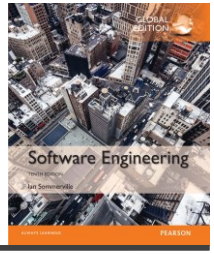
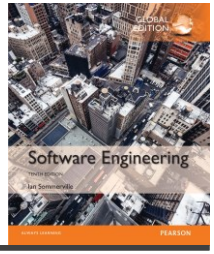


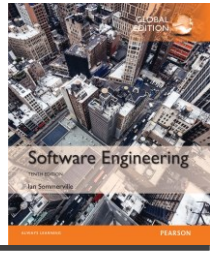
Chapter 7 – Design and Implementation

Topics covered



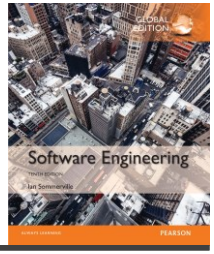
- ✧ Object-oriented design using the UML
- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open source development

Design and implementation

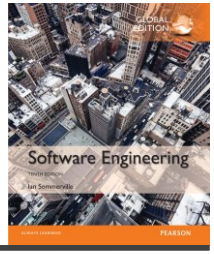


- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- Software design and implementation activities are invariably inter-leaved.
  - Software design is a creative activity in which you identify software components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

Build or buy

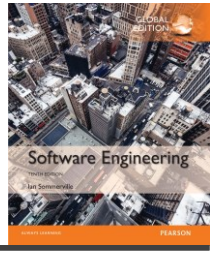


- In a wide range of domains, it is now possible to **buy** off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
  - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster 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.



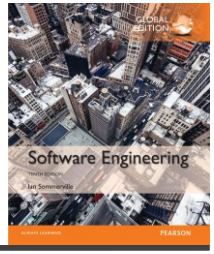
Object-oriented design using the UML

An object-oriented design process



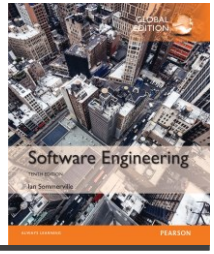
- Structured object-oriented design processes involve developing **a number of different system models**.
- They require a lot of effort for development and maintenance of these **models** and, for small systems, this may not be cost-effective.
- However, for large systems developed by different groups **design models** are an important communication mechanism.
- There is often a clear **mapping** between real-world entities and their controlling objects in the system. This improves the understandability, and hence the maintainability, of the design.

Process stages



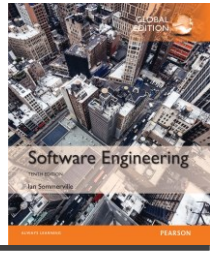
- There are a variety of different object-oriented design processes that depend on the **organization** using the process.
- Common activities in these processes include:
  - Define the **context** and **modes of use** of the system;
  - Design the **system architecture**;
  - Identify the principal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

Process stages



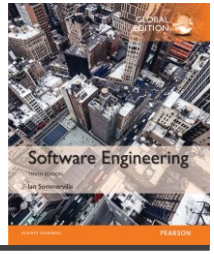
- Like all creative activities, design is *not* a clear-cut, *sequential* process.
- You develop a design by **getting ideas**, **proposing solutions**, and **refining** these solutions *as information become available*. You inevitably have to **backtrack** and **retry** when problems arise.
- Sometimes you explore options in detail to see if they work; at other times you ignore details until late in the process.
- Process illustrated here using a design for a wilderness weather station.

System context and interactions



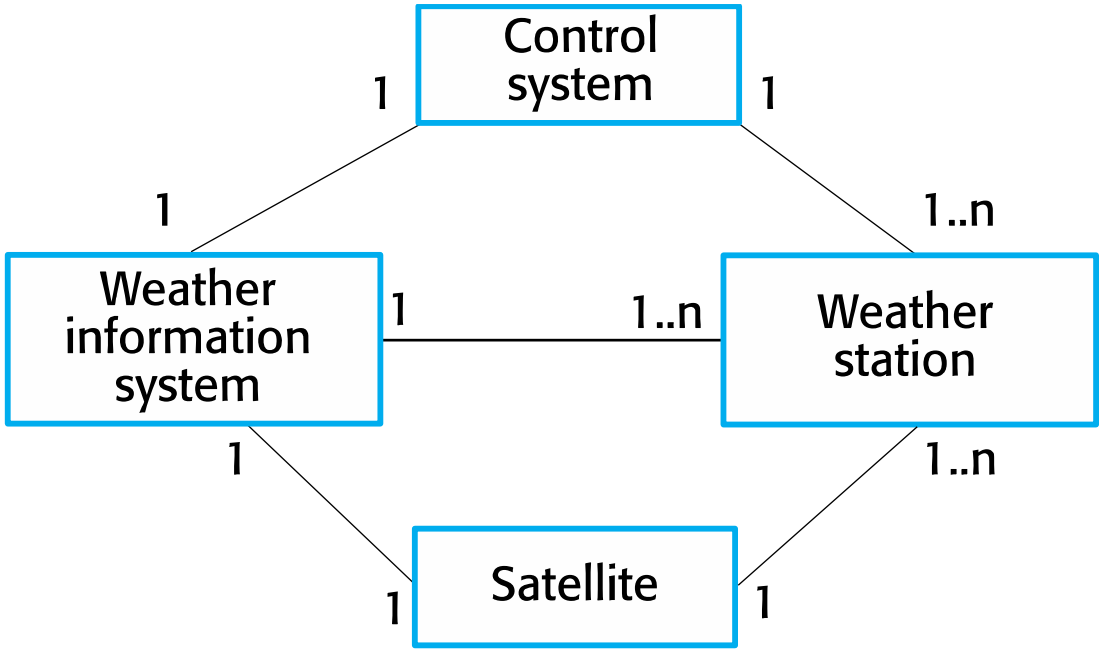
- Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system **functionality** and how to structure the system to **communicate** with its environment.
- Understanding of the **context** also lets 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

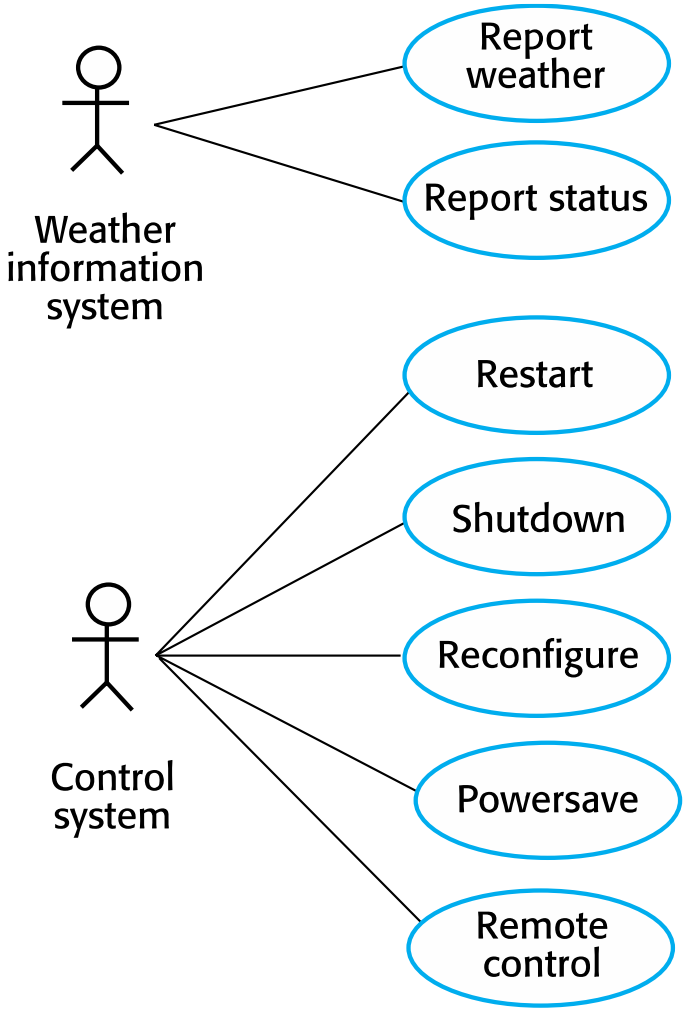


- System context models and interaction models present complementary views of the relationships between a system and its environment
- A system context model is a **structural** model that demonstrates the other systems in the environment of the system being developed. (represented with a simple block diagram)
- An interaction model is a **dynamic** model that shows how the system interacts with its environment as it is used. (represented with a use case model)
  - The **use case description** can help designers identify objects (also operations) in the system and gives them an understanding of what the system is indented to do

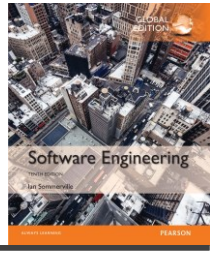
System context for the weather station



Weather station use cases



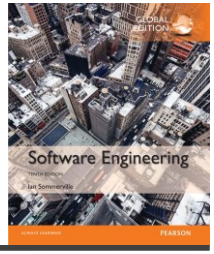
Use case description—Report weather



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.

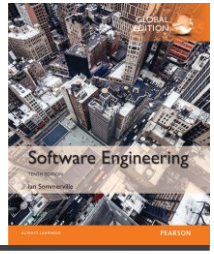
The description should clearly identify **what information** is exchanged, how the **interaction** is initiated, and so on.

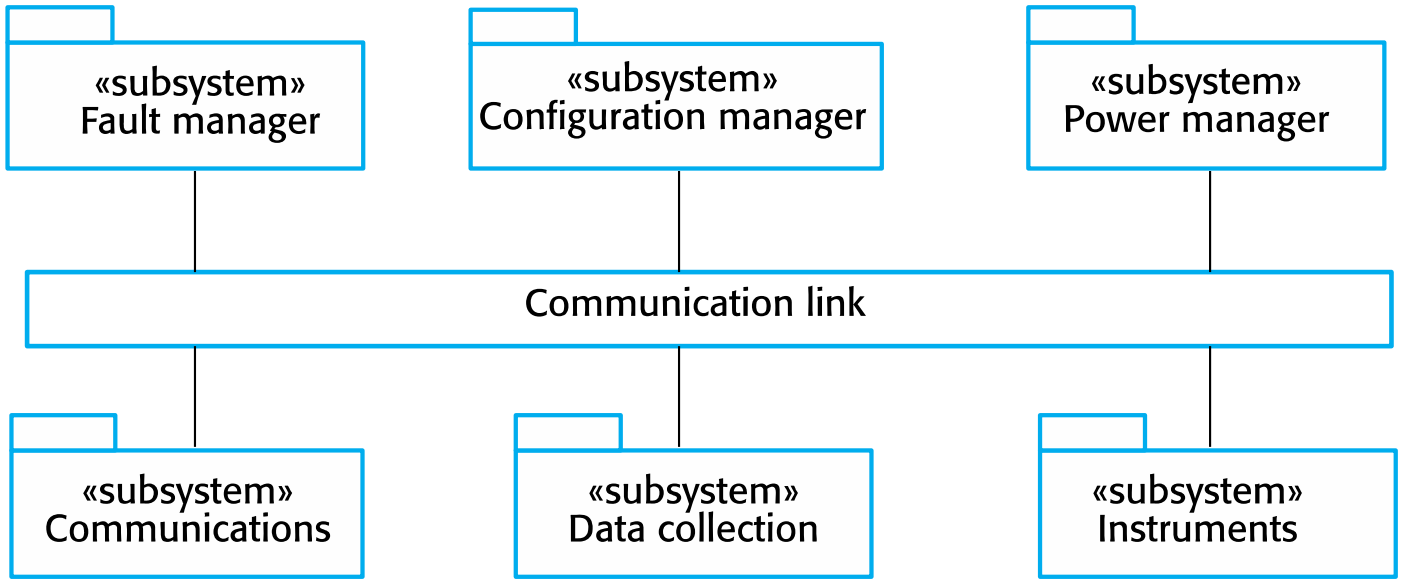
Architectural design



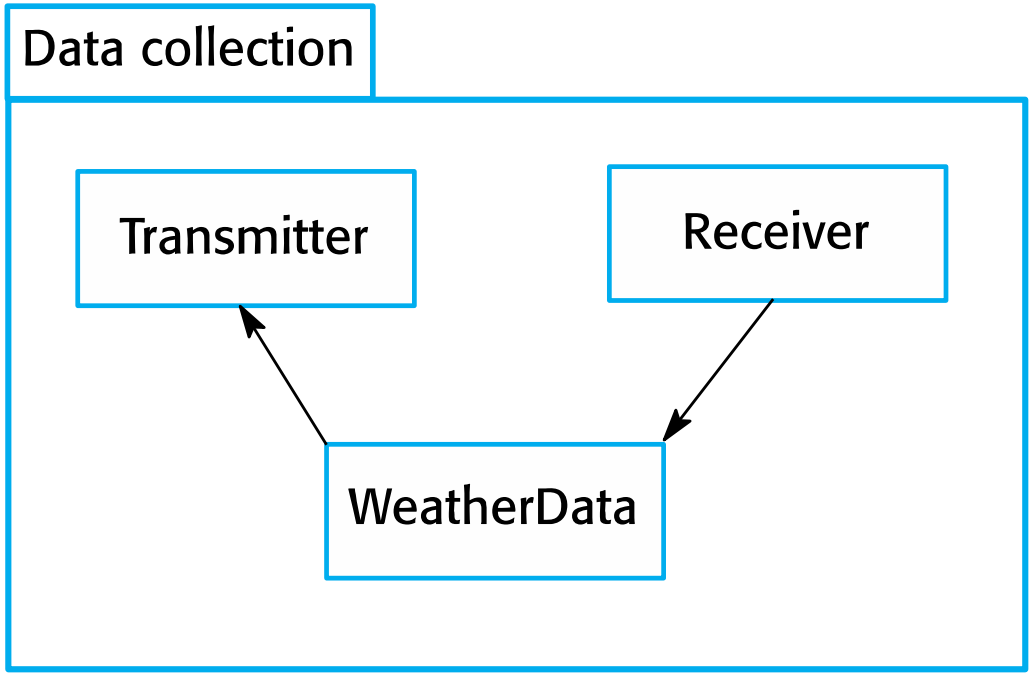
- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such 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

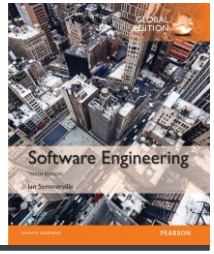




Architecture of data collection system

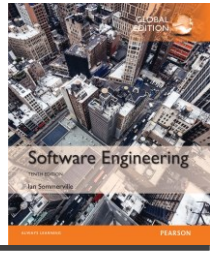


Object class identification



- Identifying object classes is often 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.
- **Object classes**, **attributes**, and **operations** that are initially identified from the informal system description can be a starting point for the design
- Further information from application domain knowledge or scenario analysis may be used to refine and extend the initial objects
  - This information can be collected from requirements documents, discussions with users, or from analyses of existing systems

Approaches to identification

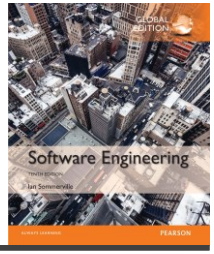


- Use a **grammatical approach** based on a natural language description of the system (used in Hood OOD method).
  - Objects and attributes are nouns; operations and services are verbs
- Base the identification on **tangible things** in the application domain.
  - Such as things, roles, events, interactions, locations, organizational units, and so on
- Use a **behavioural approach** and identify objects based on ***what participates in what behaviour***.
  - The various behaviours are assigned to different parts of the system and an understanding is derived of **who** initiates and participates in these **behaviours**. **Participants** who play significant **roles** are recognised as **objects**
- Use a **scenario-based analysis**. The **objects**, **attributes** and **methods** in each scenario are identified.
  - **Class-Responsibilities-Collaborators** (CRC) card

Class Name	***Collaborators*** • Partner • Components
Responsibility • Operations may go across several lines.	

https://www.oreilly.com/library/view/pattern-oriented-software-architecture/9781118725269/OEBPS/9781118725269_not.htm

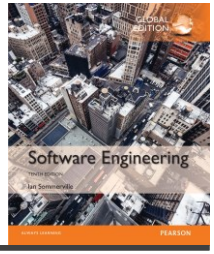
Weather station description



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.

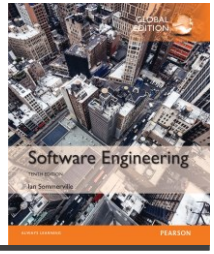
When a command is issued to transmit the **weather data**, the weather station processes and summarises the collected data. The **summarised data** is transmitted to the mapping computer when a request is received.

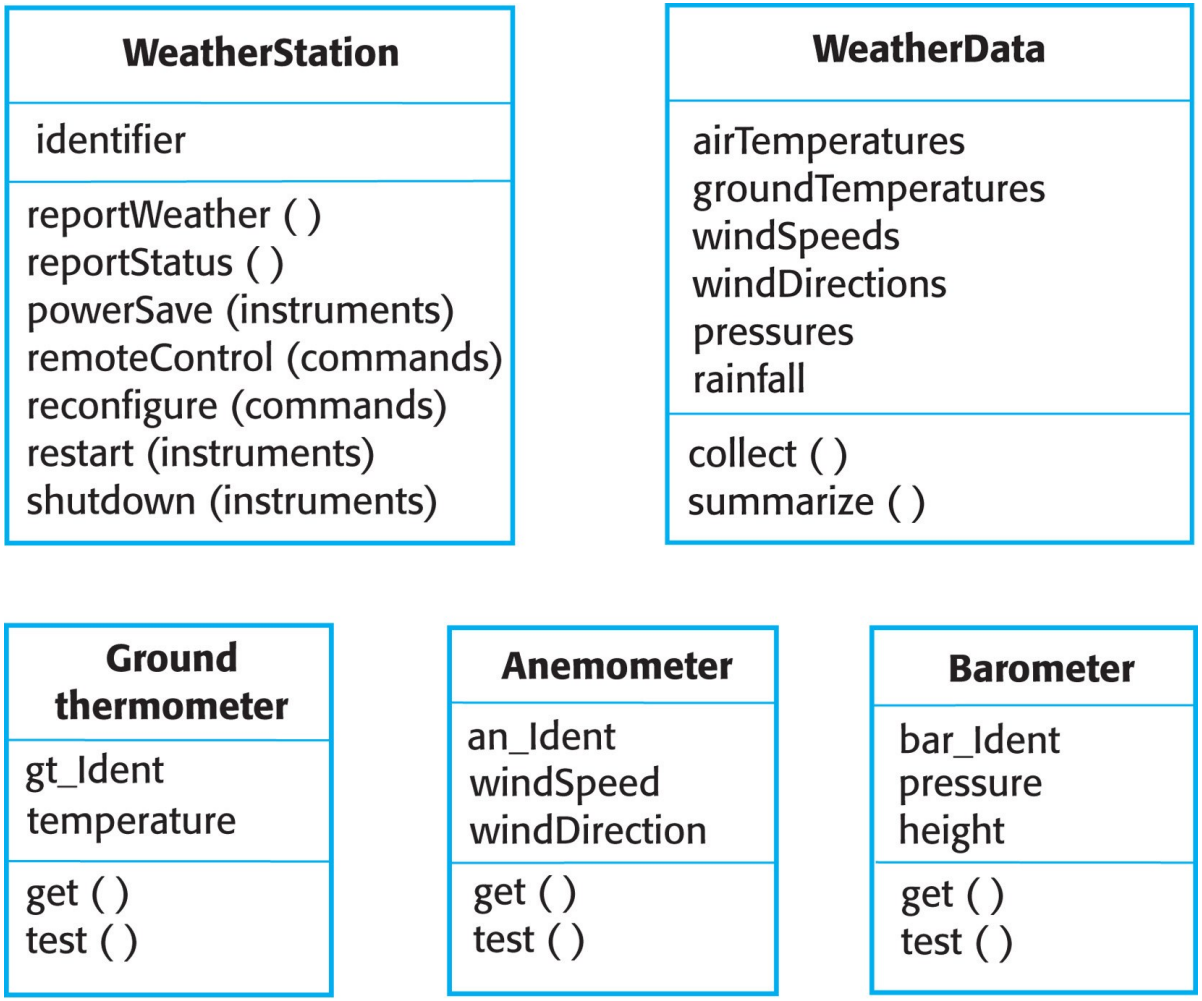
Weather station object classes



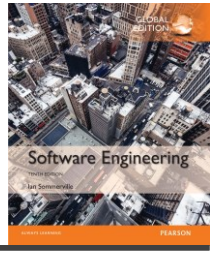
- Object class identification in the weather station system may be based on the **tangible** **hardware** and **data** in the system:
  - Ground thermometer, Anemometer, Barometer
    - Application domain objects that are 'hardware' objects related to the instruments in the system.
  - Weather station
    - 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



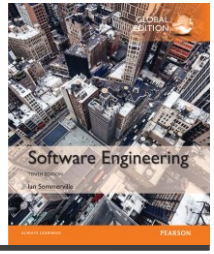


Design models



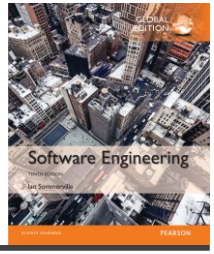
- Design models show the objects and object classes and relationships between these entities.
- There are two kinds of design model:
  - Structural (static) models describe the **static structure** of the system in terms of object classes and relationships.
  - Dynamic models describe the **dynamic interactions** between objects.
- An important step in the design process is to decide on the design models that you need (depending on the nature of the system) and the level of detail required in these models (link and bridge between requirements engineers, designers, and programmers).

Examples of design models



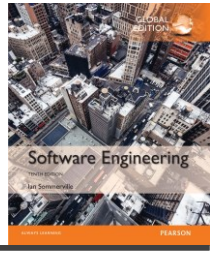
- Subsystem models that show logical groupings of objects into coherent subsystems.
- Sequence models that show the sequence of object interactions.
- State machine models that show how individual objects change their state in response to events.
- Other models include use-case models, aggregation models,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

Subsystem models



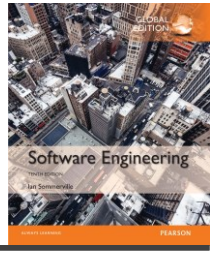
- **Subsystem model** shows how the design is organised into logically related groups of objects.
- In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.
- As well as subsystem models, you may also design detailed **object models**, showing all of the objects in the systems and their associations (inheritance, generalization, aggregation, etc.).

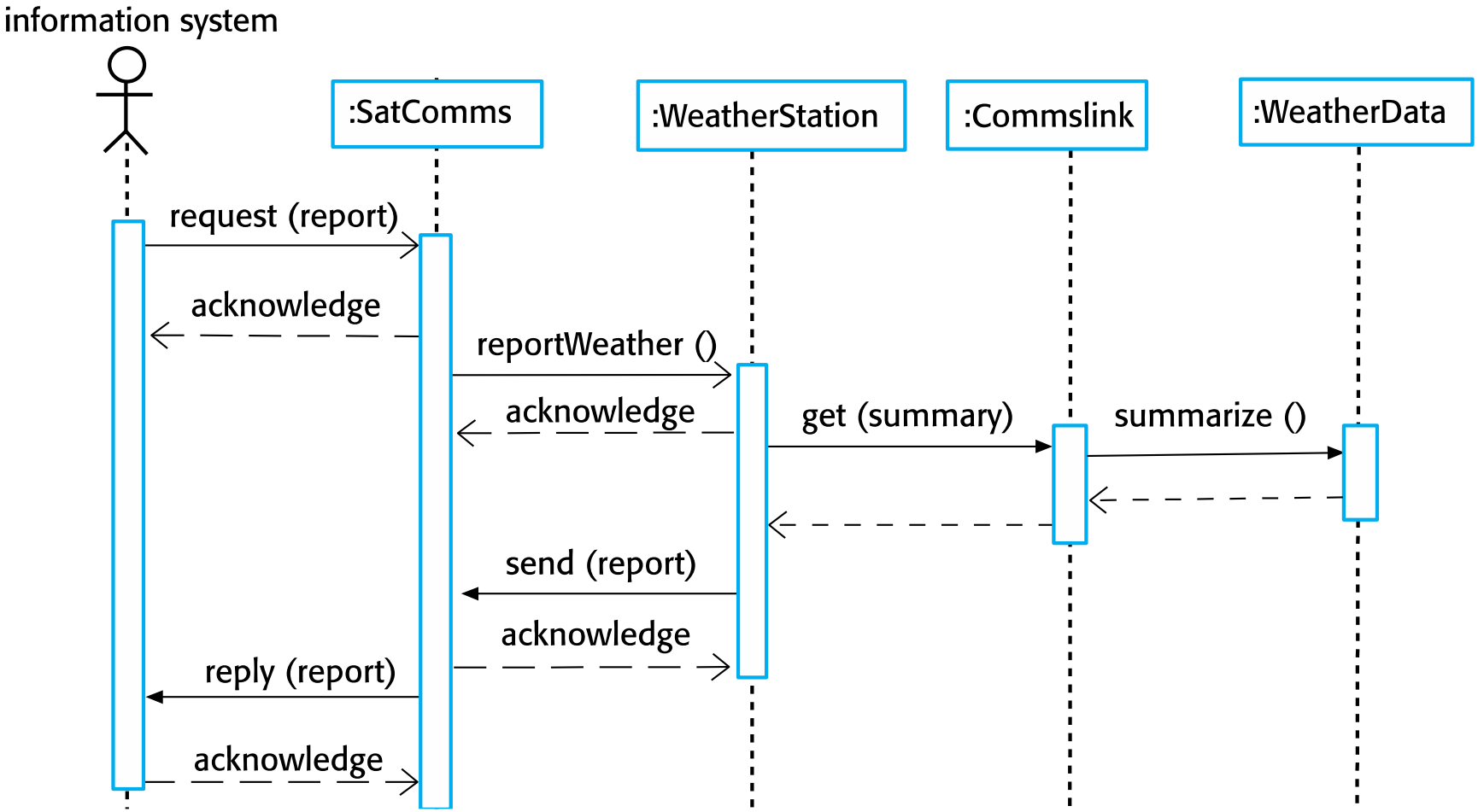
Sequence models



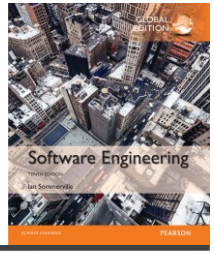
- **Sequence models** show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read top to bottom;
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.
- There should be **a sequence model for EACH use case** that you have identified.

Sequence diagram describing data collection



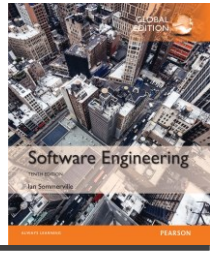


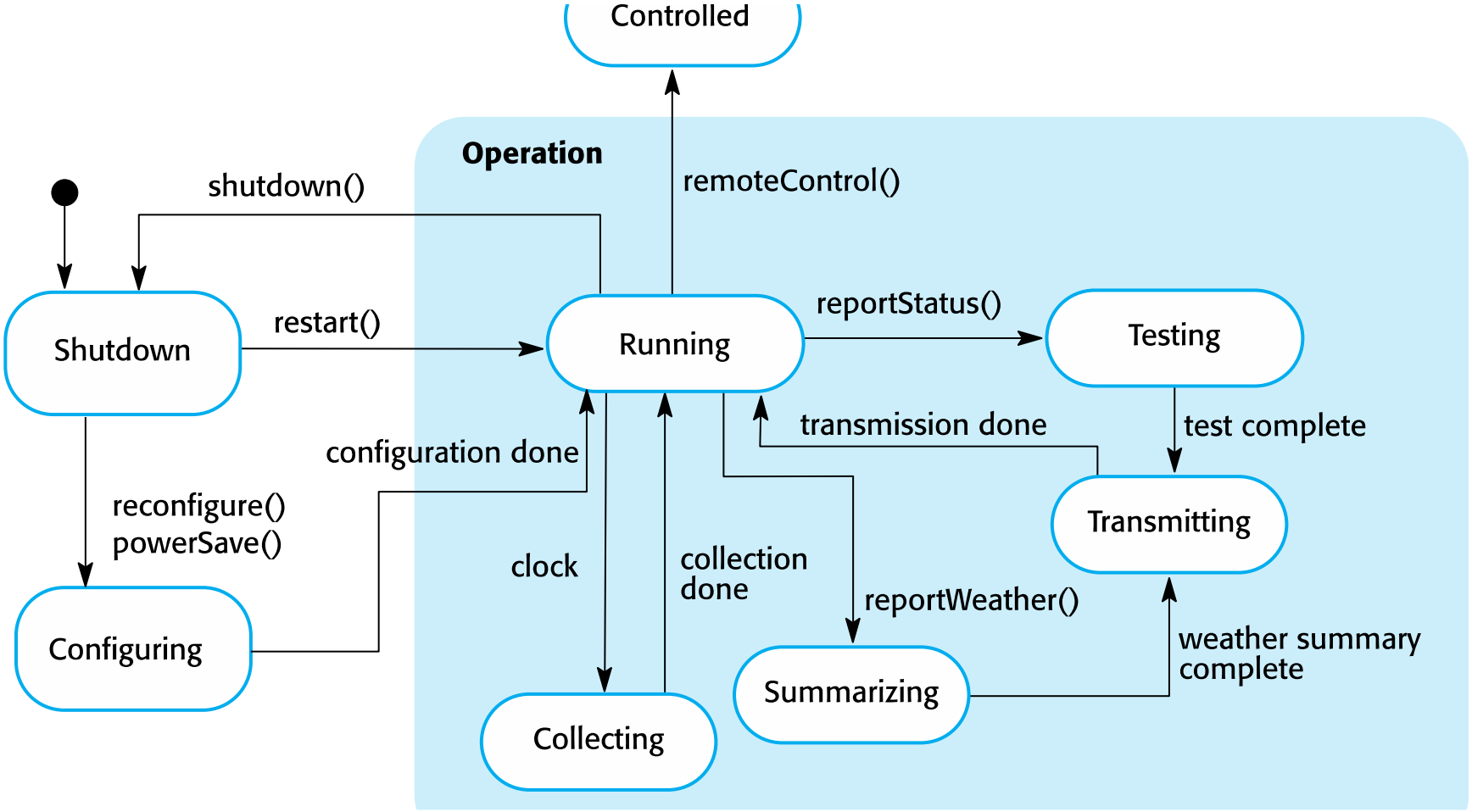
State diagrams



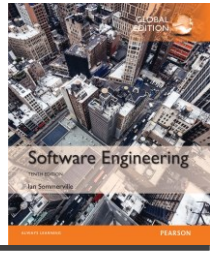
- **State diagrams** are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- State diagrams are useful high-level models of a system or an object's run-time behavior (i.e., operation).
- You don't usually need a state diagram for all 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



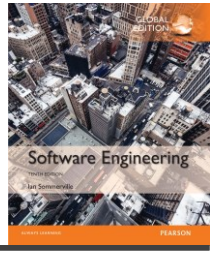


Interface specification



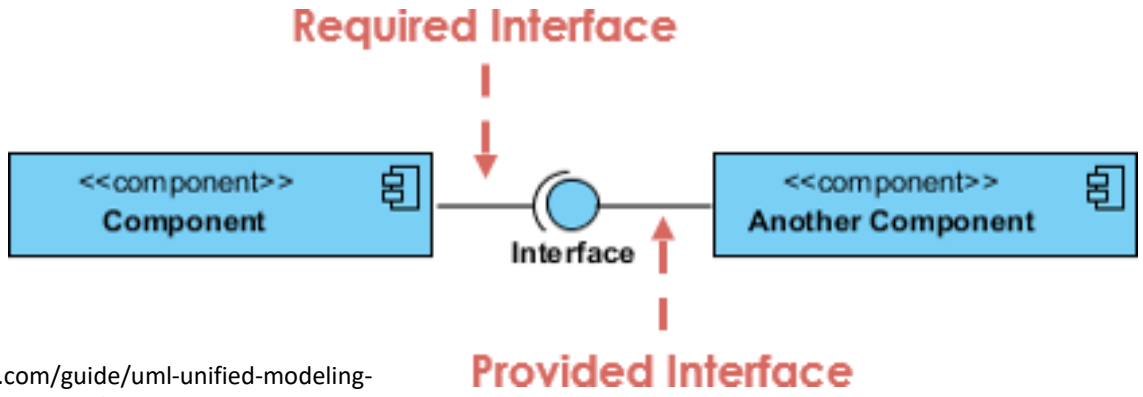
- **Object interfaces** have to be specified so that the objects and other components can be **designed in parallel**.
- Interface design is concerned with specifying the **detail of the interface** to an object or to a group of objects
  - This means defining the signatures and semantics of the services that are provided by the object or by a group of objects
- Designers should avoid designing the detail data representation (i.e., such as attributes) in the interface design 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



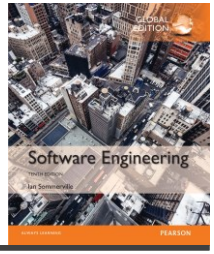
«interface» Reporting
weatherReport (WS-Ident): Wreport statusReport (WS-Ident): Sreport

«interface» Remote Control
startInstrument(instrument): iStatus stopInstrument (instrument): iStatus collectData (instrument): iStatus provideData (instrument): string



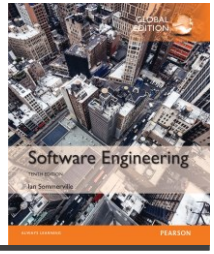
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/

SOLID Design Principle
by Robert Martin



- **S**ingle Responsibility Principle (SRP)
  - each class has to have one single purpose, a responsibility and a reason to change (*a class should have only a single responsibility*)
- **O**pen Closed Principle (OCP)
  - a class should be open for extension, but closed for modification
- **L**iskov substitution Principle (LSP)
  - the derived class should always extend its parent class, without changing its behavior at all (*subtypes must be substitutable for their base types or subclasses must fulfil a contract defined by the base class*)
- **I**nterface Segregation Principle (ISP)（role interfaces）
  - no client should be forced to depend on methods it does not use
  - "many **client-specific** interfaces are better than one **general-purpose** interface."
- **D**ependency Inversion Principle (DIP)
  - one should "depend upon abstractions, not concretions"

SOLID principles



SOLID Principles... Contribute to cl

Single Responsibility Principle: A class should have only one responsibility

Open / Closed Principle: A software module (can be a class or method) should be open for extension but closed for modification

Liskov Substitution Principle: Parent classes should be easily substituted with their child classes without blowing up the application.

Interface Segregation Principle: Make fine grained interfaces that are client specific. Clients should not be forced to depend upon interfaces that they do not use.

Dependency Inversion Principle: High-level modules should not depend on low-level modules, both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

http://krishnamurtypammi-technology.blogspot.com/p/solid-principles.html

Single Resposibility Principle
A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

Open / Closed Principle
A software module (it can be a class or method) should be open for extension but closed for modification.

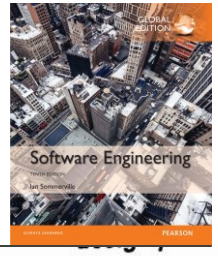
Liskov Substitution Principle
Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Interface Segregation Principle
Clients should not be forced to depend upon the interfaces that they do not use.

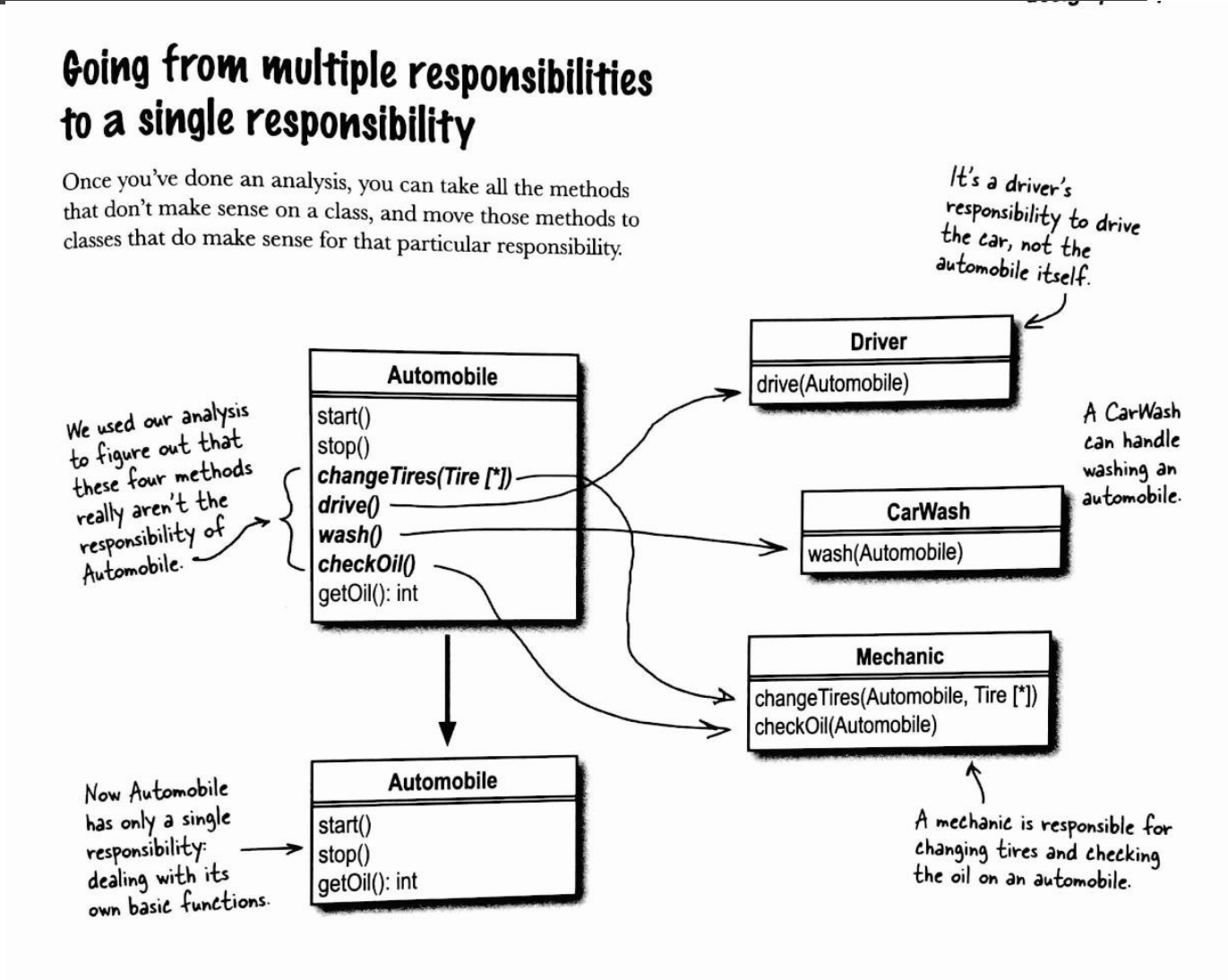
Dependency Inversion Principle
Program to an interface, not to an implementation.

https://devopedia.org/solid-design-principles

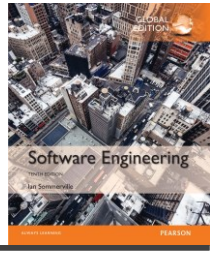
Practice is the key



- Just knowing the principles will not turn a bad programmer into a good one.
- This means that programmers need to understand why these principles make sense.
- They need to apply them with judgement. If they see code that violates these principles, they must try to see if the violations can be justified. If not, they can apply one or more principles to improve the code.
- Practice is the key.



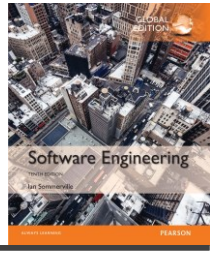
Symptoms of Rotting Design
by Robert Martin



- Some symptoms of rotting design due to improperly managed **dependencies** across modules:
  - **Rigidity:** Implementing even a small change is difficult since it's likely to translate into a cascade of changes.
  - **Fragility:** Any change tends to break the software in many places, even in areas not conceptually related to the change.
  - **Immobility:** We're unable to reuse modules from other projects or within the same project because those modules have lots of dependencies.
  - **Viscosity (黏度):** When code changes are needed, developers will prefer the easier route even if they break existing design.
- **Antipatterns** and improper understanding of design principles can lead to **STUPID** code: **Singleton**, **Tight Coupling**, **Untestability**, **Premature Optimization**, **Indescriptive Naming**, and **Duplication**. SOLID can help developers stay clear of these

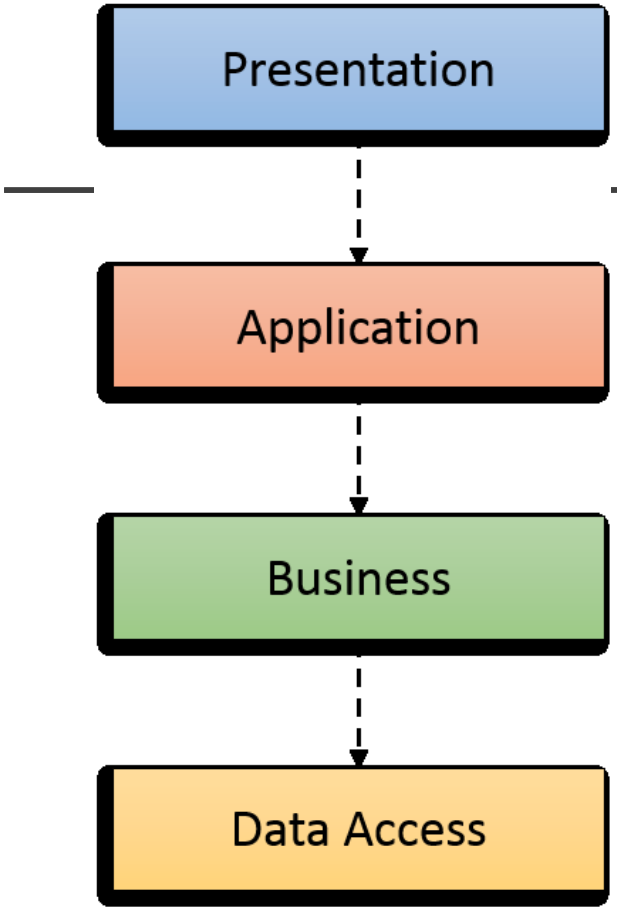
https://devopedia.org/solid-design-principles

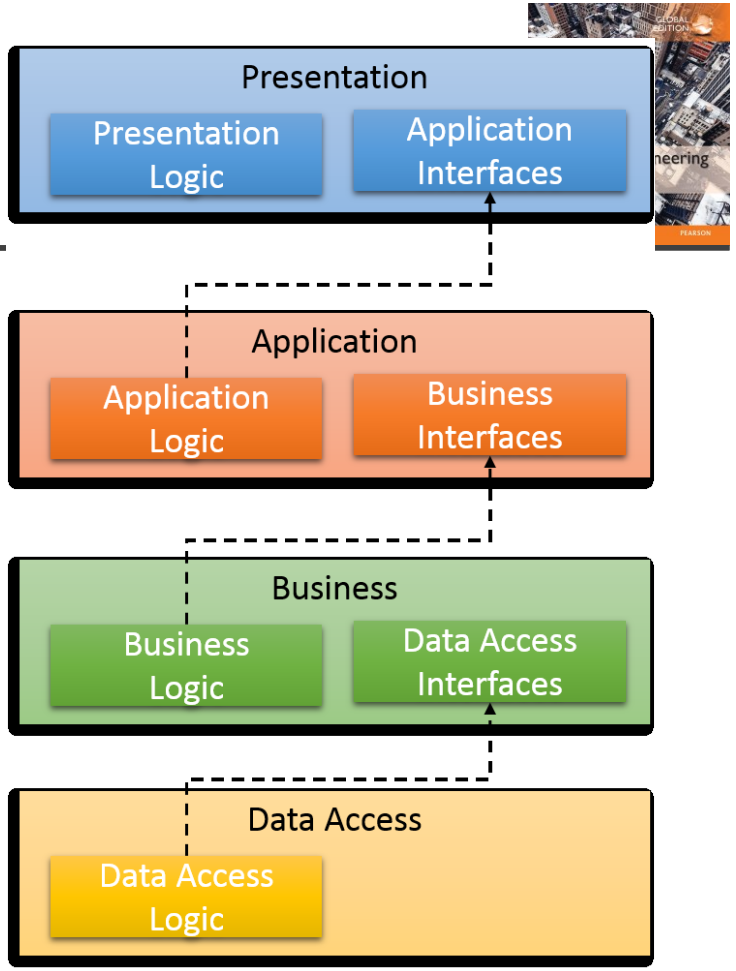
DIP (Dependency Inversion Principle)



- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

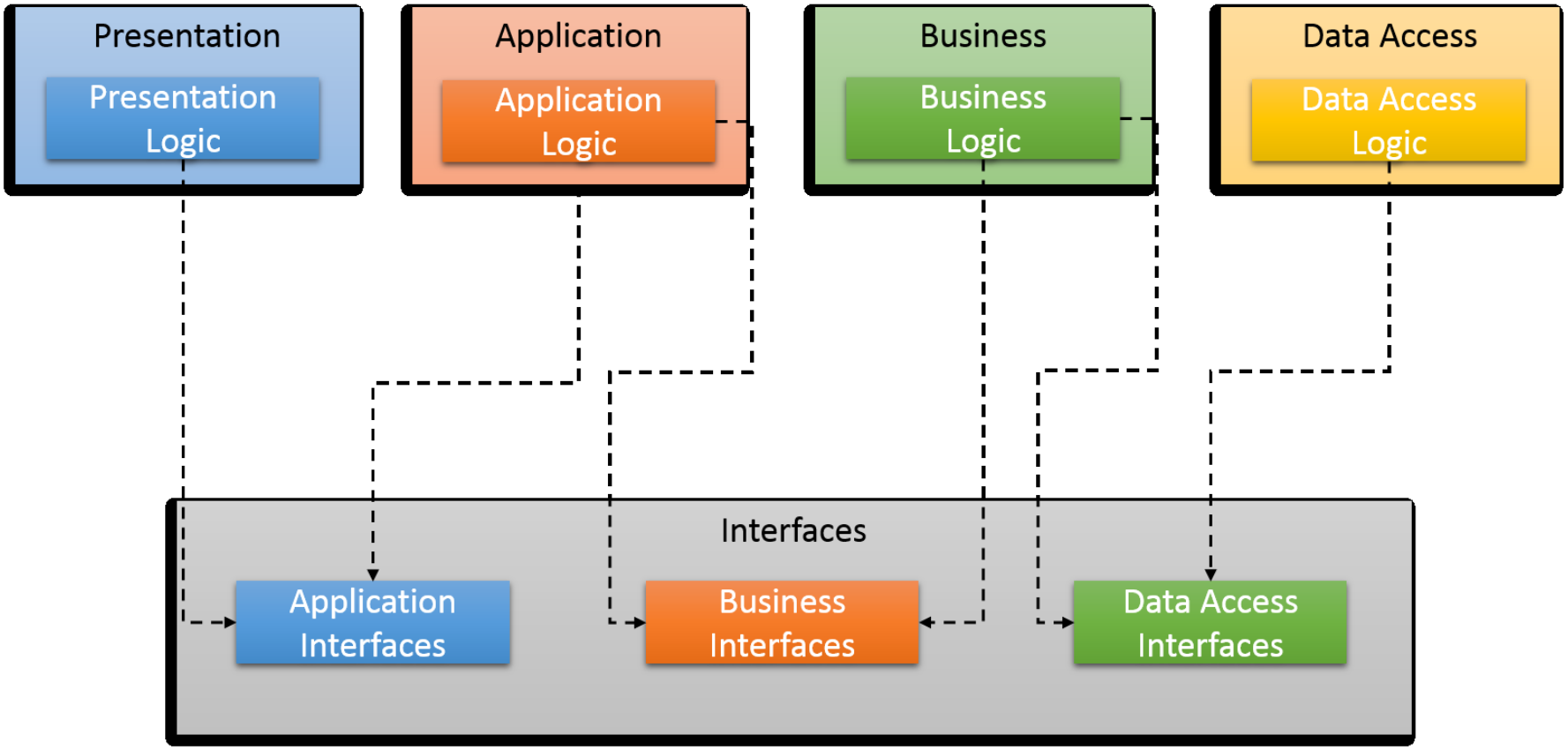
- https://en.wikipedia.org/wiki/Dependency_inversion_principle





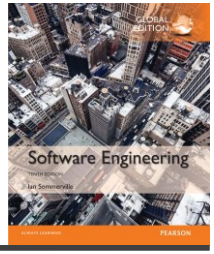
In traditional architecture “Higher” level modules depend on “lower” level modules

When the Dependency Inversion Principle is applied this relationship is reversed. The higher layer defines the abstractions it needs to do its job and the lower layers implement those abstractions



abstractions should not depend upon details rather details should depend upon abstractions

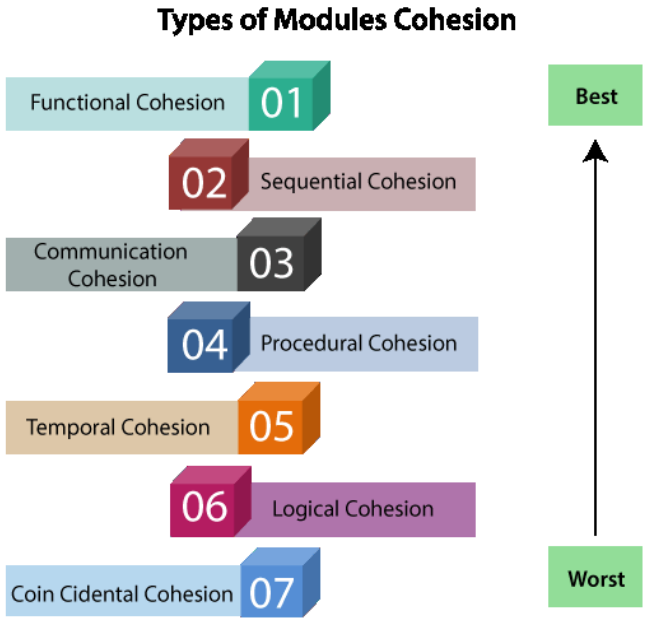
Cohesion



- **Cohesion** (the greater the cohesion, the better is the program design)
  - is a measure that defines the degree of intra-dependability within elements of a module.
- Types of cohesion
  - **Co-incidental cohesion** - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
  - **Logical cohesion** - When logically categorized elements are put together into a module, it is called logical cohesion.
  - **Temporal Cohesion** - When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.

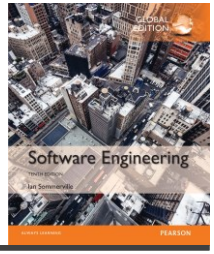
Cohesion

- **Procedural cohesion** - When elements of module are grouped together, which are executed sequentially in order to perform a task, it is callec procedural cohesion.
- **Communicational cohesion** - When elements c module are grouped together, which are execute sequentially and work on same data (information it is called communicational cohesion.
- **Sequential cohesion** - When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- **Functional cohesion** - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.



https://www.javatpoint.com/software-engineering-coupling-and-cohesion

Coupling

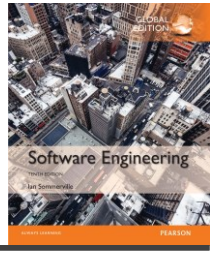


- **Coupling** (the lower the coupling, the better the program)
  - A measure that defines the level of inter-dependability among modules of a program.
  - It tells at what level the modules interfere and interact with each other.

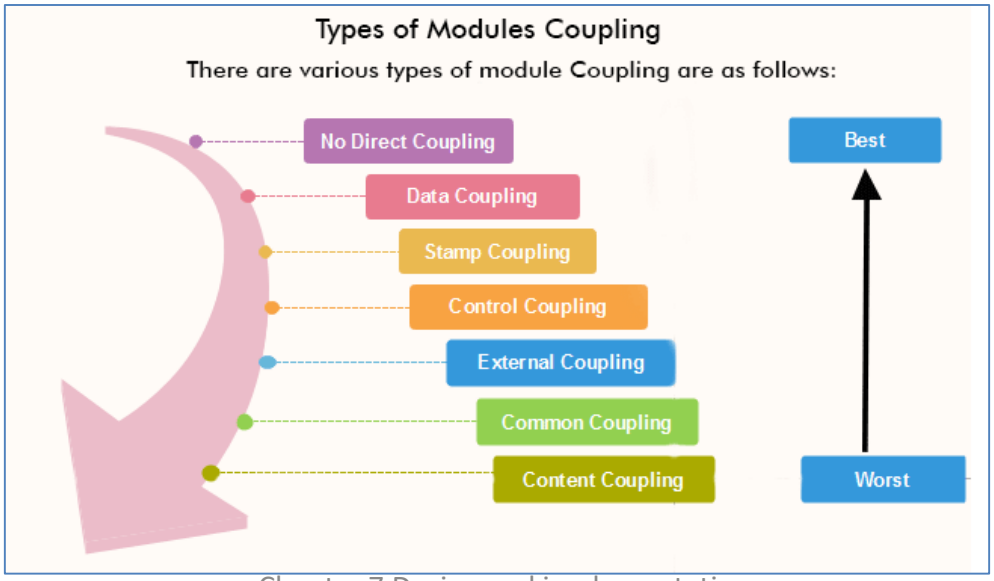
- Types of coupling
  - **Content coupling** - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
  - **Common coupling** - When multiple modules have read and write access to some global data, it is called common or global coupling.
  - **Control coupling** - Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.

https://en.wikipedia.org/wiki/Coupling_(computer_programming)

Coupling

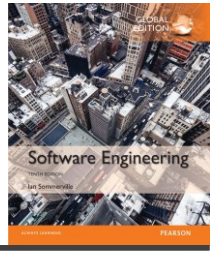


- **External coupling** - When two modules share an externally imposed data format, communication protocol, or device interface. This is basically related to the communication to external tools and devices
- **Stamp coupling** - When multiple modules share common data structure and work on different part of it, it is called stamp coupling.
- **Data coupling** - Data coupling is when two modules interact with each other by means of passing data (as parameter).



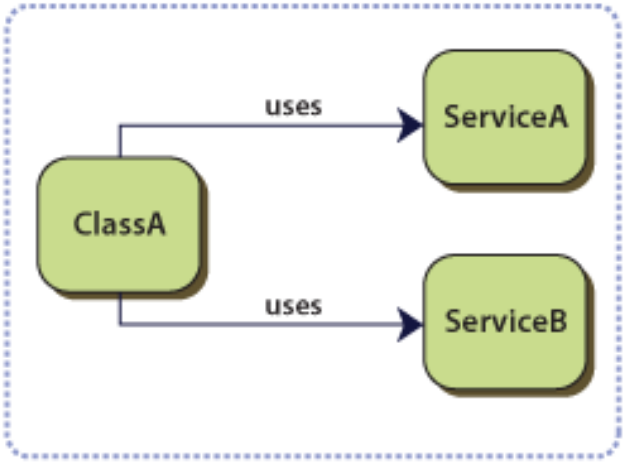
https://www.javatpoint.com/software-engineering-coupling-and-cohesion

Dependency Injection (DI)



- You have classes that have dependencies on services or components whose concrete type is specified at design time. In this example, ClassA has dependencies on ServiceA and ServiceB

- This situation has the following problems:
  - To replace or update the dependencies, you need to change your classes' source code.
  - The concrete implementations of the dependencies have to be available at compile time.
  - Your classes are difficult to test in isolation because they have direct references to dependencies. This means that these dependencies cannot be replaced with stubs or mocks.
  - Your classes contain repetitive code for creating, locating, and managing their dependencies.



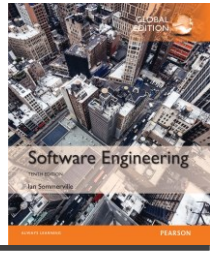
Solution: Delegate the function of selecting a concrete implementation type for the classes' dependencies to an external component or source.

http://jst4fun.com/dev/inversion-of-control-ioc

Dependency Injection (DI)

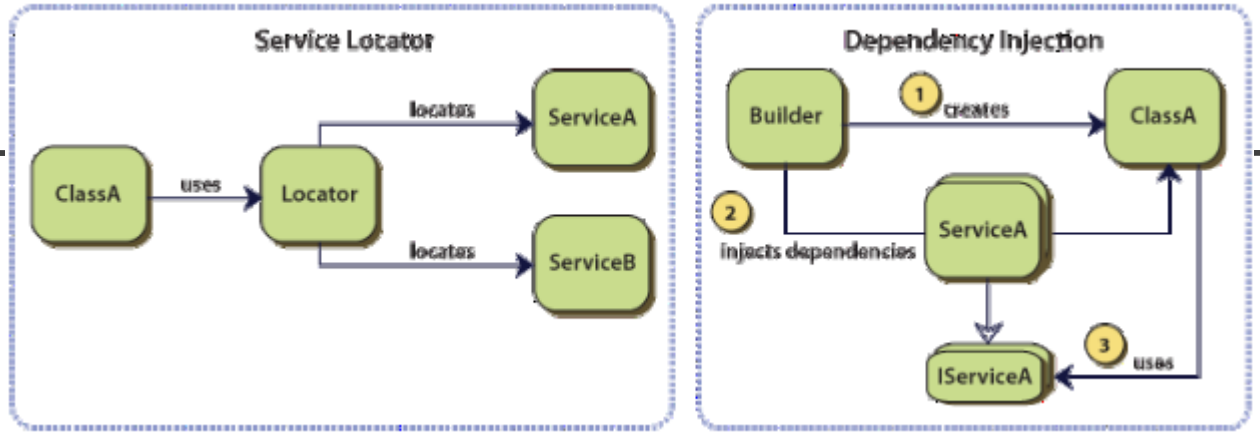
- **Dependency Injection (DI)** is a technique in which an object receives other objects that it depends on. These other objects are called dependencies.
  - In the typical "using" relationship the receiving object is called a client and the passed (that is, "injected") object is called a service
  - The code that passes the service to the client can be many kinds of things and is called the injector
  - Passing the service to the client, rather than allowing a client to build or find the service
  - The intent behind dependency injection is to achieve **separation of concerns** of construction and use of objects. This can increase readability and code reuse.
    - This means the client does not need to know about the injector, how to construct the services, or even which services it is actually using.
    - The client only needs to know the interfaces of the services, because these define how the client may use the services. (separates 'use' from 'construction'.)

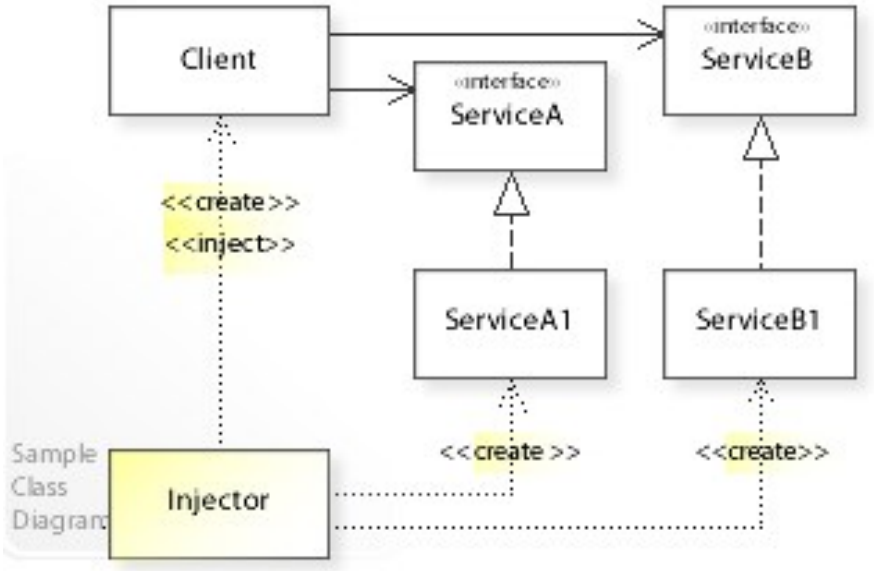
Dependency Injection (DI)

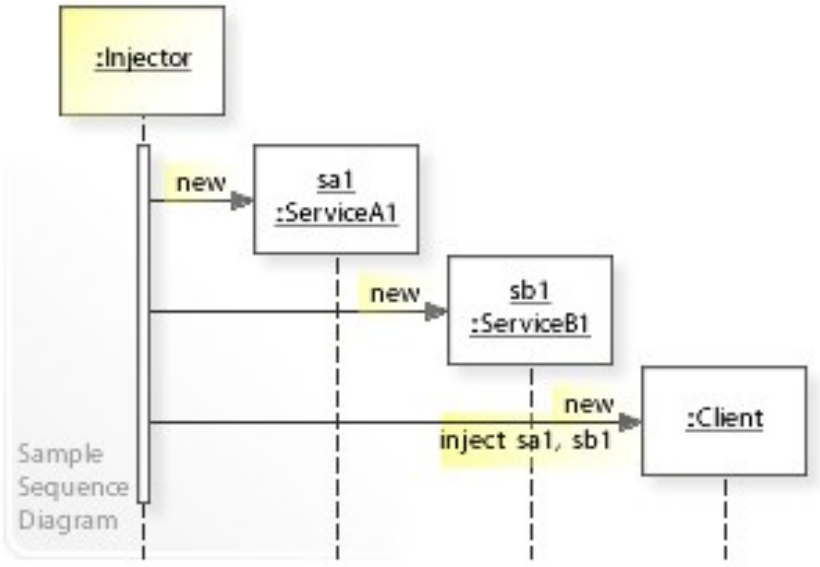


- **Dependency injection** separates the creation of a client's dependencies from the client's behavior, which allows program designs to be *loosely coupled* and to follow the dependency inversion and single responsibility principles.
  - It directly contrasts with the **service locator pattern (anti-pattern)**, which allows clients to know about the system they use to find dependencies.
- Dependency injection involves four roles:
  - the service object(s) to be used
  - the client object that is depending on the service(s) it uses
  - the interfaces that define how the client may use the services
  - the injector, which is responsible for constructing the services and injecting them into the client
    - DI (or IoC) container framework

https://en.wikipedia.org/wiki/Dependency_injection

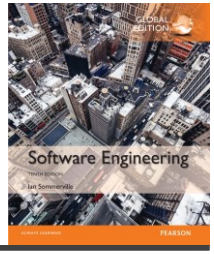






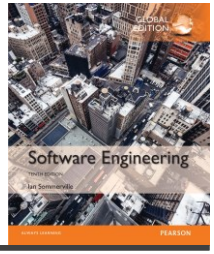
https://en.wikipedia.org/wiki/Dependency_injection

Types of Dependency Injections



- Constructor Injection: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.
- Property Injection: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.
- Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

Inversion of Control (IoC)



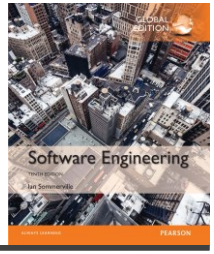
- Dependency injection is one form of the broader technique of **inversion of control (IoC)** (e.g., template method (IoC, inheritance) vs. strategy pattern (DI))
  - is a programming principle or design principle in which custom-written portions of a computer program **receive the flow of control from a generic framework**. (inverts the flow of control as compared to traditional control flow)
  - **Delegate** the function of **selecting a concrete implementation type** for the classes' dependencies to **an external component or source**.
  - Inversion of control is sometimes referred to as the "Hollywood Principle: Don't call us, we'll call you".
  - Dependency injection implements IoC through **composition**
- Implementation techniques for IOC
  - DI, factory pattern, a contextualized lookup (method injection), template method, strategy pattern

Some examples of Dependency Injection

- Constructor injection
- Setter injection (Property injection)
- Interface injection (Method injection)

```java
public class Client {
    // Internal reference to the service used by this client
    private ExampleService service;

    // Constructor
    Client() {
        // Specify a specific implementation instead of using dependency injection
        service = new ExampleService();
    }

    // Method that uses the services
    public String greet() {
        return "Hello " + service.getName();
    }
}
```

Constructor Injection




```
// Constructor
Client(Service service, Service otherService) {
    if (service == null) {
        throw new InvalidParameterException("service must not be null");
    }
    if (otherService == null) {
        throw new InvalidParameterException("otherService must not be null");
    }

    // Save the service references inside this client
    this.service = service;
    this.otherService = otherService;
}
```

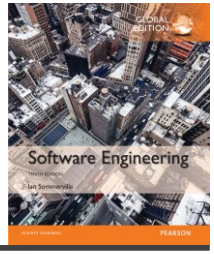
Require the client to provide a parameter in a constructor for the dependency.

May lack the flexibility to have its dependencies changed later

Setter injection

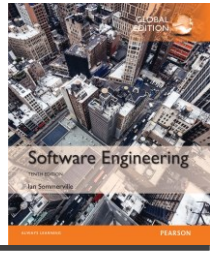
- Require the client to provide a setter method for the dependency.
- This offers flexibility, but it is difficult to ensure that all dependencies are injected before the client is used.

```
// Set the service to be used by this client
public void setService(Service service) {
    this.service = service;
}

// Set the other service to be used by this client
public void setOtherService(Service otherService) {
    this.otherService = otherService;
}

// Check the service references of this client
private void validateState() {
    if (service == null) {
        throw new IllegalStateException("service must not be null");
    }
    if (otherService == null) {
        throw new IllegalStateException("otherService must not be null");
    }
}

// Method that uses the service references
public void doSomething() {
    validateState();
    service.doYourThing();
    otherService.doYourThing();
}
```



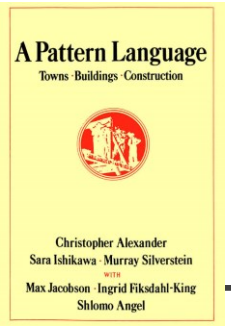
Design patterns

Design patterns



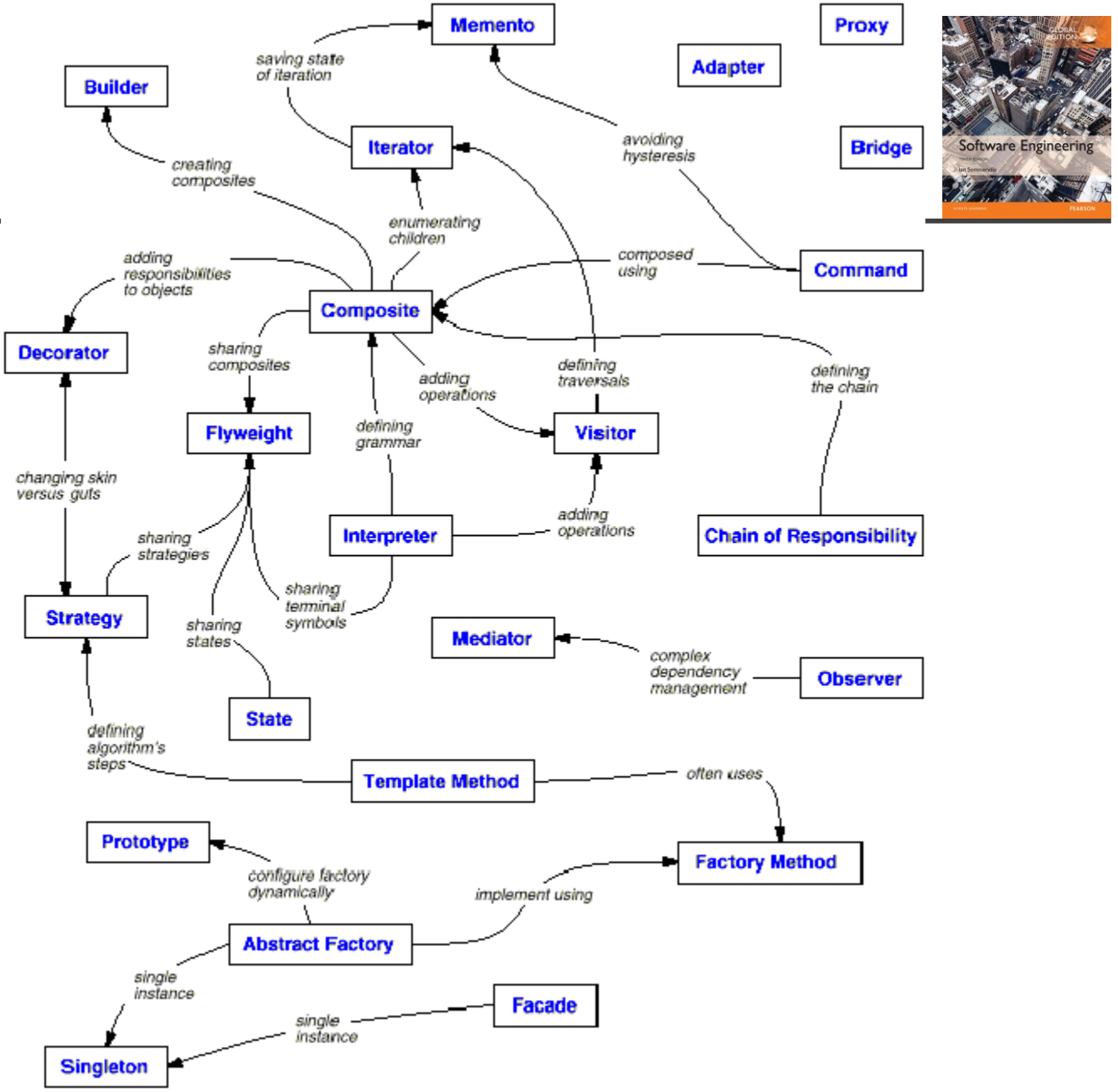
- A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
  - You can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem
- It should be sufficiently abstract to be reused in different settings.
  - Patterns support high-level, concept reuse
- Pattern descriptions usually make use of object-oriented characteristics such as **inheritance** and **polymorphism**.

Patterns

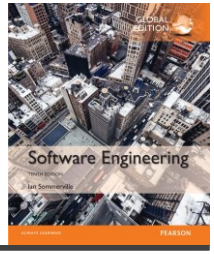


- *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to **reuse** this experience.*
- A **pattern language** is a method of describing good design practices or patterns of useful organization within a field of expertise. (*an organized and coherent set of patterns*)
  - The term was coined by architect Christopher Alexander and popularized by his 1977 book *A Pattern Language*.
  - Many patterns form a language
    - Just as words must have grammatical and semantic relationships to each other, design patterns must be related to each other in position and utility order to form a pattern language.

Pattern Map from the GoF collection

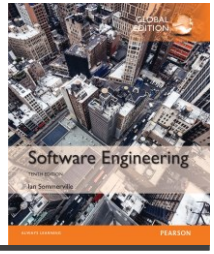


Alexandrian form (canonical form)



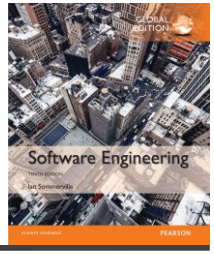
- Name
  - meaningful name
- Problem
  - the statement of the problem
- Context
  - a situation giving rise to a problem
- Forces
  - a description of relevant forces and constraints
- Solution
  - proven solution to the problem
- Examples
  - sample applications of the pattern
- Resulting context (force resolution)
  - the state of the system after pattern has been applied
- Rationale
  - explanation of steps or rules in the pattern
- Related patterns
  - static and dynamic relationship
- Known use
  - occurrence of the pattern and its application within existing system

Pattern elements



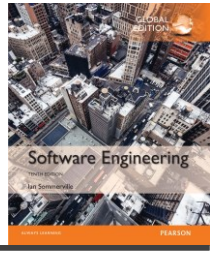
- Name
  - A meaningful pattern identifier.
- Problem description (Intent, Motivation, Applicability)
  - Explain when the pattern may be applied
- Solution description (Structure, Participants, Collaborations, Implementation)
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- Consequences
  - The results and trade-offs of applying the pattern.

The Observer pattern



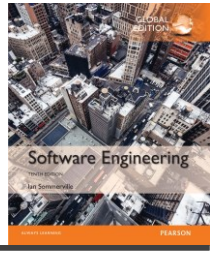
- Name
  - Observer.
- Description
  - Separates the display of object state from the object itself.
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically (Gamma et al.)
- Problem description
  - Used when 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)



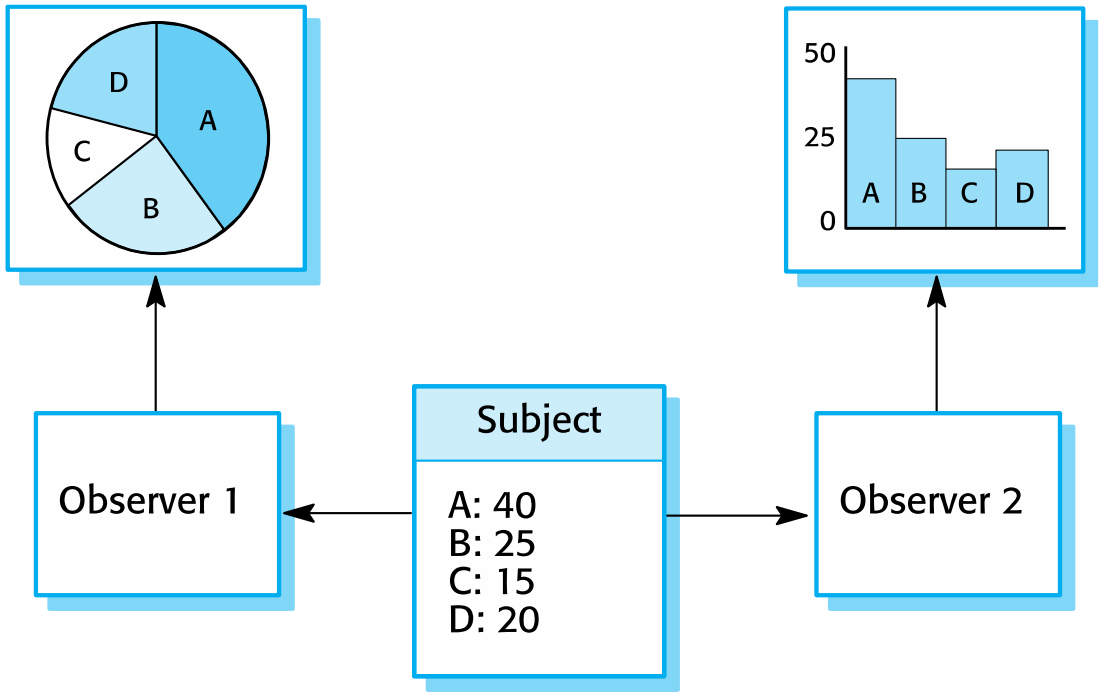
Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you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



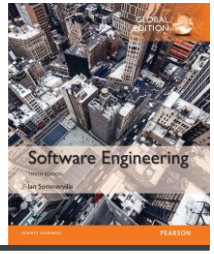
Pattern name	Observer
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObserver, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all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

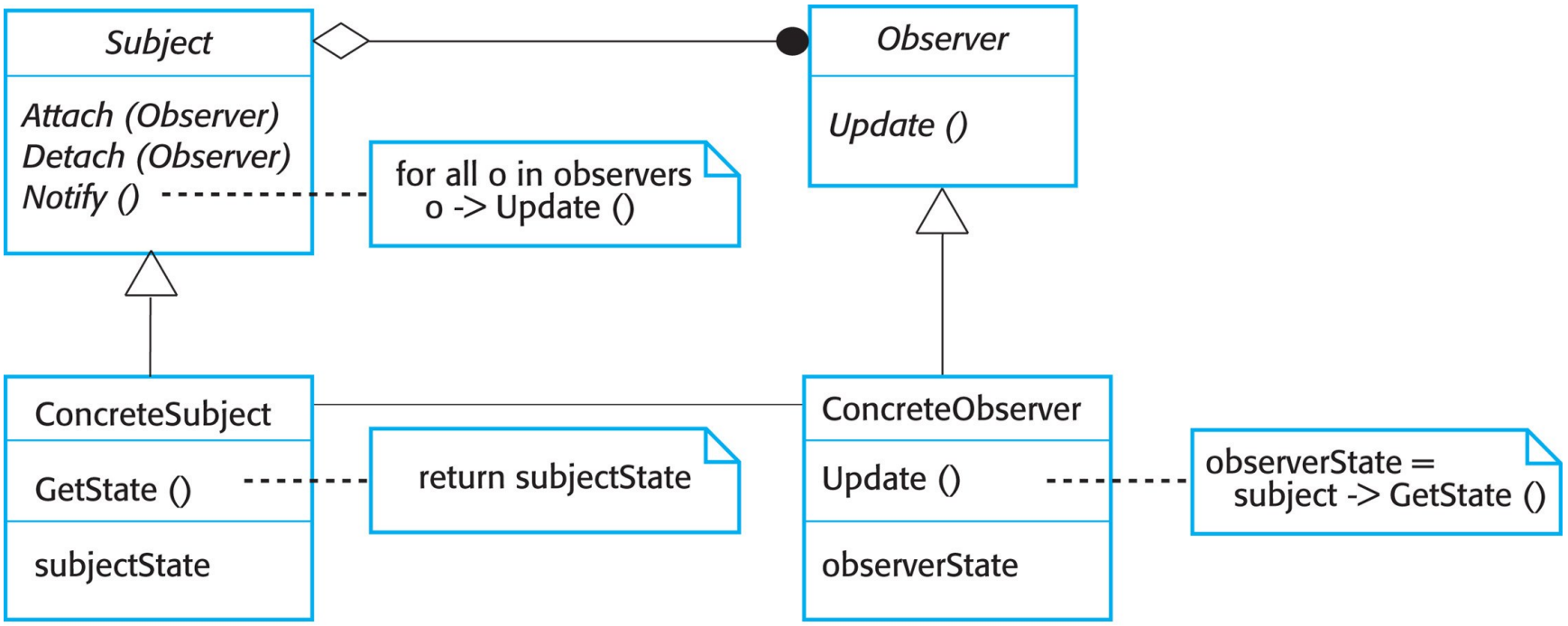
Multiple displays using the Observer pattern



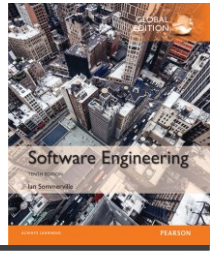
The observer pattern is also a key part in the model–view–controller (MVC) architectural pattern

A UML model of the Observer pattern



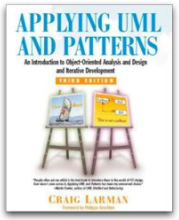


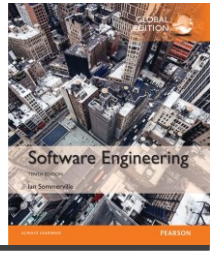
Uses of Design Patterns



- To use patterns in your design, you need to recognize that any design problem you are facing (experiencing a problem) 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).
- Design patterns provide a standard terminology (common platform for developers) and can be used as best practices for solving design problems during software development and provide a faster way to learn software design

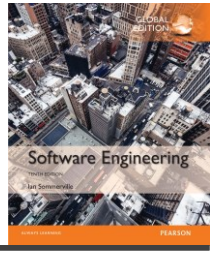
GRASP Design Patterns





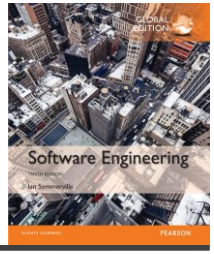
- GRASP (General Responsibility Assignment Software Patterns)
  - Best practices for **object responsibility assignment**
- There are 9 GRASP patterns
  - **Controller:** who should be the first to receive a message in the domain layer?
  - **Creator:** who should be responsible for creating a specific object?
  - **Information expert:** who should be responsible for knowing about a particular object?
  - **Low coupling:** how strongly should elements be tied to others?
  - **High cohesion:** how should responsibilities be grouped within classes?
  - **Indirection:** where to assign a responsibility to avoid direct coupling between two or more things?
  - **Polymorphism:** how to handle alternatives (the variation of behaviors) based on type?
  - **Protected variations:** how to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
  - **Pure fabrication:** is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived This kind of class is called a "service" in domain-driven design.

https://myweb.ntut.edu.tw/~wkchen/courses/ooad/files/chapter17-21.pdf



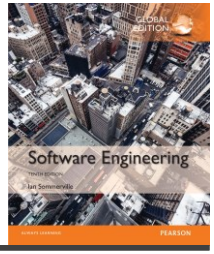
Implementation issues

Implementation issues



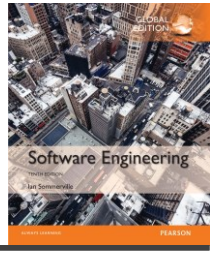
- Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:
  - Reuse Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much 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



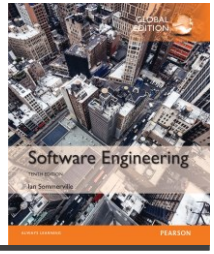
- From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- Costs and schedule pressure mean that this approach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

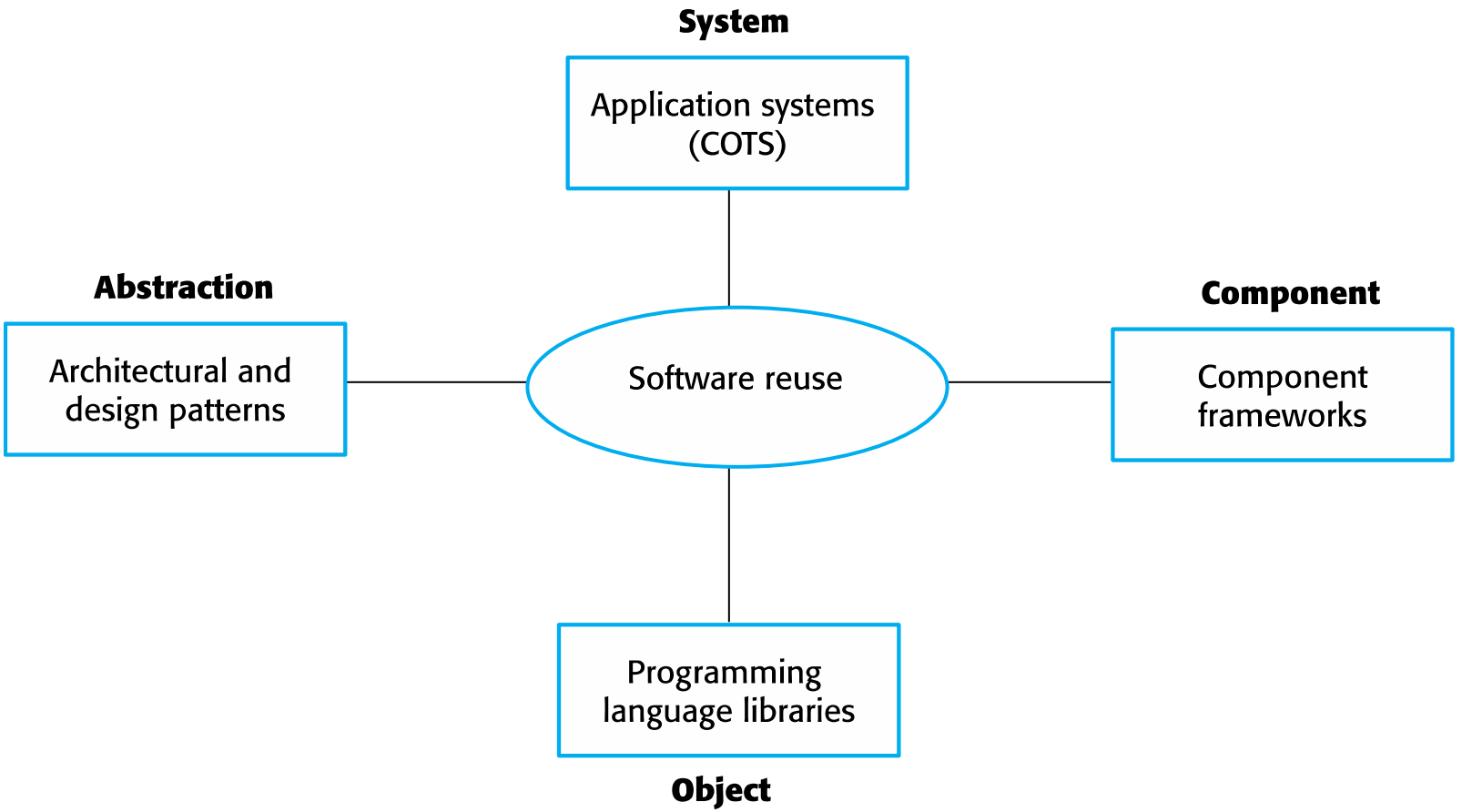
Reuse levels



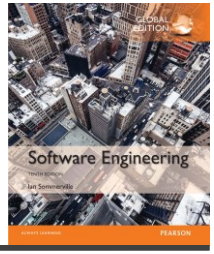
- The abstraction level
  - At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software. (e.g., design pattern, architecture pattern)
- The object level
  - At this level, you directly reuse objects from a library rather than writing the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

Software reuse



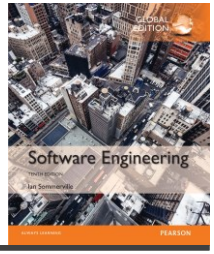


Reuse costs



- 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



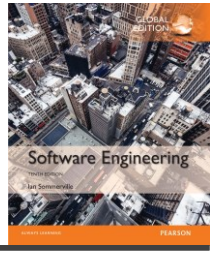
- Configuration management is the name given to the general process of managing a changing software system.
- The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.
- See also Chapter 25.

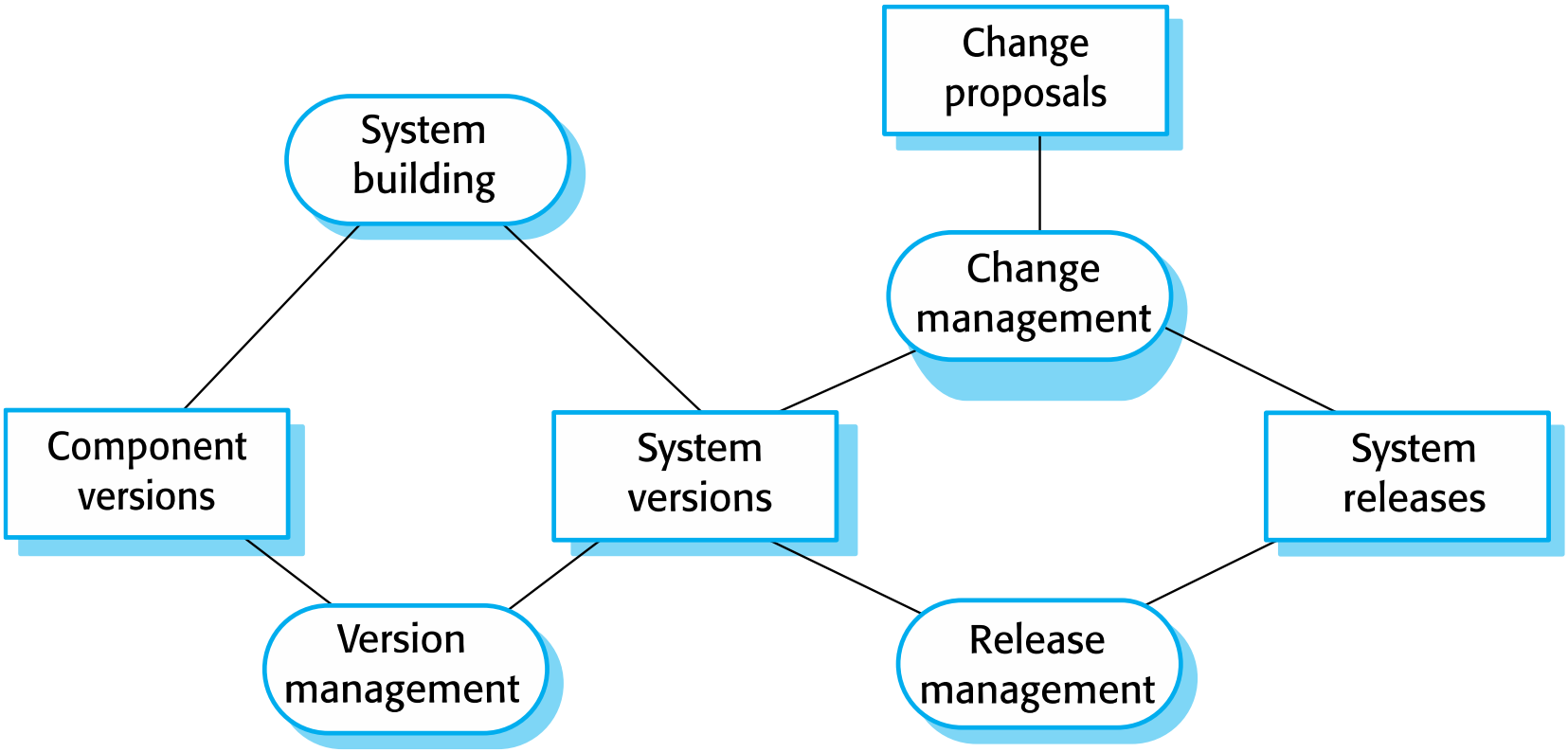
Configuration management activities

- The SCM process defines a series of tasks:
  - Identification of objects in the software configuration
  - Version Control
  - Change Control
  - Configuration Audit, and
  - Reporting

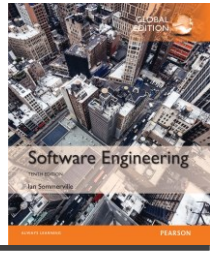
- Version management, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- System integration, where support is provided to help developers define what versions of components 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.
- Release management, where new versions of software system are released to customers. Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

Configuration management tool interaction



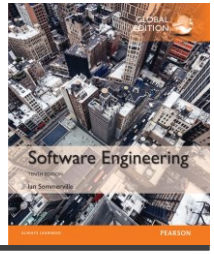


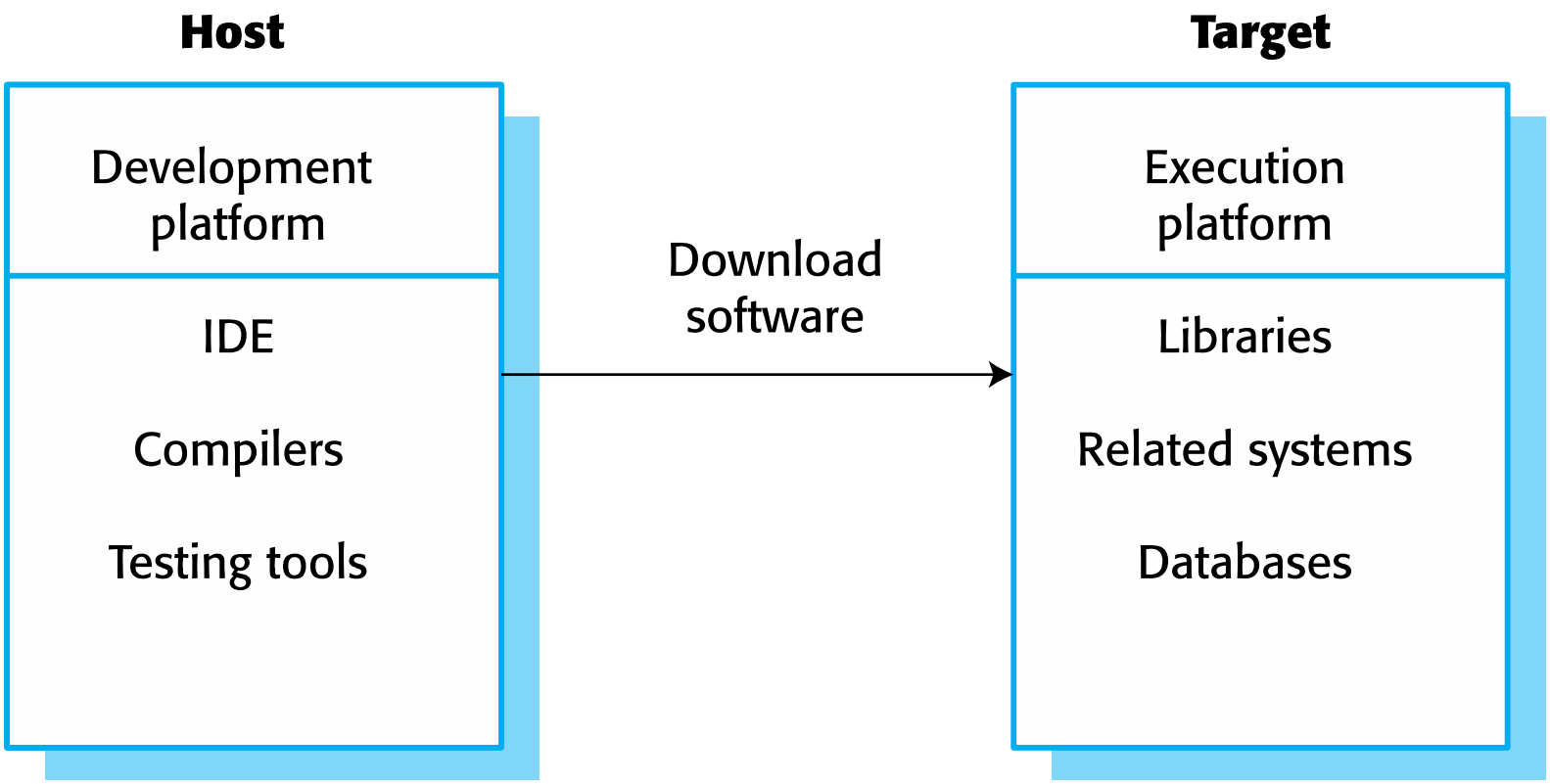
Host-target development



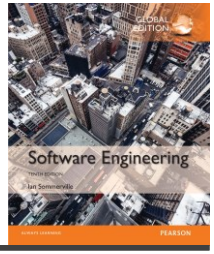
- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software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

Host-target development



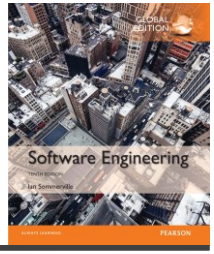


Development platform tools



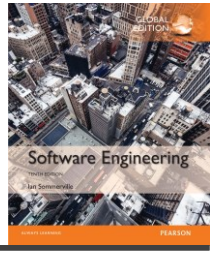
- An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- A language debugging system.
- Graphical editing tools, such as tools to edit UML models.
- Testing tools, such as JUnit that can automatically run a set of tests on a new version of a program.
- Tools to support refactoring and program visualization.
- Configuration management tools to manage source code versions and to integrate and build systems.

Integrated development environments (IDEs)

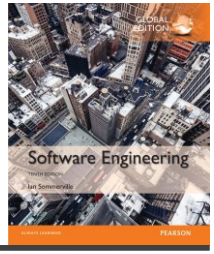


- Software development tools are often grouped to create an integrated development environment (IDE).
- An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- IDEs are created to support development in a specific programming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

Component/system deployment factors

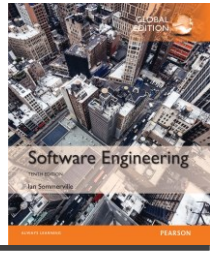


- As part of the development process, you need to make decisions about how the developed software will be deployed on the target platform (especially for distributed systems). You may consider the following issues:
  - **The hardware and software requirements of a component**.
    - If a component is 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.
  - **The availability requirements of the system**
    - 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.
  - **Component communications**
    - 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.



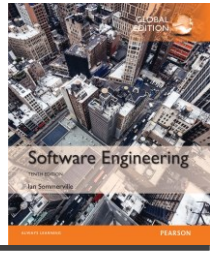
Open source development

Open source development



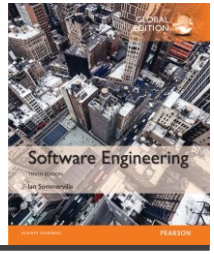
- Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- Its roots are in the **Free Software Foundation** (www.fsf.org), which advocates that source code should not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



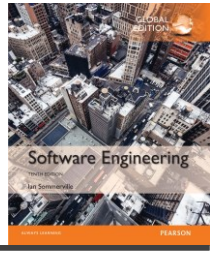
- The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- Other important open source products are Java, the Apache web server and the mySQL database management system.
- It is usually cheap or free to acquire open source software
- The other key benefit of using open source products is that mature open source systems are usually very reliable
  - Bugs are discovered and repaired more quickly that is usually possible with proprietary software

Open source issues



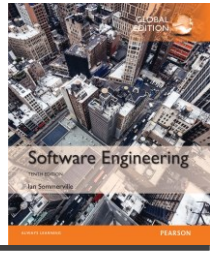
- For a company involved in software development, there are two open source issues that have to be considered:
  - Should the product that is being developed make use of open source components?
  - Should an open source approach be used for the software's development?
- The answers to these questions depend on
  - The type of software that is being developed
    - Software product for sale (time to market and reduced cost are critical) (may consider open source) or software to a specific set of organizational requirements (open source may not be an option due to compatability)
  - The background and experience of the development team
    - Involving the open source

Open source business



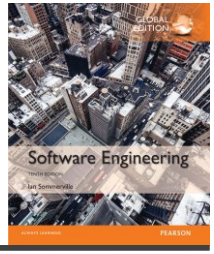
- More and more product companies are using an open source approach to development.
  - Their business model is not reliant on selling a software product but on **selling support** for that product.
  - They believe that involving the **open source community** will allow software to be developed more cheaply, more quickly and will create a community of users for the software. (may apply to general software products only)
  - Many companies believe that adopting an open source approach will reveal confidential business knowledge to their competitors. However, if you are working in a small company and you open source your software, this may reassure customers that they will be able to support the software if your company goes out of business

Open source licensing



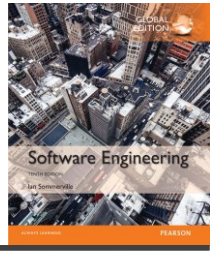
- A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
  - **Legally**, the developer of the code (either a company or an individual) **still owns the code**. They can place **restrictions** 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**. (reciprocal)
  - 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



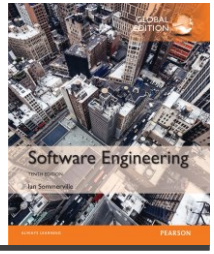
- The GNU General Public License (GPL). This is a so-called **'reciprocal' license** (互惠的) that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
- The GNU Affero Public license (AGPL) extends the GPLv3 license by giving end-users access to the source code for software accessed over a network
- 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.
  - Other similar licenses: Apache Licence, MIT

License management



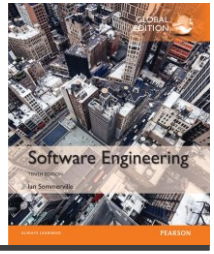
- Companies managing projects that use open source should
  - Establish a system for maintaining information about open-source components that are downloaded and used.
  - Be aware of the different types of licenses and understand how a component is licensed before it is used.
  - Be aware of evolution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

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.

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.