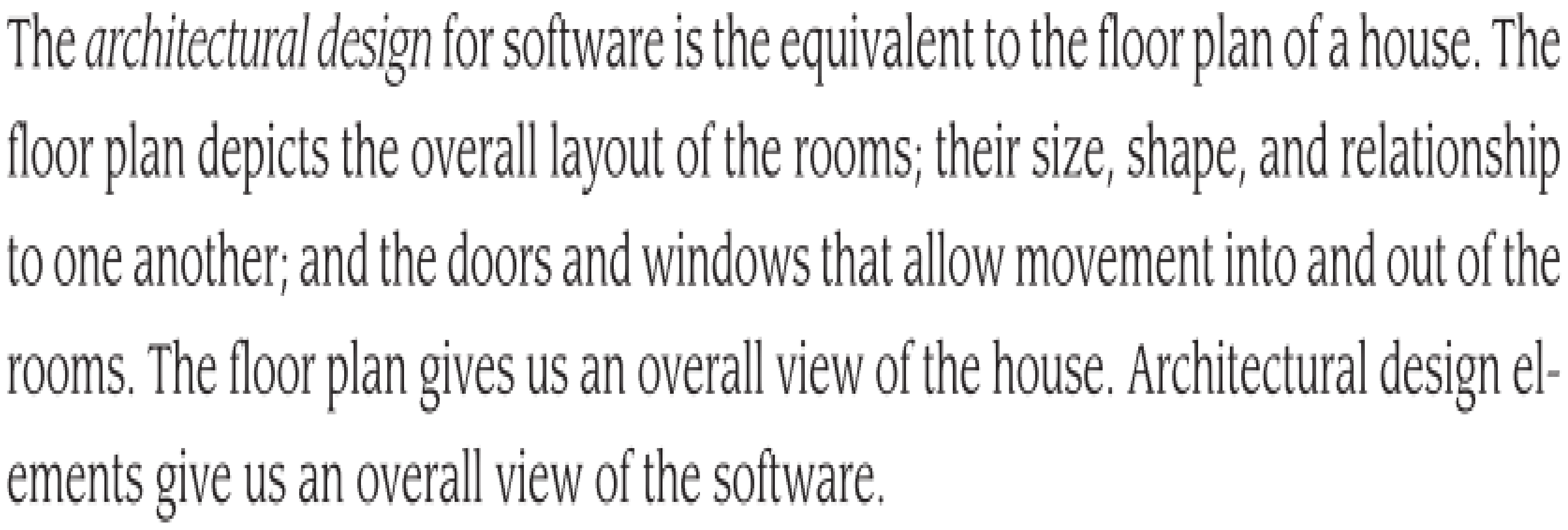
Advantages: **Stakeholder communications,System Analysis and Large scale reuse.**

The System architecture affects the performance ,robustness, distributability and maintainability of a system. The particular style and structure chosen for an application may therefore depend on NFR.(*performance, security, safety, availability, maintainability).*

Conflit:

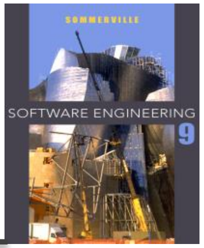
Architectural Design Elements





The *architectural design* for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

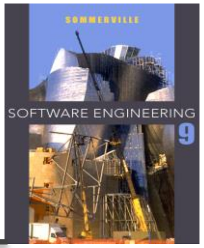
Interface Design Elements

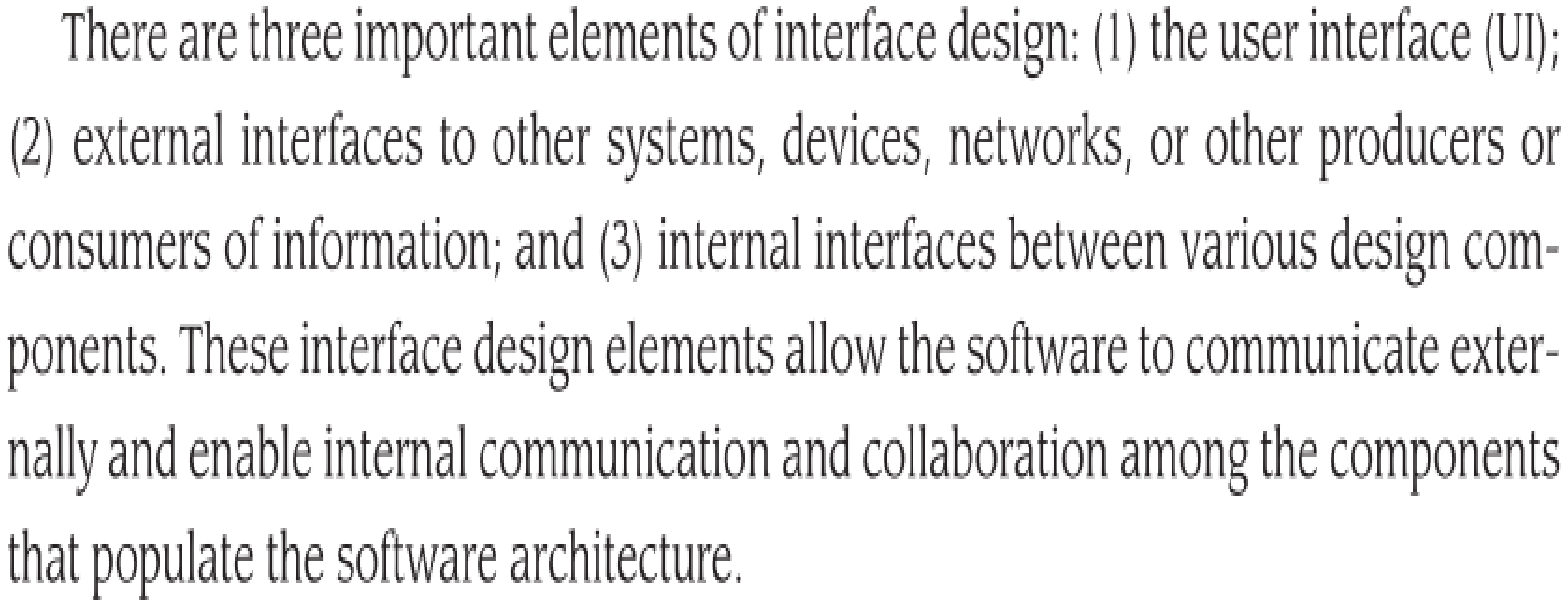


The interface design for software is analogous to a set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. These drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone) come into the house and are distributed among the rooms depicted in the floor plan.

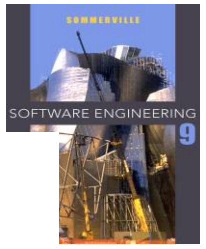
They tell us where the doorbell is located, whether an intercom is to be used to announce a visitor's presence, and how a security system is to be installed. In essence, the detailed drawings (and specifications) for the doors, windows, and external utilities tell us how things and information flow into and out of the house and within the rooms that are part of the floor plan. The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

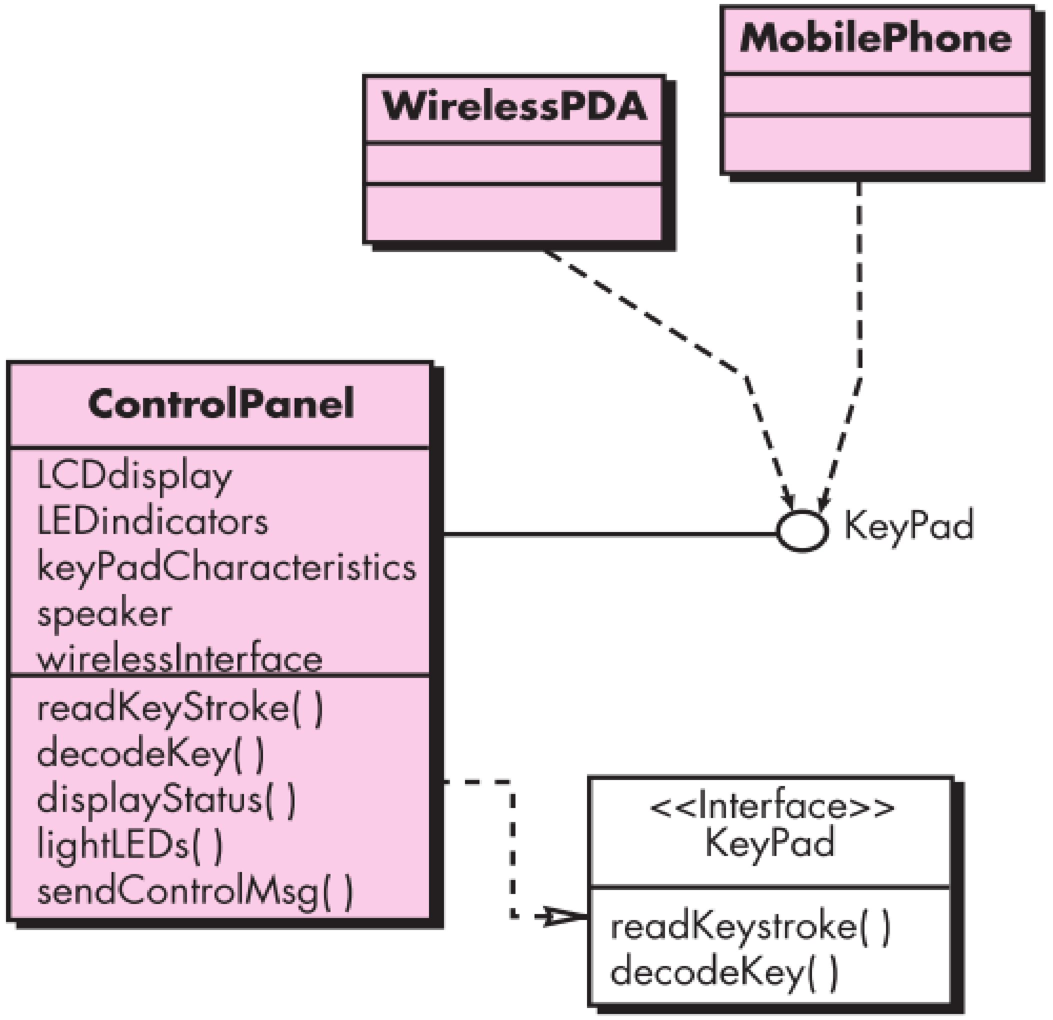
Elements of Interface Design



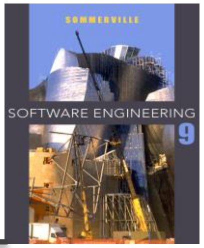


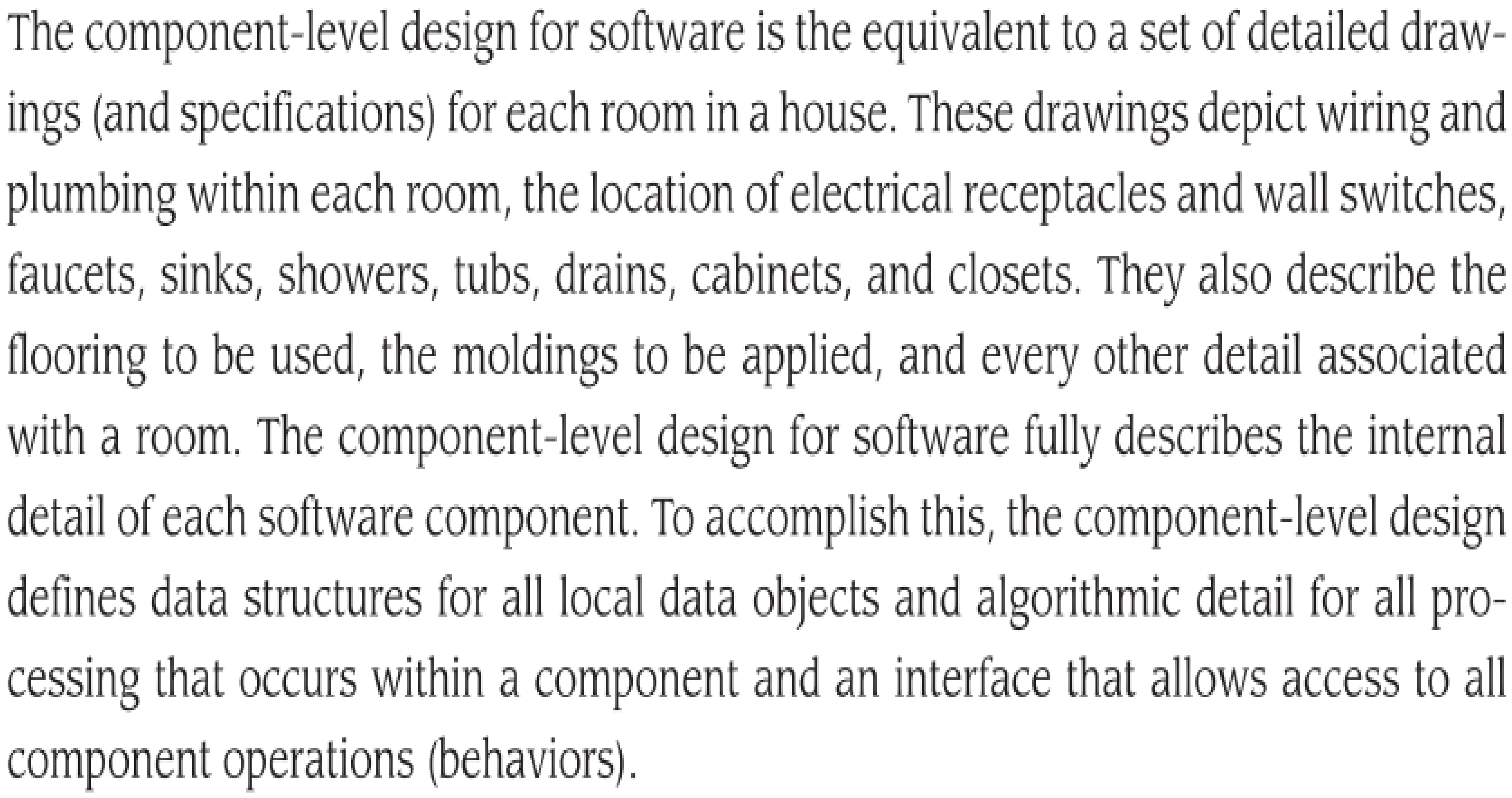
There are three important elements of interface design: (1) the user interface (UI); (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

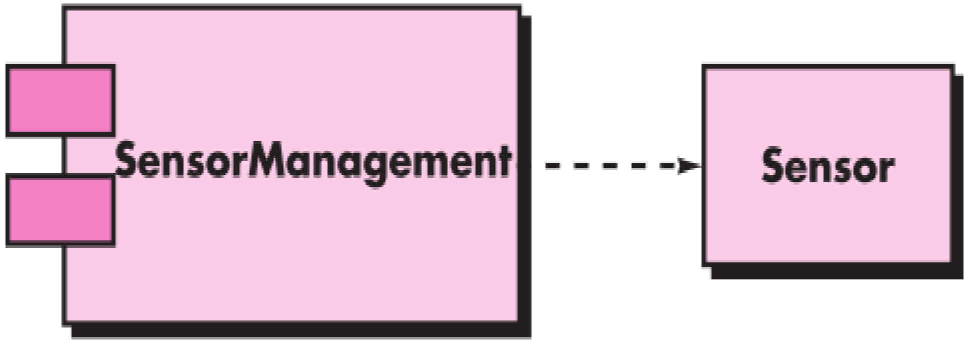



WirelessPDA
MobilePhone
ControlPanel
LCDdisplay
LEDindicators
keyPadCharacteristics
speaker
wirelessInterface
readKeyStroke( )
decodeKey( )
displayStatus( )
lightLEDs( )
sendControlMsg( )
KeyPad
<<Interface>>
KeyPad
readKeystroke( )
decodeKey( )

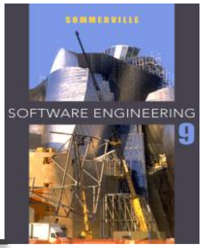
Component level design elements

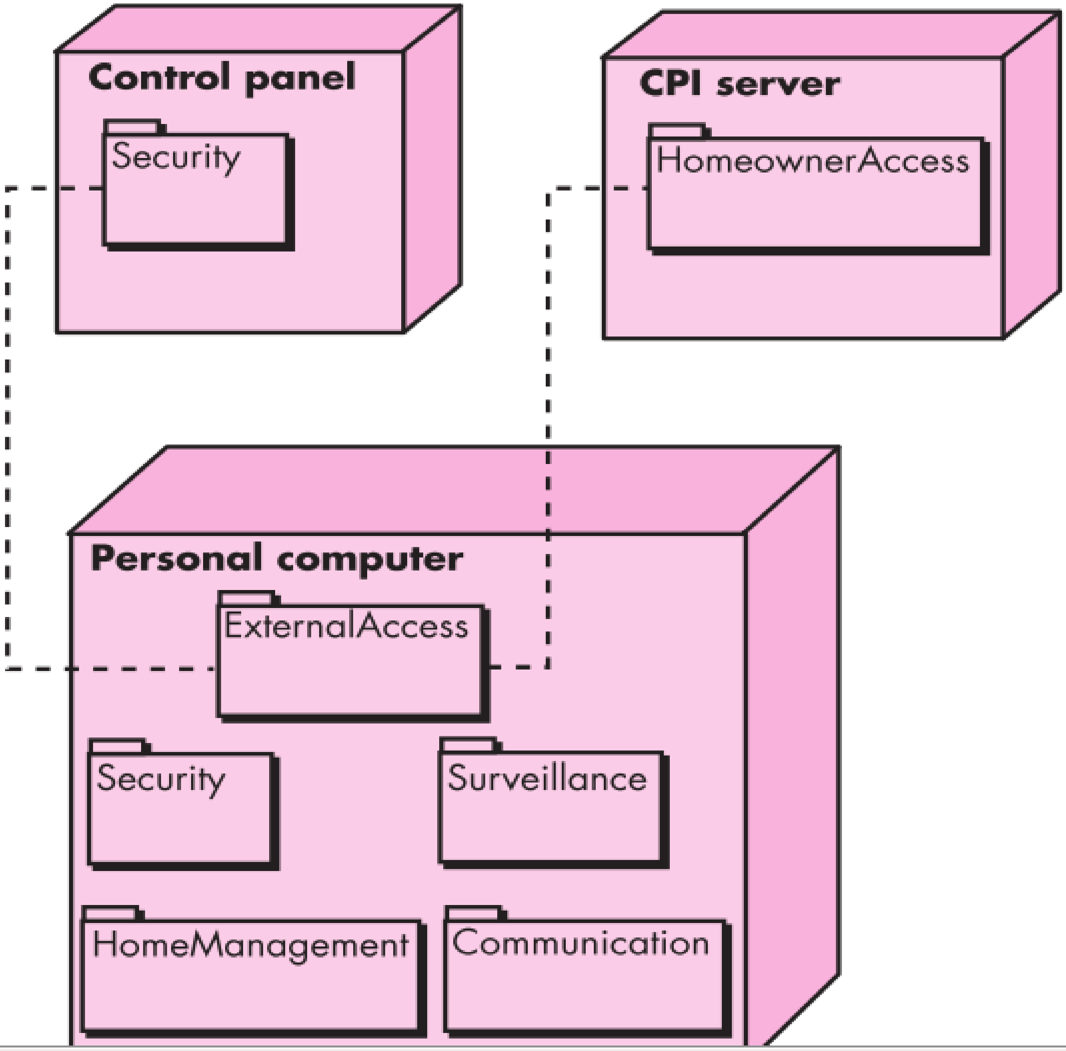


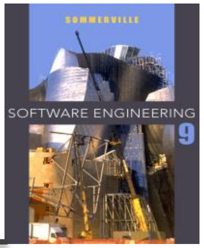

The component-level design for software is the equivalent to a set of detailed draw-
ings (and specifications) for each room in a house. These drawings depict wiring and
plumbing within each room, the location of electrical receptacles and wall switches,
faucets, sinks, showers, tubs, drains, cabinets, and closets. They also describe the
flooring to be used, the moldings to be applied, and every other detail associated
with a room. The component-level design for software fully describes the internal
detail of each software component. To accomplish this, the component-level design
defines data structures for all local data objects and algorithmic detail for all pro-
cessing that occurs within a component and an interface that allows access to all
component operations (behaviors).


SensorManagement
Sensor

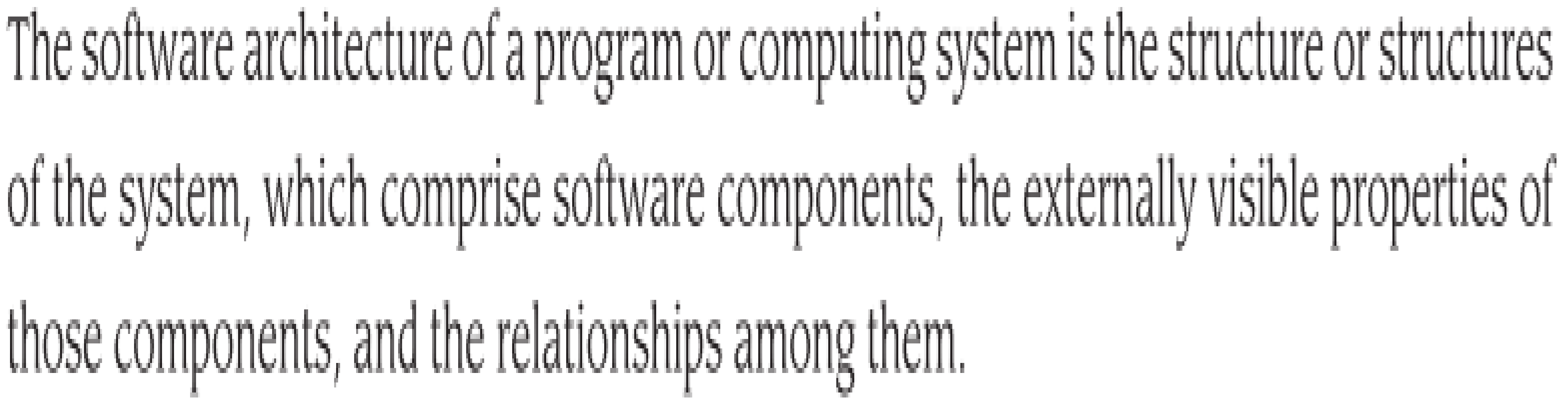
Deployment level design elements





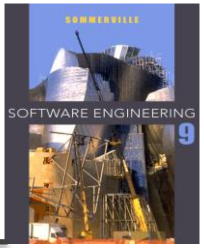


Software Architecture

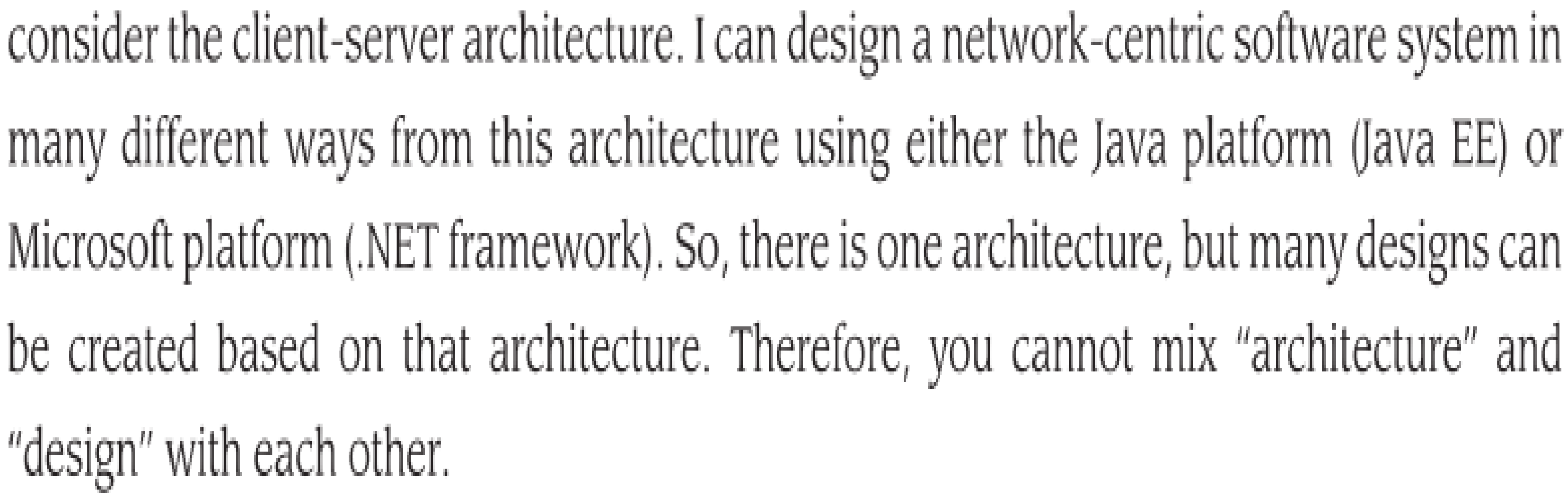


The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Architecture Vs Design

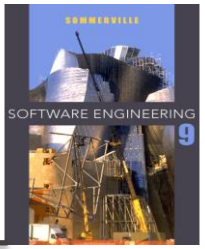


- ✧ Design is an instance of architecture like object is an instance of class.

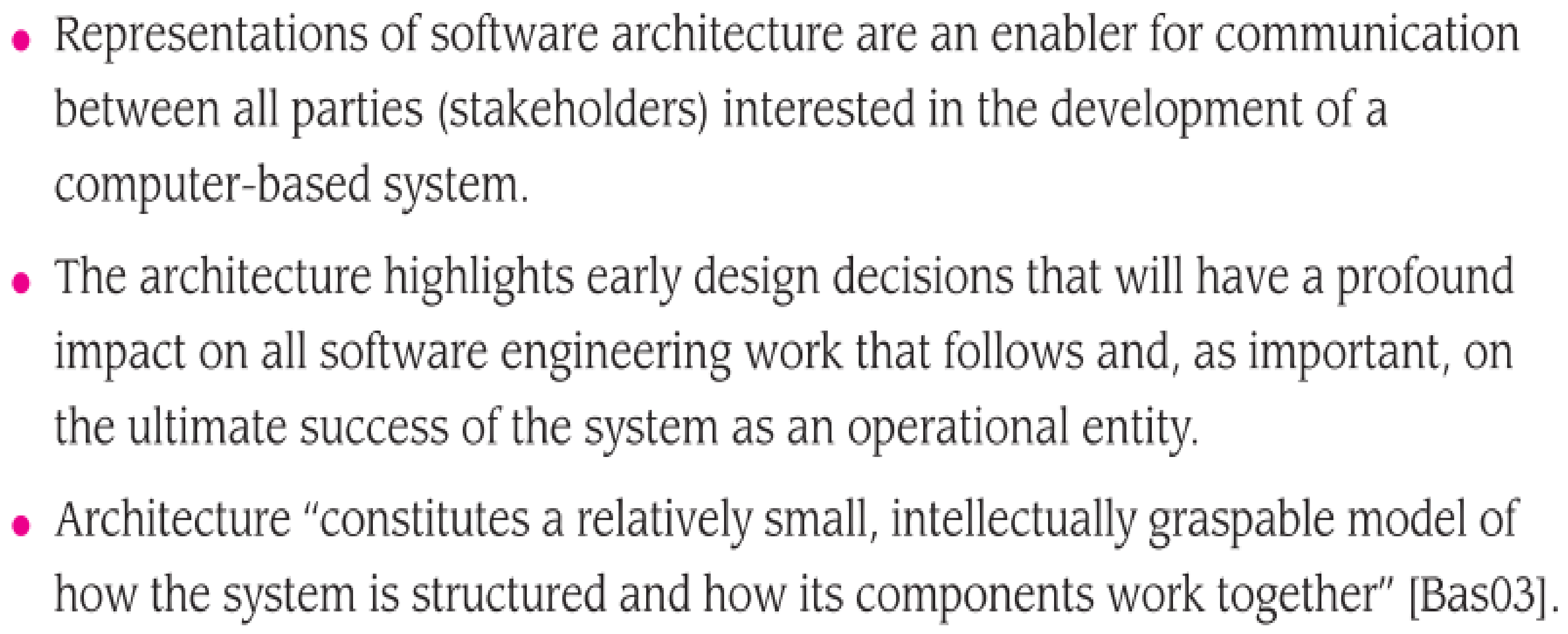


consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. Therefore, you cannot mix "architecture" and "design" with each other.

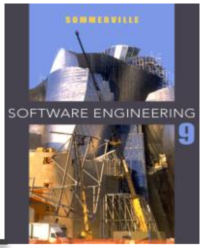
Two levels of Design



- ✧ Data Design
- ✧ Architectural Design
- ✧ **Importance of Architecture:**

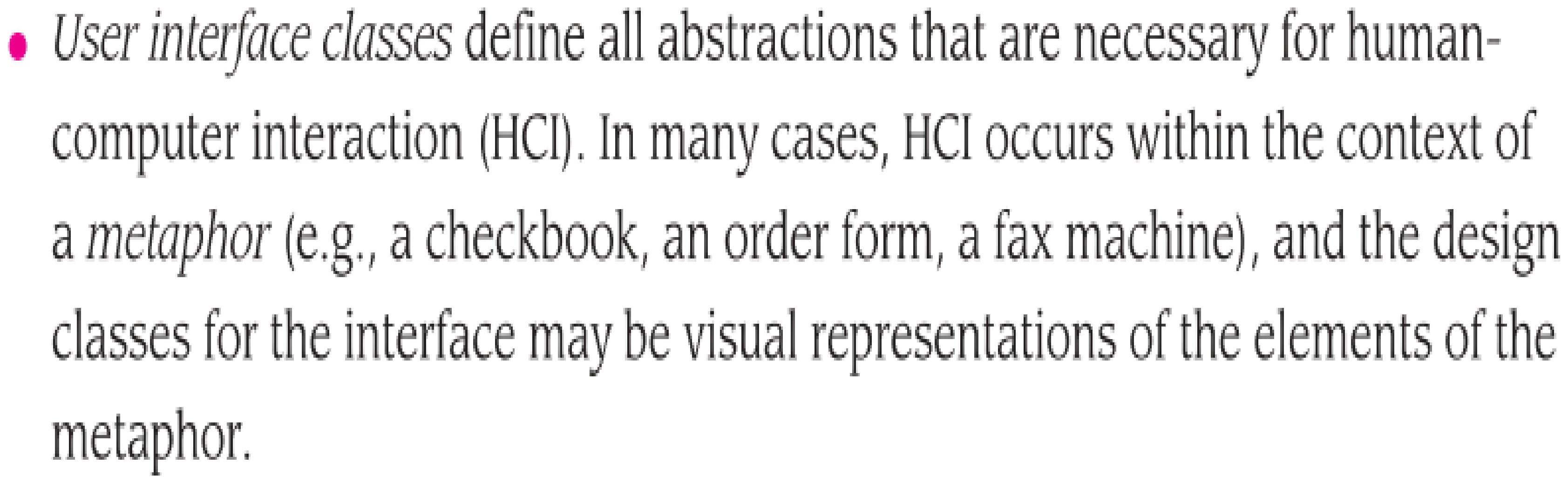


- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together" [Bas03].

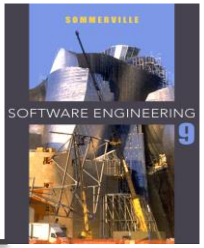


✧ **Design Classes:**

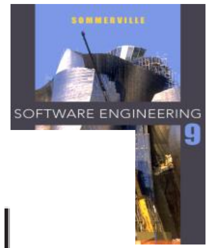
✧ Types of design classes:

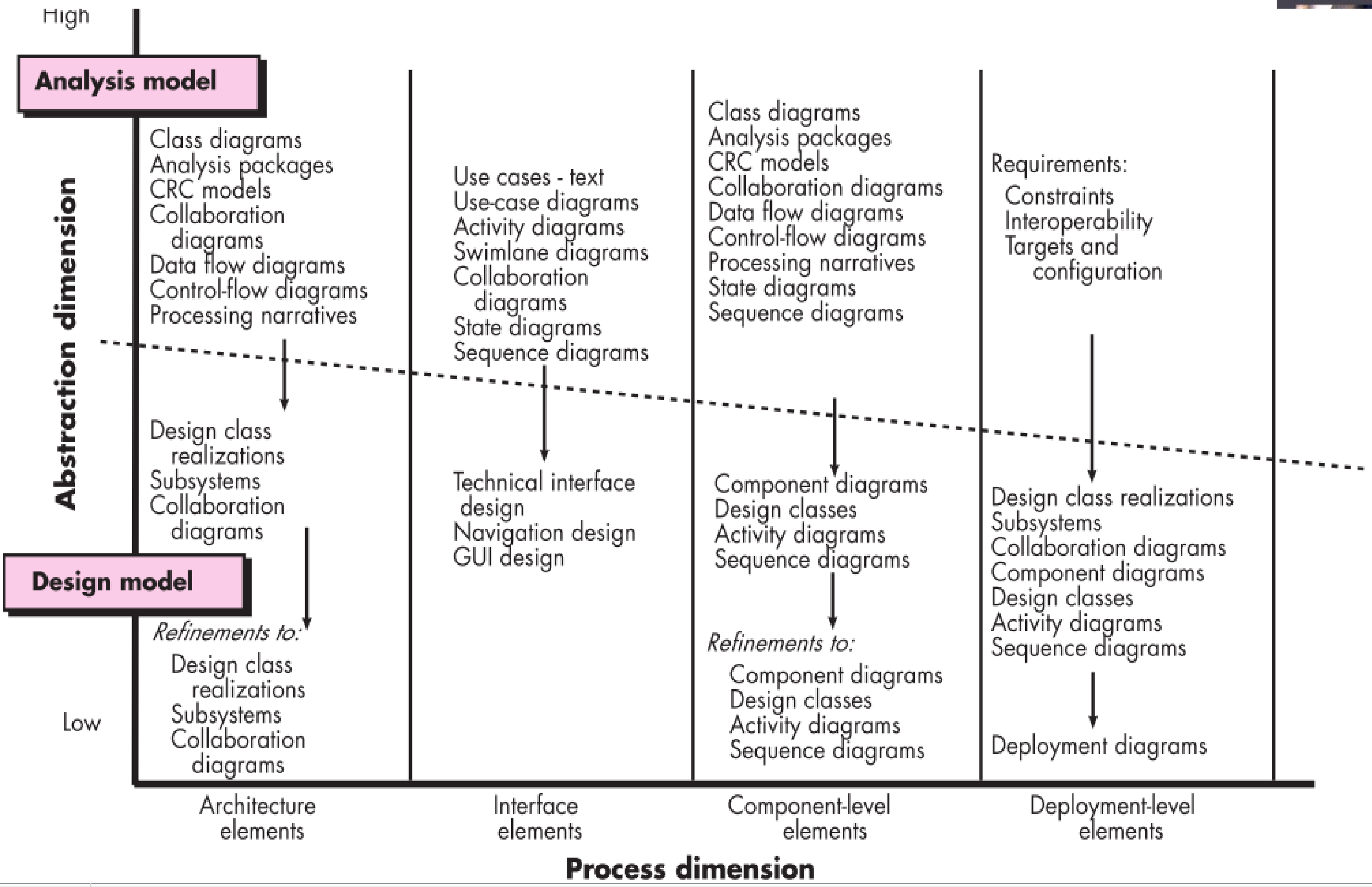


- *User interface classes* define all abstractions that are necessary for human-computer interaction (HCI). In many cases, HCI occurs within the context of a *metaphor* (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.

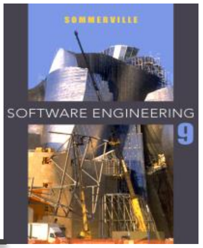


- *Business domain classes* are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- *Process classes* implement lower-level business abstractions required to fully manage the business domain classes.
- *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software.
- *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.



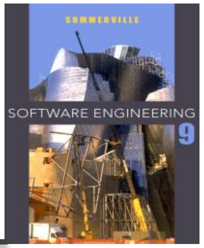

High
Analysis model
Abstraction dimension
Class diagrams
Analysis packages
CRC models
Collaboration diagrams
Data flow diagrams
Control-flow diagrams
Processing narratives
Use cases - text
Use-case diagrams
Activity diagrams
Swimlane diagrams
Collaboration diagrams
State diagrams
Sequence diagrams
Class diagrams
Analysis packages
CRC models
Collaboration diagrams
Data flow diagrams
Control-flow diagrams
Processing narratives
State diagrams
Sequence diagrams
Requirements:
Constraints
Interoperability
Targets and configuration
Design class realizations
Subsystems
Collaboration diagrams
Technical interface design
Navigation design
GUI design
Component diagrams
Design classes
Activity diagrams
Sequence diagrams
Design class realizations
Subsystems
Collaboration diagrams
Component diagrams
Design classes
Activity diagrams
Sequence diagrams
Design model
Refinements to:
Design class realizations
Subsystems
Collaboration diagrams
Refinements to:
Component diagrams
Design classes
Activity diagrams
Sequence diagrams
Deployment diagrams
Low
Architecture elements
Interface elements
Component-level elements
Deployment-level elements
Process dimension

Topics covered



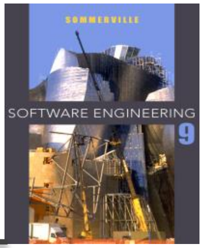
- ✧ Object-oriented design using the UML
- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open source development

Design and implementation



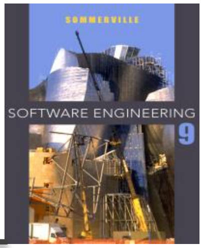
- ✧ Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- ✧ Software design and implementation activities are invariably inter-leaved.
  - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
  - Implementation is the process of realizing the design as a program.

Build or buy



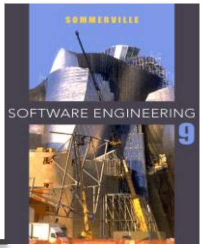
- ✧ In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
  - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- ✧ When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

An object-oriented design process



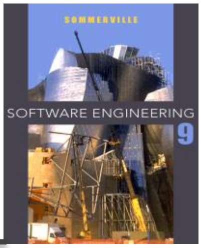
- ✧ Structured object-oriented design processes involve developing a number of different system models.
- ✧ They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- ✧ However, for large systems developed by different groups design models are an important communication mechanism.

Process stages



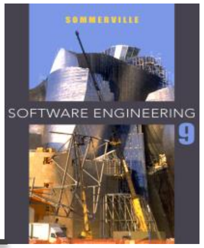
- ✧ There are a variety of different object-oriented design processes that depend on the organization using the process.
- ✧ Common activities in these processes include:
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.
- ✧ Process illustrated here using a design for a wilderness weather station.

System context and interactions



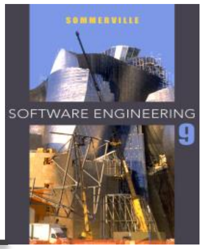
- ✧ Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- ✧ Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

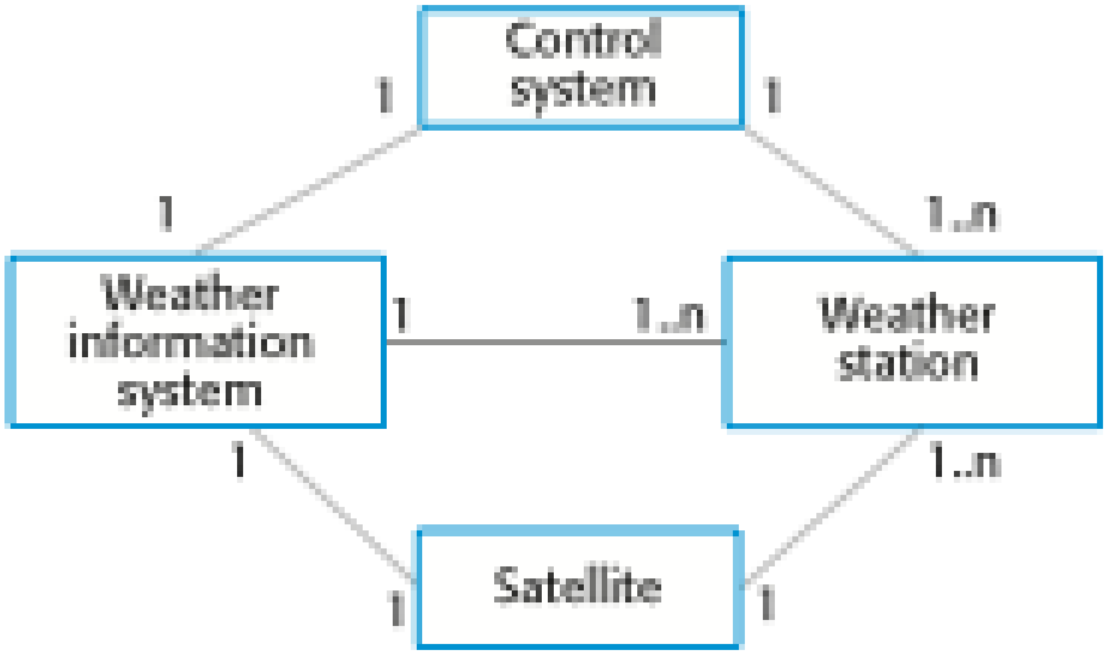
Context and interaction models



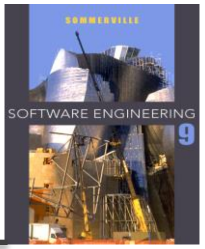
- ✧ A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
- ✧ An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

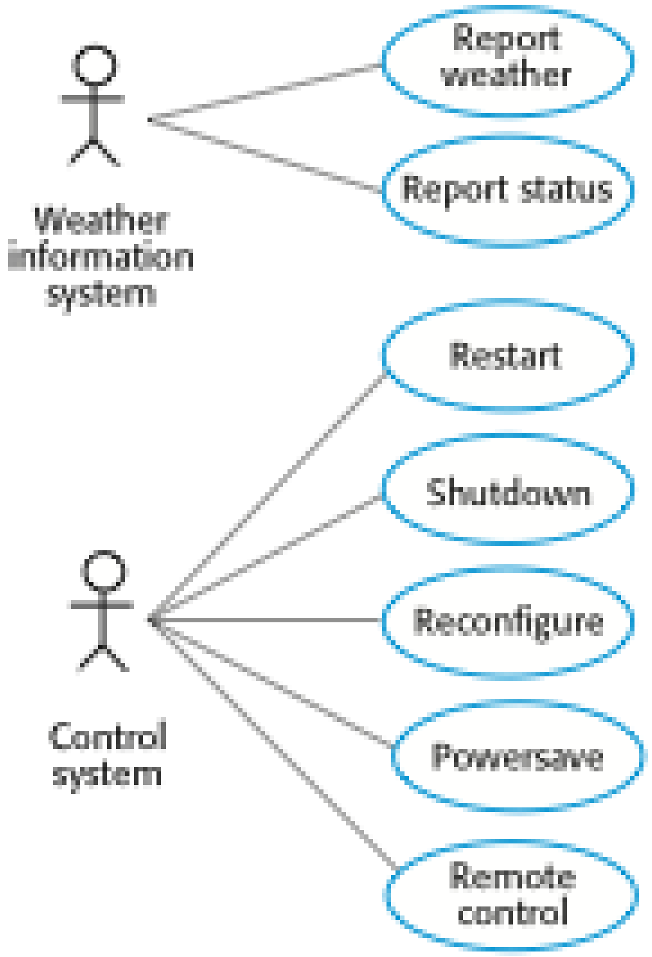
System context for the weather station



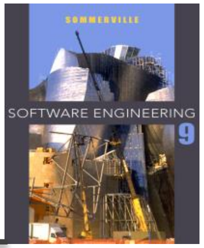


Weather station use cases



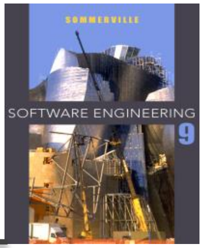


Use case description—Report weather



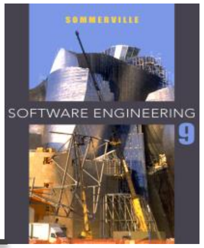
System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

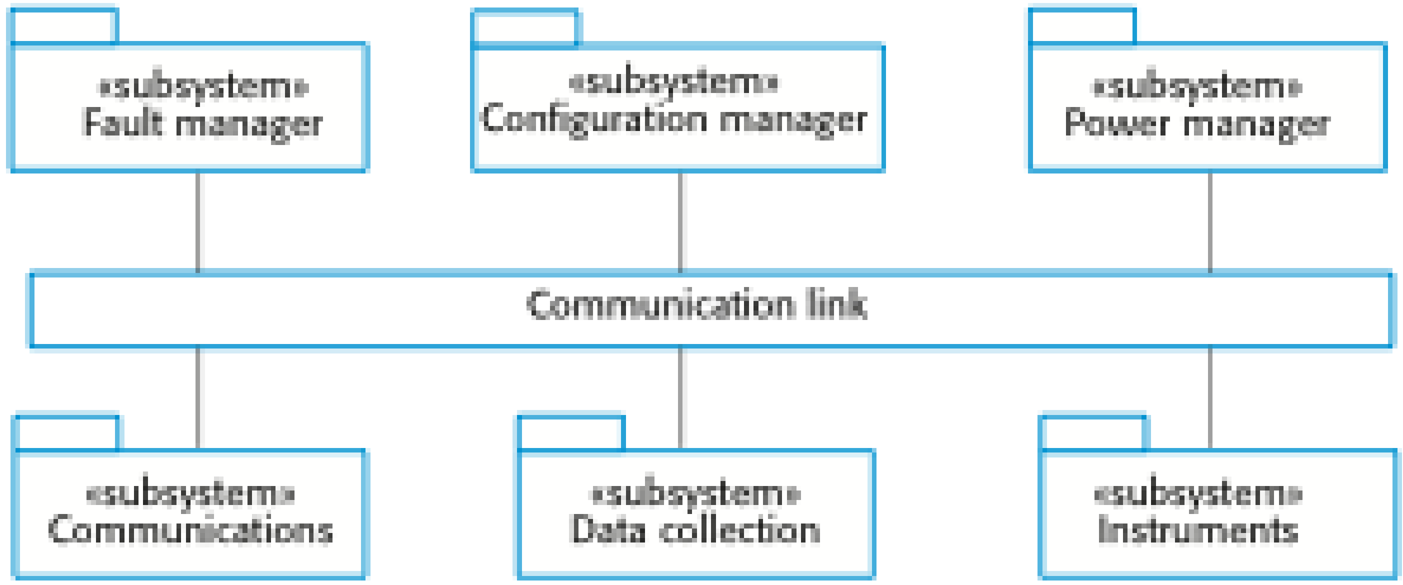
Architectural design



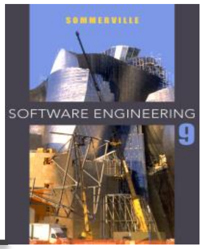
- ✧ Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- ✧ You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.
- ✧ The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.

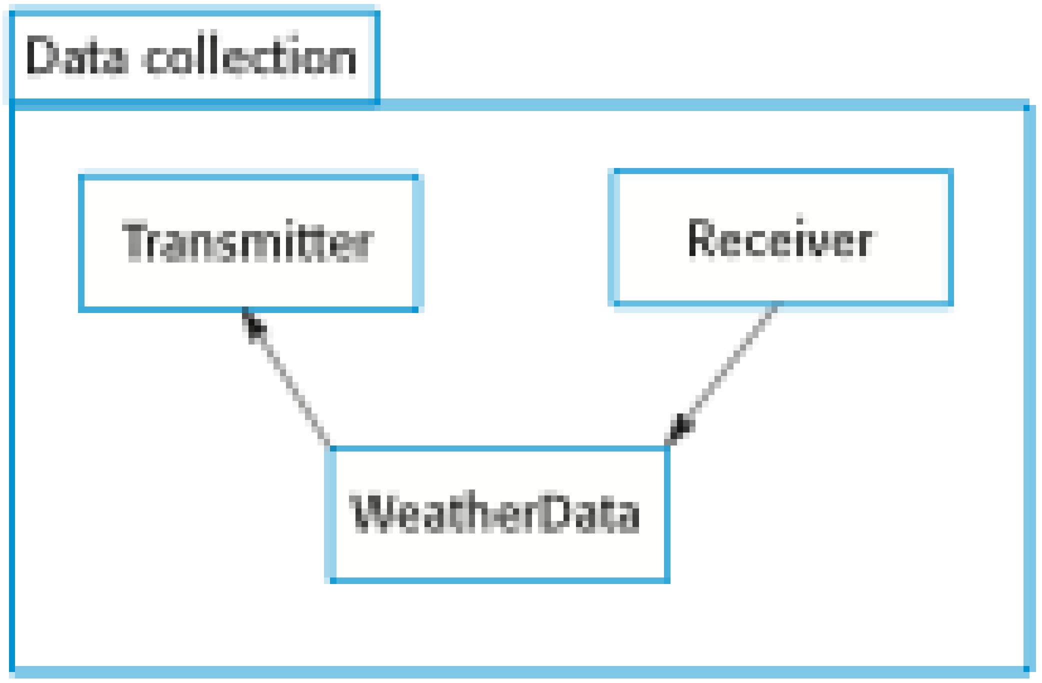
High-level architecture of the weather station



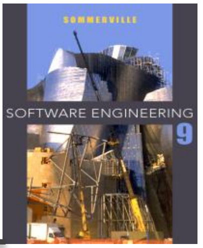

«subsystem» Fault manager
«subsystem» Configuration manager
«subsystem» Power manager
Communication link
«subsystem» Communications
«subsystem» Data collection
«subsystem» Instruments

Architecture of data collection system



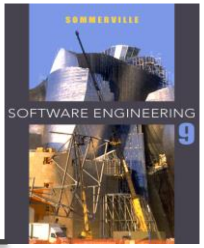


Object class identification



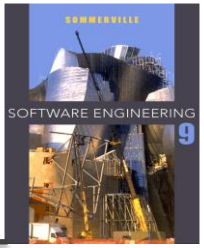
- ✧ Identifying object classes is toften a difficult part of object oriented design.
- ✧ There is no 'magic formula' for object identification. It relies on the skill, experience
  and domain knowledge of system designers.
- ✧ Object identification is an iterative process. You are unlikely to get it right first time.

Approaches to identification



- ✧ Use a grammatical approach based on a natural language description of the system (used in Hood OOD method).
- ✧ Base the identification on tangible things in the application domain.
- ✧ Use a behavioural approach and identify objects based on what participates in what behaviour.
- ✧ Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

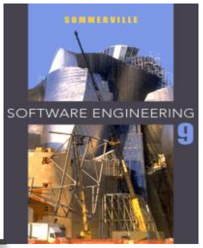
Weather station description



A weather station is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected periodically.

When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarised data is transmitted to the mapping computer when a request is received.

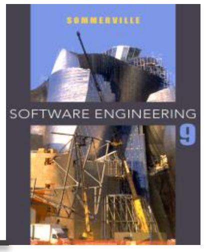
Weather station object classes

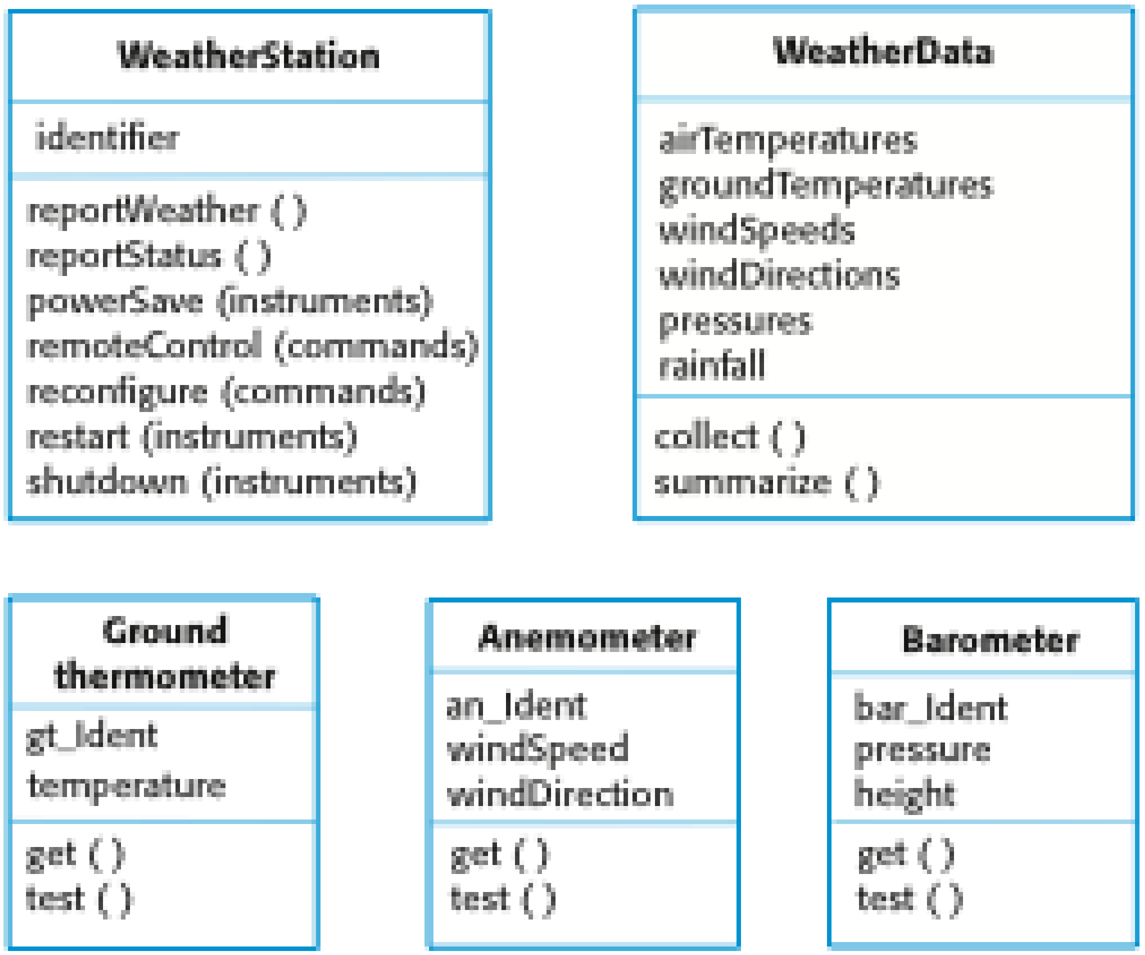


✧ Object class identification in the weather station system may be based on the tangible hardware and data in the system:

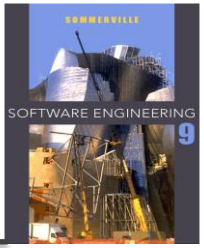
- Ground thermometer, Anemometer, Barometer
  - Application domain objects that are ‘hardware’ objects related to the instruments in the system.
- Weather station
  - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
- Weather data
  - Encapsulates the summarized data from the instruments.

Weather station object classes



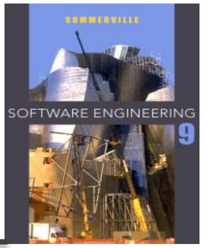


Design models



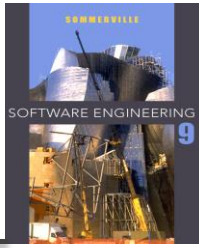
- ✧ Design models show the objects and object classes and relationships between these entities.
- ✧ Static models describe the static structure of the system in terms of object classes and relationships.
- ✧ Dynamic models describe the dynamic interactions between objects.

Examples of design models



- ✧ Subsystem models that show logical groupings of objects into coherent subsystems.
- ✧ Sequence models that show the sequence of object interactions.
- ✧ State machine models that show how individual objects change their state in response to events.
- ✧ Other models include use-case models, aggregation models, generalisation models, etc.

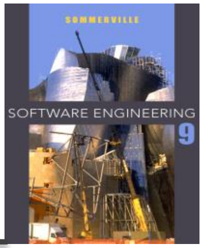
Subsystem models



✧ Shows how the design is organised into logically related groups of objects.

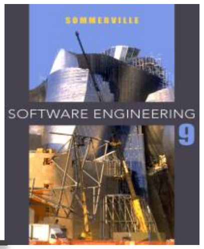
✧ In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

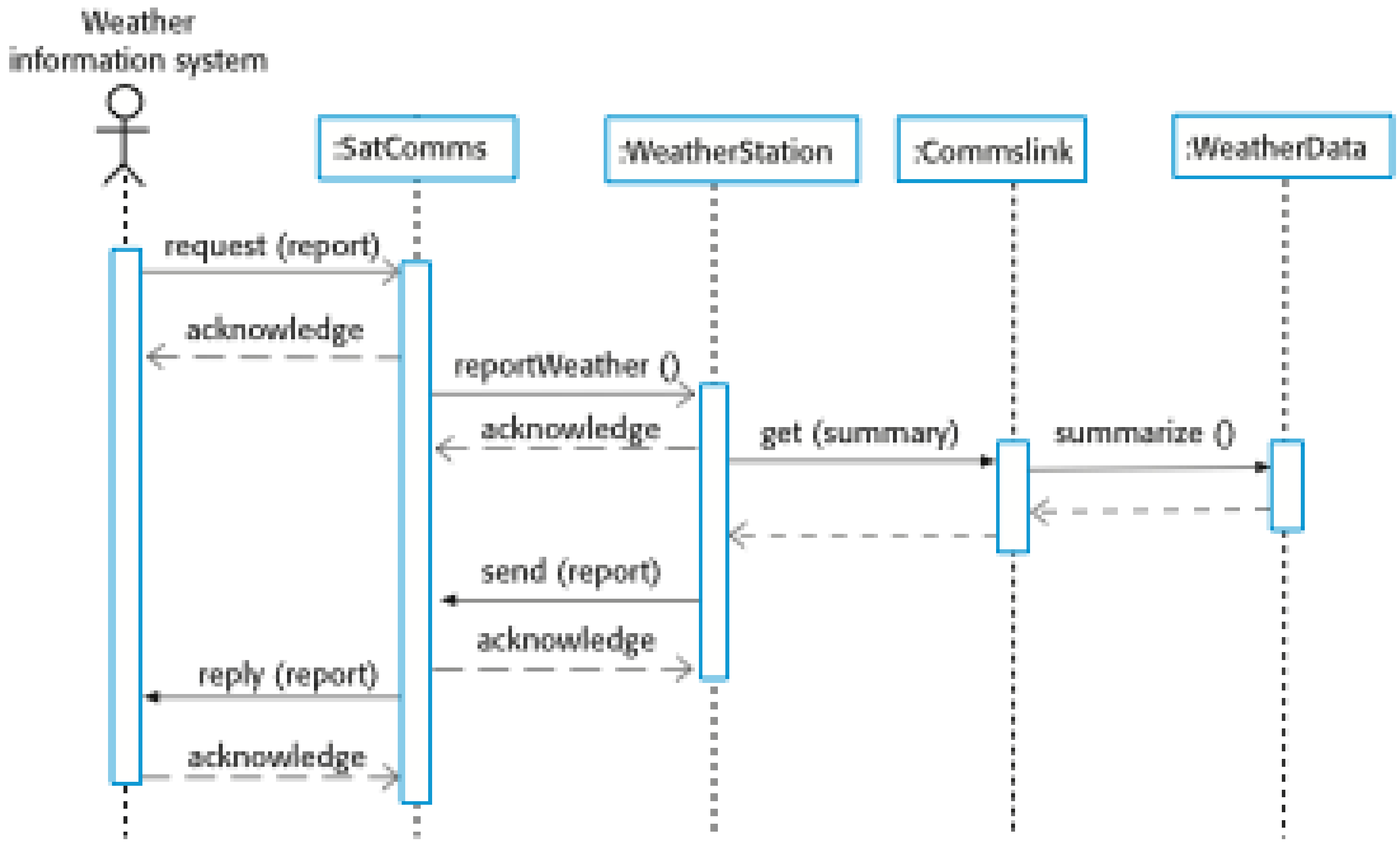
Sequence models



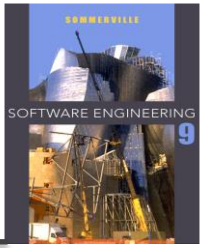
- Sequence models show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read top to bottom;
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

Sequence diagram describing data collection



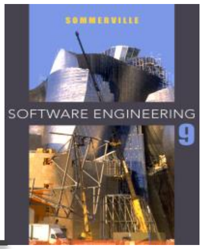


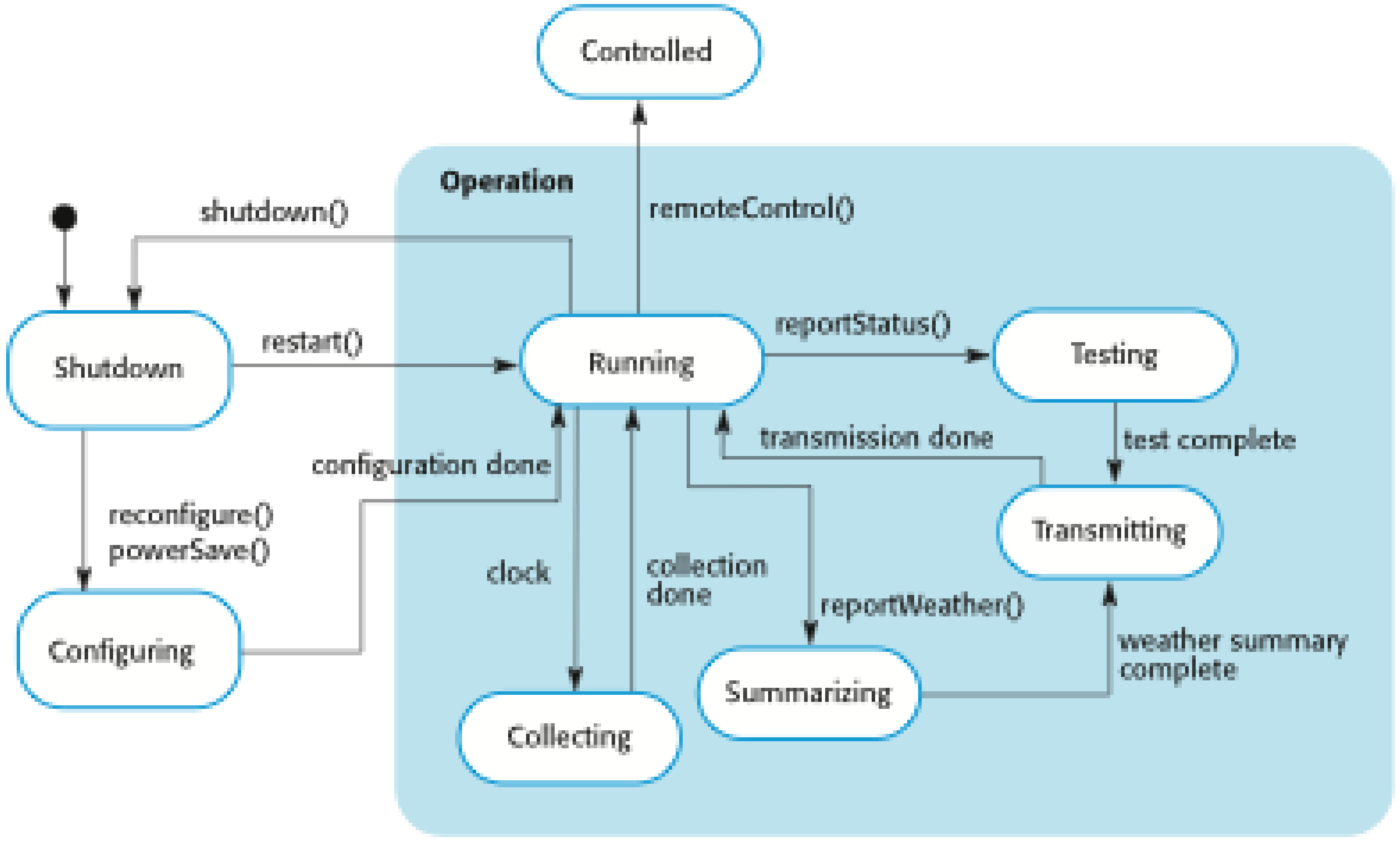
State diagrams



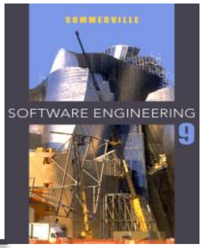
- ✧ State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- ✧ State diagrams are useful high-level models of a system or an object’s run-time behavior.
- ✧ You don’t usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

Weather station state diagram



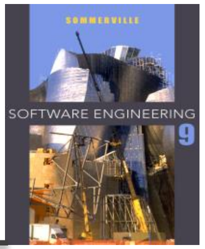


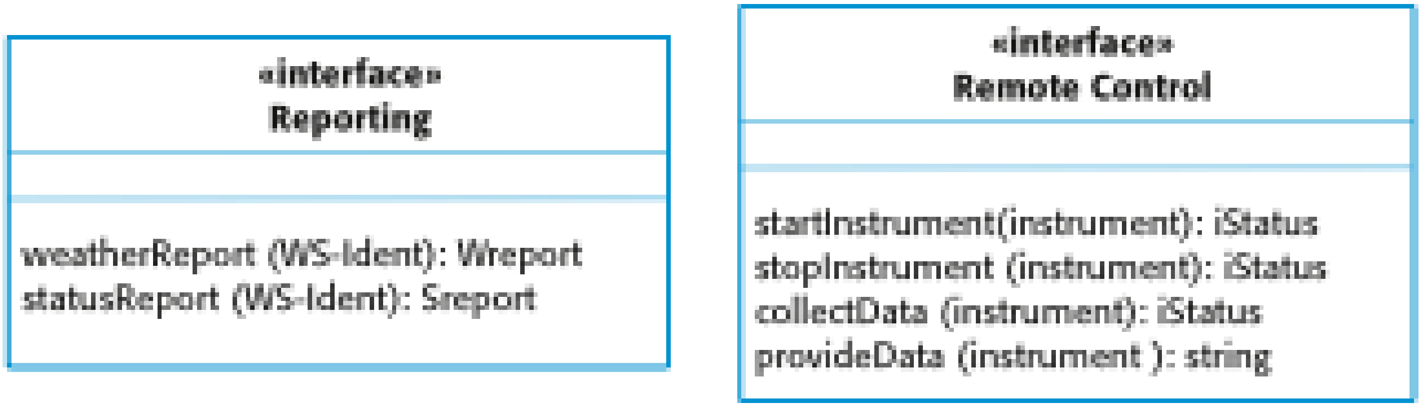
Interface specification



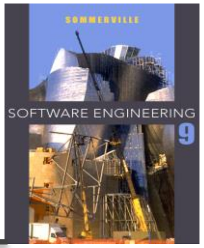
- ✧ Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- ✧ Designers should avoid designing the interface representation but should hide this in the object itself.
- ✧ Objects may have several interfaces which are viewpoints on the methods provided.
- ✧ The UML uses class diagrams for interface specification but Java may also be used.

Weather station interfaces




«interface»
Reporting
weatherReport (WS-Ident): Wreport
statusReport (WS-Ident): Sreport
«interface»
Remote Control
startInstrument(instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument): string

Key points

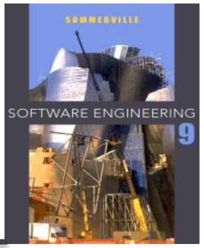


✧ Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.

✧ The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.

✧ A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).

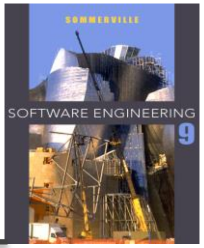
✧ Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.



Chapter 7 – Design and Implementation

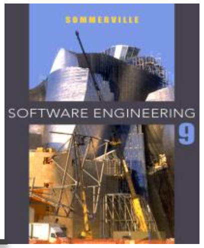
Lecture 2

Design patterns



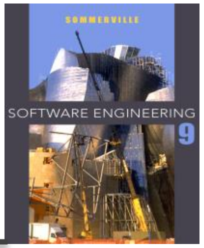
- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern descriptions usually make use of object-oriented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

Pattern elements



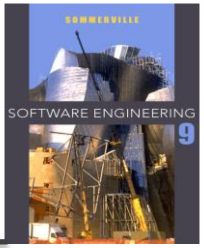
- ✧ Name
  - A meaningful pattern identifier.
- ✧ Problem description.
- ✧ Solution description.
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- ✧ Consequences
  - The results and trade-offs of applying the pattern.

The Observer pattern



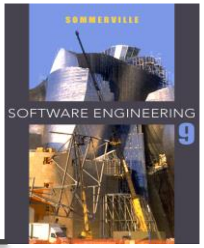
- ✧ Name
  - Observer.
- ✧ Description
  - Separates the display of object state from the object itself.
- ✧ Problem description
  - Used when multiple displays of state are needed.
- ✧ Solution description
  - See slide with UML description.
- ✧ Consequences
  - Optimisations to enhance display performance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

The Observer pattern (1)



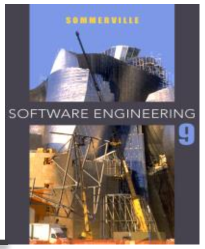
Pattern name	**Observer**
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, 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.

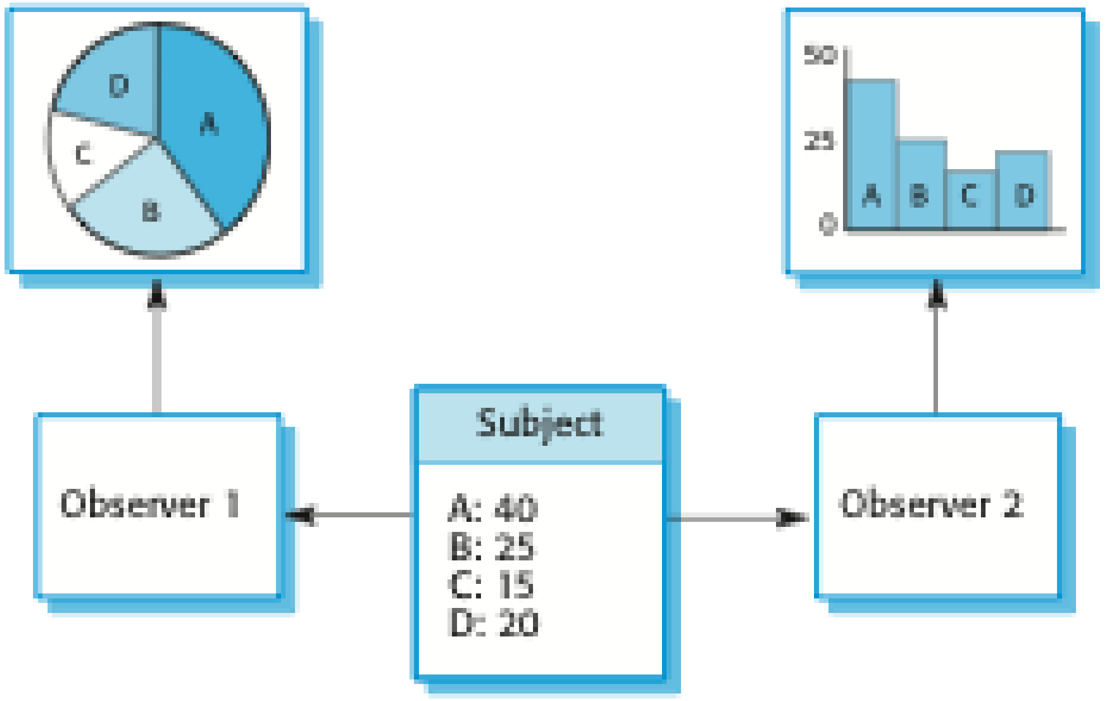
The Observer pattern (2)



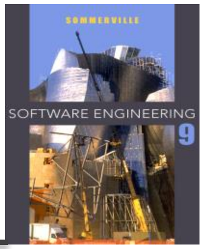
Pattern name	Observer
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. 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.

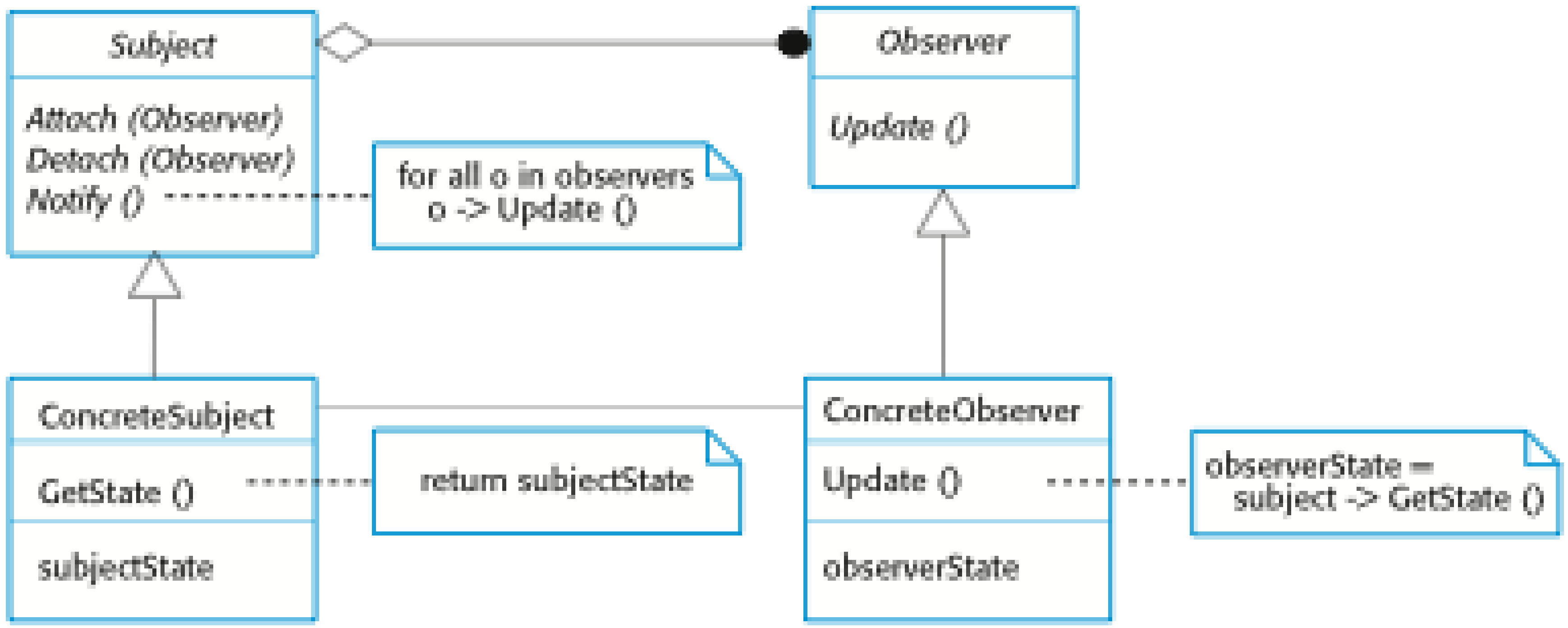
Multiple displays using the Observer pattern



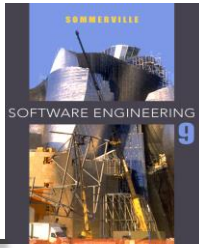


A UML model of the Observer pattern



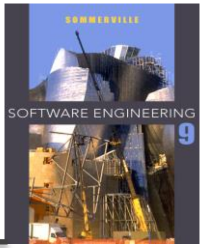


Design problems



- 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).

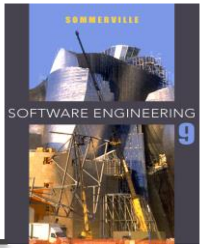
Implementation issues



✧ Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:

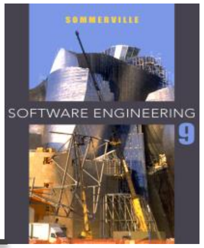
- Reuse Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
- Configuration management During the development process, you have to keep track of the many different versions of each software 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



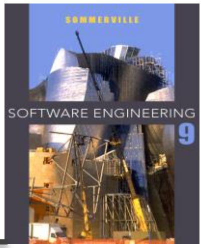
- ✧ From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- ✧ Costs and schedule pressure mean that this 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



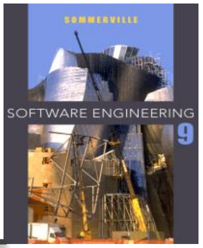
- ✧ The abstraction level
  - At this level, you don’t reuse software directly but use knowledge of successful abstractions in the design of your software.
- ✧ The object level
  - At this level, you directly reuse objects from a library rather than writing the code yourself.
- ✧ The component level
  - 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.

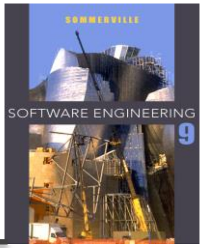
Configuration management



- ✧ Configuration management is the name given to the general process of managing a changing software system.
- ✧ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

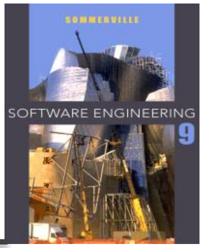
.

Configuration management activities



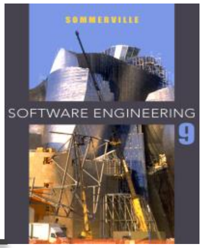
- ✧ Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- ✧ System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- ✧ Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Host-target development



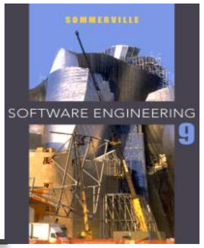
- ✧ Most software is developed on one computer (the host), but runs on a separate machine (the target).
- ✧ More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting 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.

Development platform tools



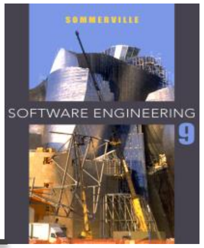
- An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- A language debugging system.
- Graphical editing tools, such as tools to edit UML models.
- Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
- Project support tools that help you organize the code for different development projects.

Integrated development environments (IDEs)



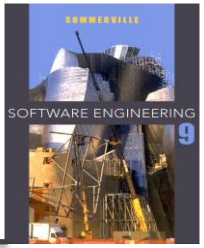
- ✧ Software development tools are often grouped to create an integrated development environment (IDE).
- ✧ An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- ✧ IDEs are created to support development in a specific 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



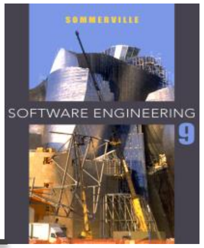
- ✧ If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- ✧ High availability systems may require components to be deployed on more than one platform. 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



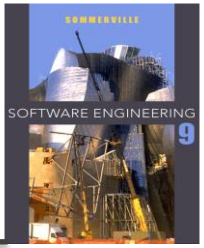
- ✧ Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- ✧ Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not 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.

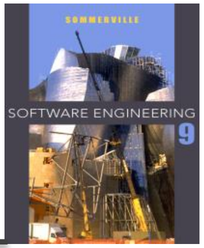
Open source issues



✧ Should the product that is being developed make use of open source components?

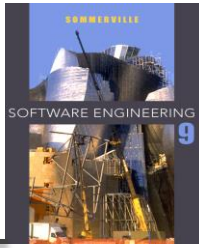
✧ Should an open source approach be used for the software's development?

Open source business



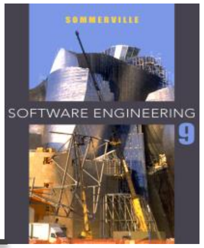
- ✧ More and more product companies are using an open source approach to development.
- ✧ Their business model is not reliant on selling a software product but on selling support for that product.
- ✧ They believe that involving the open source community will allow software to be developed more cheaply,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

Open source licensing



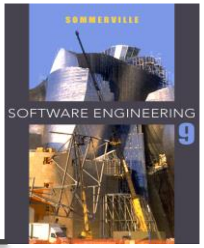
- Afundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
  - Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
  - Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
  - Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

License models

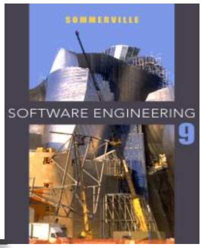


- ✧ The GNU General Public License (GPL). This is a so-called 'reciprocal' license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- ✧ The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
- ✧ The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

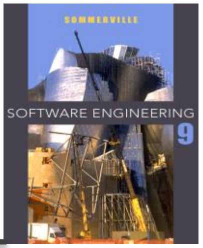
License management



- ✧ Establish a system for maintaining information about open-source components that are downloaded and used.
- ✧ Be aware of the different types of licenses and understand how a component is licensed before it is used.
- ✧ Be aware of evolution pathways for components.
- ✧ Educate people about open source.
- ✧ Have auditing systems in place.
- ✧ Participate in the open source community.

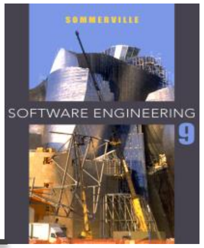


- **Object-oriented analysis (OOA)** looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed.
- Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt during object-oriented design (OOD).
- Analysis is done before the Design

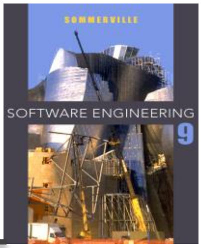


✧ The sources for the analysis can be a written requirements statement, a formal vision document, interviews with stakeholders or other interested parties.

✧ A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately. .



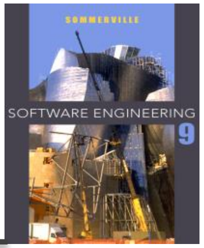
- ✧ The result of object-oriented analysis is a description of *what* the system is functionally required to do, in the form of a conceptual model.
- ✧ That will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock-up.
- ✧ The purpose of object oriented analysis is to develop a model that describes computer software as it works to satisfy a set of customer defined requirements.



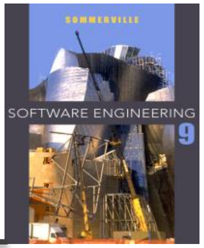
- **Object-oriented design (OOD)** transforms the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional - technological or environmental - constraints, such as transaction throughput, response time, run-time platform, development environment, or programming language.

- The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of *how* the system is to be built.

Key points

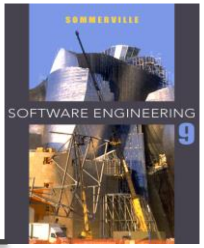


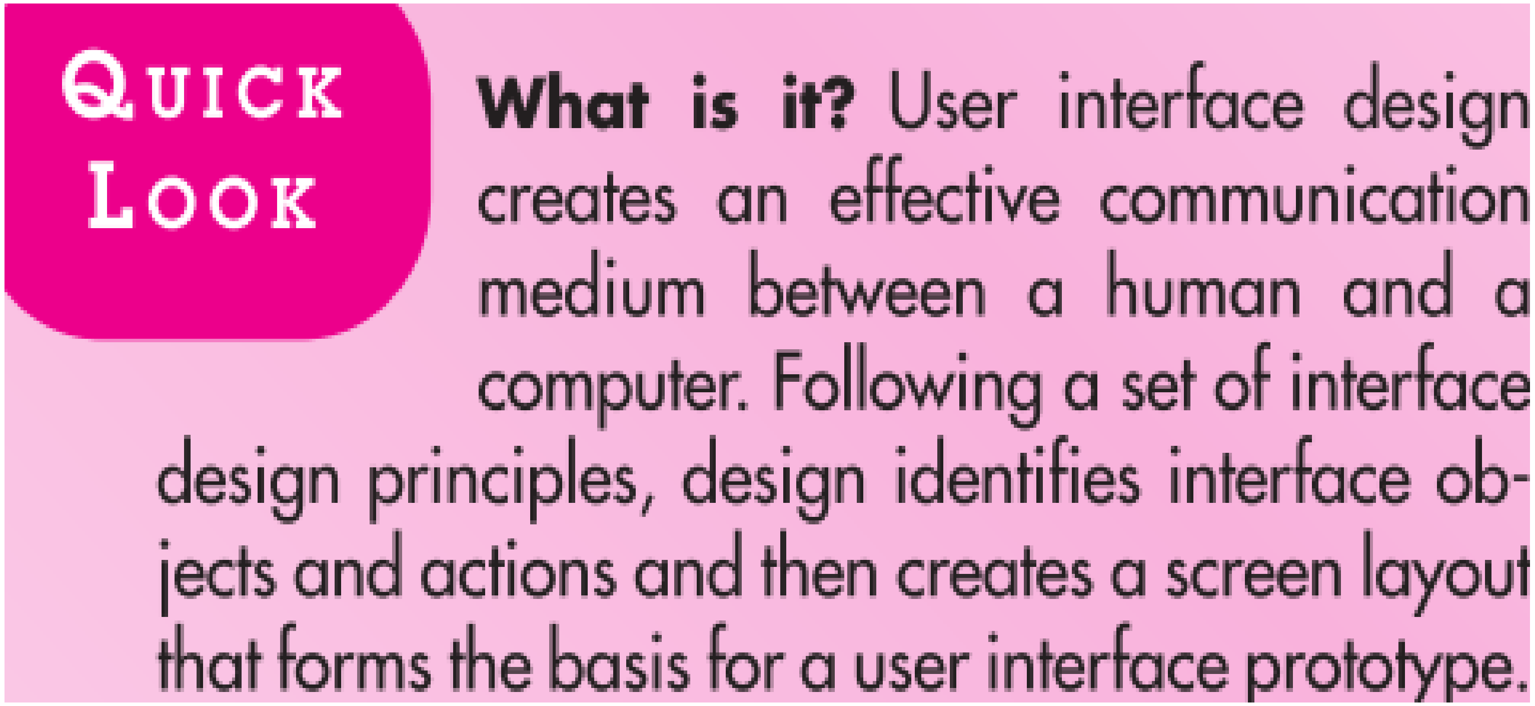
- When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating 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.



✧ The purpose of design is to specify a working solution that can be easily translated into a programming language code.

User Interface Design

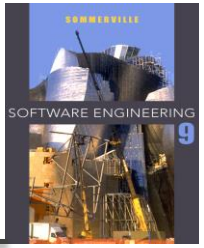




Quick Look

What is it? User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

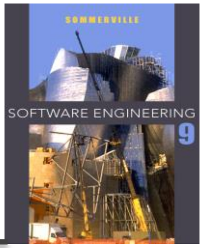
THE GOLDEN RULES
Theo Mandel [Man97] coins three golden rules:



1. Place the user in control.
2. Reduce the user’s memory load.
3. Make the interface consistent.

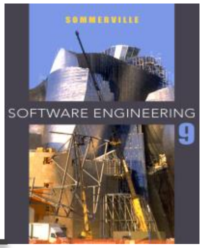
These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

interaction mechanisms:



A set of interaction mechanisms were defined to enable software designers to build systems that properly implemented the golden rules.

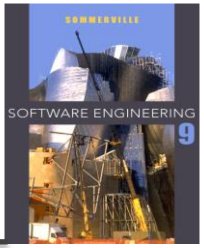
These interaction mechanisms, collectively called the graphical user interface (GUI), have eliminated some of the most egregious problems associated with human interfaces.



✧ But even in a “Windows world,” we all have encountered user interfaces that are difficult to learn, difficult to use, confusing, counterintuitive, unforgiving, and in many cases, totally frustrating.

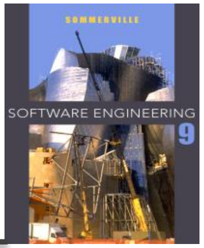
Yet, someone spent time and energy building each of these interfaces, and it is not likely that the builder created these problems purposely.

1. Place the user in control.



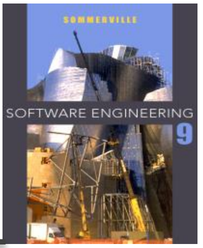
- ✧ Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- ✧ Provide for flexible interaction.
- ✧ Allow user interaction to be interruptible and undoable.
- ✧ Streamline interaction as skill levels advance and allow the interaction to be customized.
- ✧ Hide technical internals from the casual user.
- ✧ Design for direct interaction with objects that appear on the screen.

2.Reduce the user’s memory load.



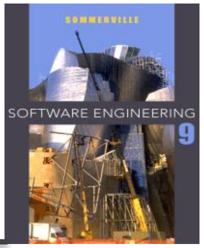
- ✧ Reduce demand on short-term memory.
- ✧ Establish meaningful defaults.
- ✧ Define shortcuts that are intuitive.
- ✧ The visual layout of the interface should be based on a real-world metaphor.
- ✧ Disclose information in a progressive fashion.

3.Make the interface consistent.



- ✧ Allow the user to put the current task into a meaningful context.
- ✧ Maintain consistency across a family of applications
- ✧ If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

USER INTERFACE ANALYSIS AND DESIGN



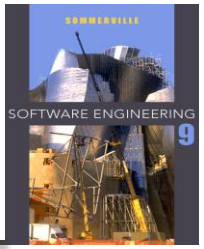
✧ Interface Analysis and Design Models:

A human engineer (or the software engineer) establishes a **user model.**

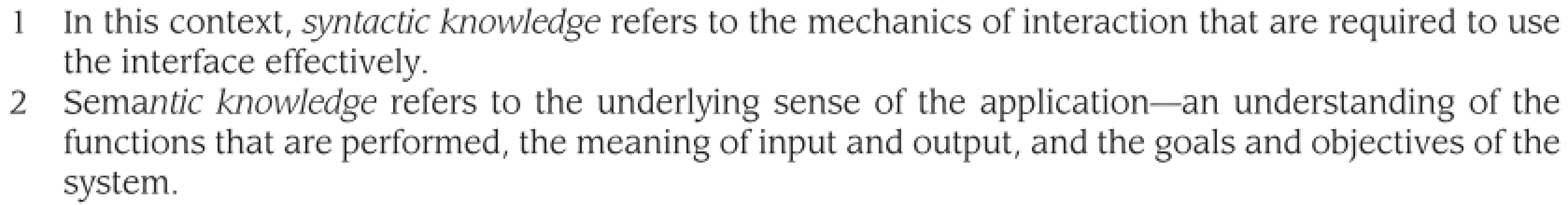
The software engineer creates a **design model,**

The end user develops a mental image that is often called the user’s **mental model or the system perception,**

The implementers of the system create an **implementation model.**



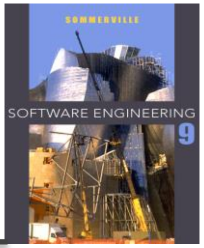
- To build an effective user interface, “all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality”. In addition, users can be categorized as:
- Novices.
- Knowledgeable, intermittent users.
- Knowledgeable, frequent users.

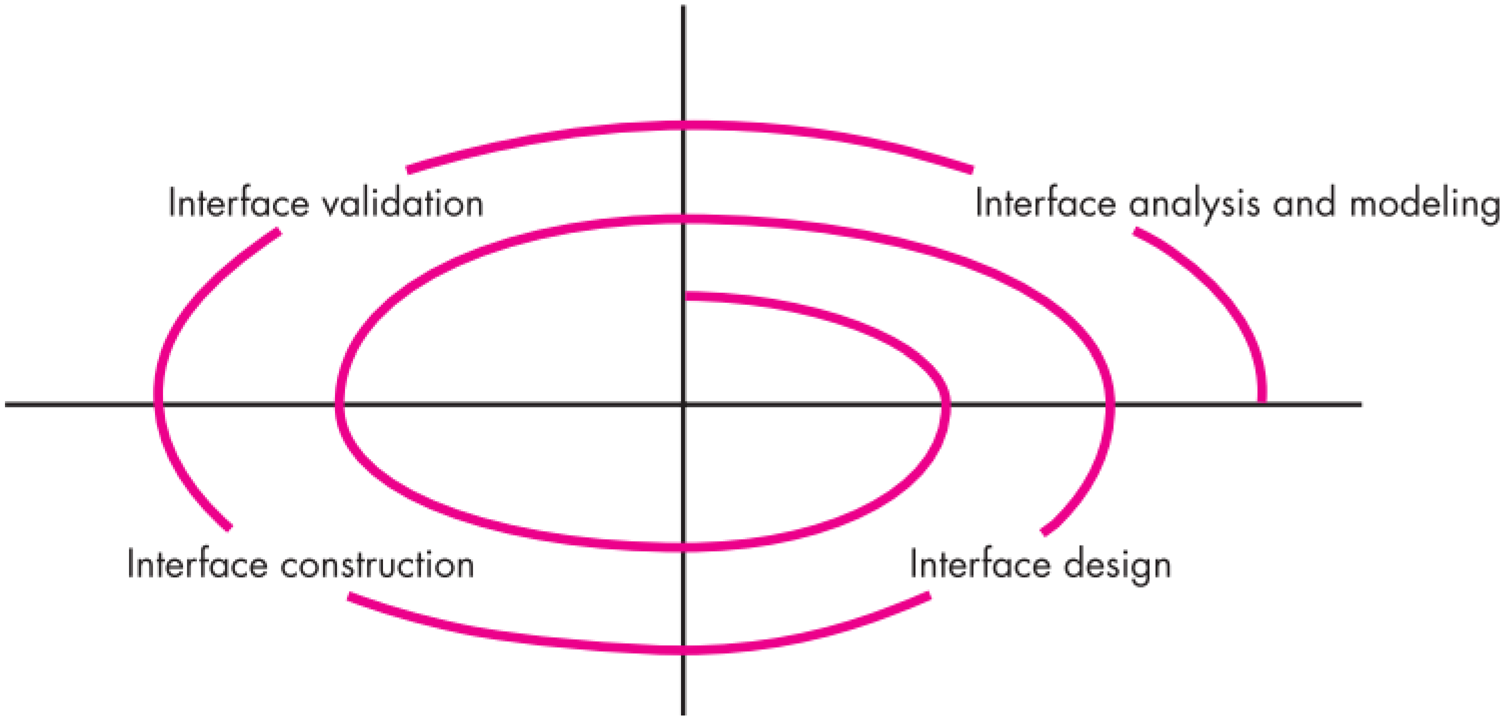


1 In this context, *syntactic knowledge* refers to the mechanics of interaction that are required to use the interface effectively.

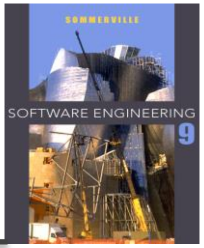
2 Sem*antic knowledge* refers to the underlying sense of the application—an understanding of the functions that are performed, the meaning of input and output, and the goals and objectives of the system.

The user interface design process



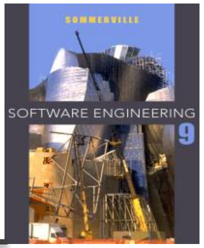


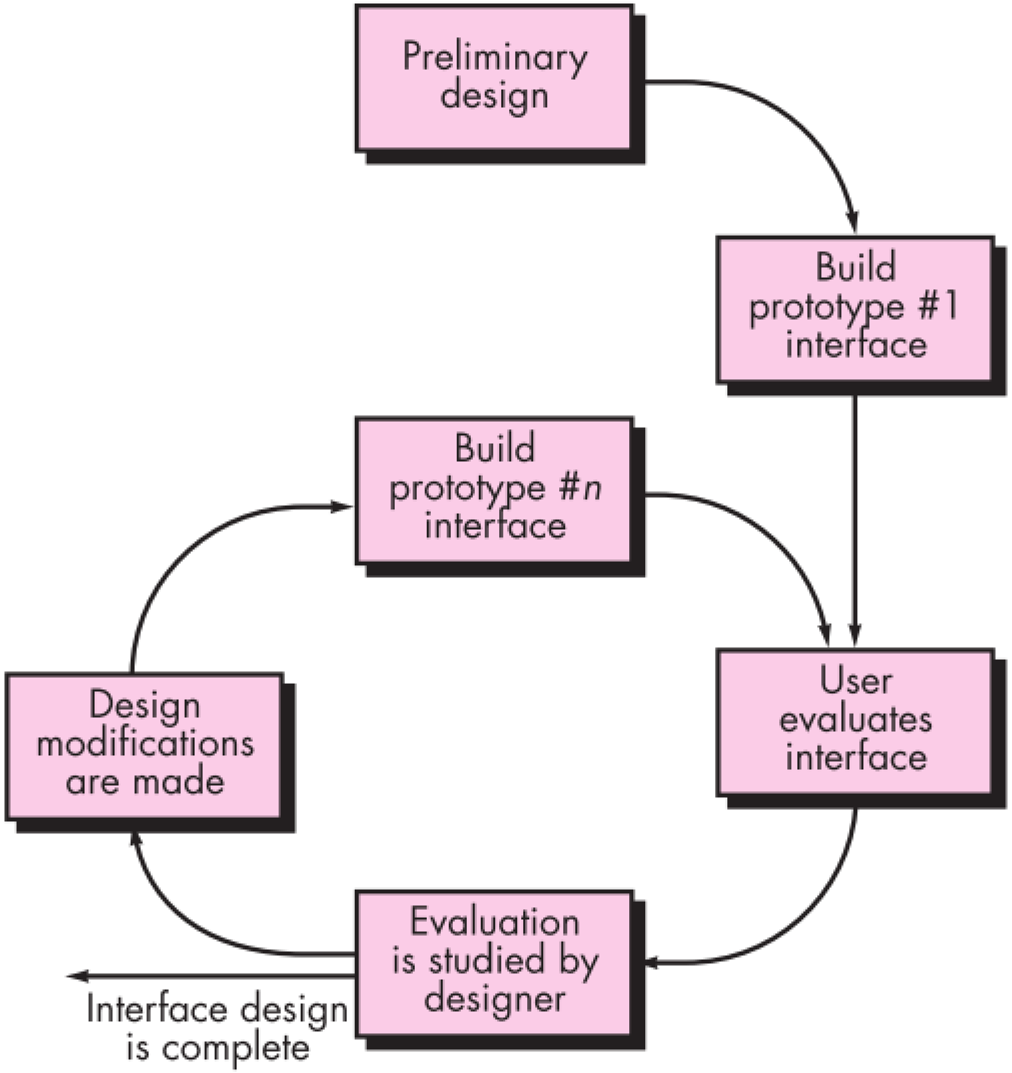
INTERFACE DESIGN STEPS



1. Using information developed during interface analysis (Section 11.3), define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

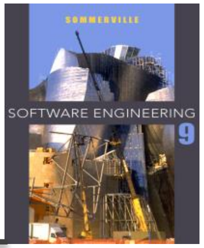
The interface design evaluation cycle





Preliminary design
Build prototype #1 interface
Build prototype #*n* interface
User evaluates interface
Design modifications are made
Evaluation is studied by designer
Interface design is complete

Tools



✧ LegaSuite GUI, developed by Seagull Software

✧ MotifCommonDesktopEnvironment, developed by The Open Group (www.osf.org/tech/desktop/cde/),

✧ Altia Design 8.0, developed by Altia (www.altia.com),