UML Diagrams

What is UML?

- At the end of the 1980s and the beginning of 1990s, the first object-oriented development processes appeared
- The proliferation of methods and notations tended to cause considerable confusion
- The **Unified Modelling Language is a standard graphical language** for modelling object oriented software
- UML was developed as a collaborative effort by James Rumbaugh, Grady Booch and Ivar Jacobson, each of whom had developed their own notation
- In 1997 the Object Management Group (OMG) started the process of UML standardization

UML diagrams

- Class diagrams
  - describe classes and their relationships
- Interaction diagrams
  - show the behaviour of systems in terms of how objects interact with each other
- State diagrams and activity diagrams
  - show how systems behave internally
- Component and deployment diagrams
  - show how the various components of systems are arranged logically and physically

UML features

- It has detailed *semantics*
- It has *extension* mechanisms
- It has an associated textual language
  - *Object Constraint Language* (OCL)

- The objective of UML is to assist in software development
  - It is not a *methodology*

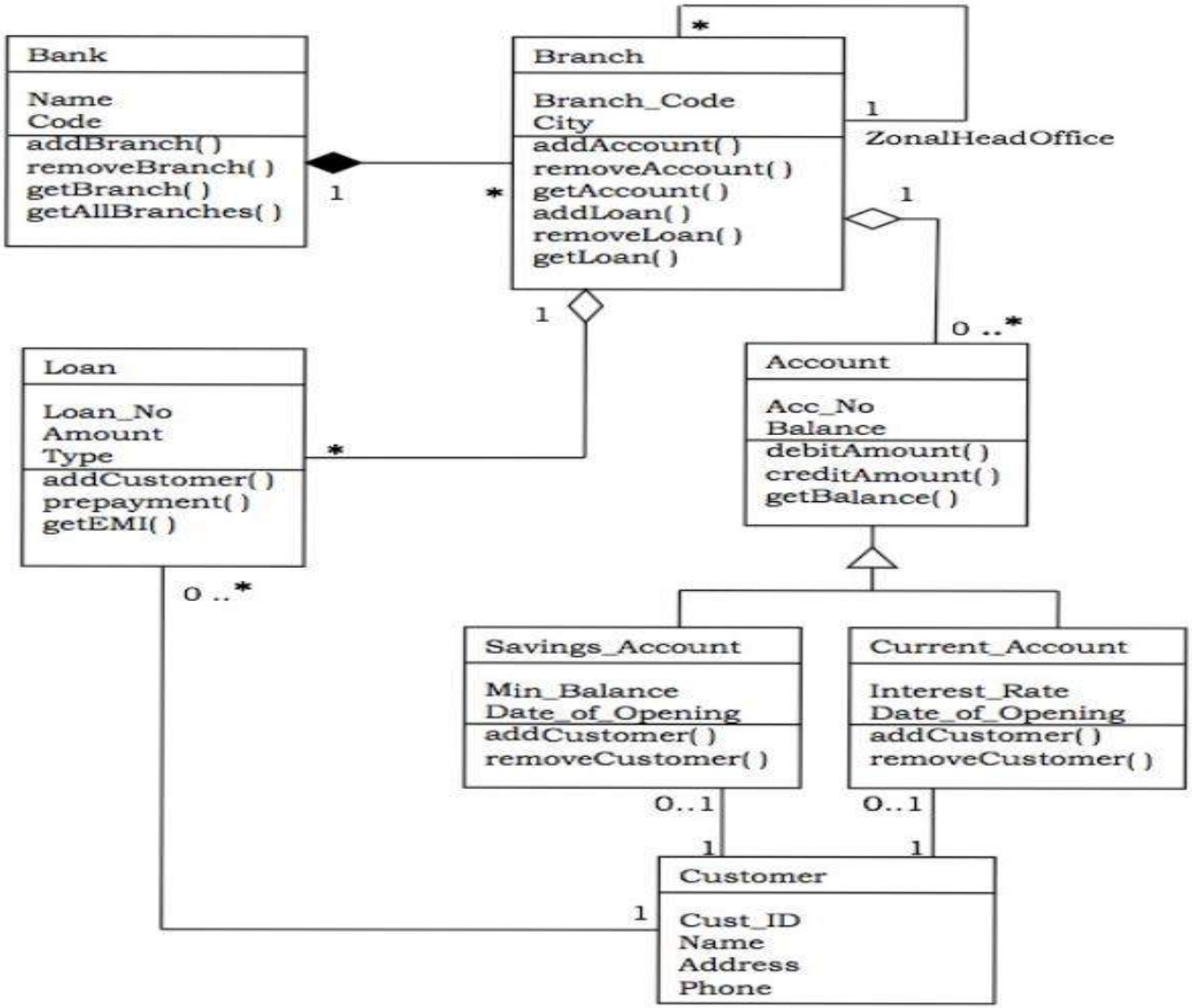
UML Class Diagram

- A type **of static structure diagram**
- In the design of a system**, a number of classes are identified and grouped together that helps to determine the static relations** between them.

Essentials of UML Class Diagrams

- *The main symbols shown on class diagrams are:*
  - *Classes*
    - represent the types of data themselves
  - *Associations*
    - represent linkages between instances of classes
  - *Attributes*
    - are simple data found in classes and their instances
  - *Operations*
    - represent the functions performed by the classes and their instances
  - *Generalizations*
    - group classes into inheritance hierarchies

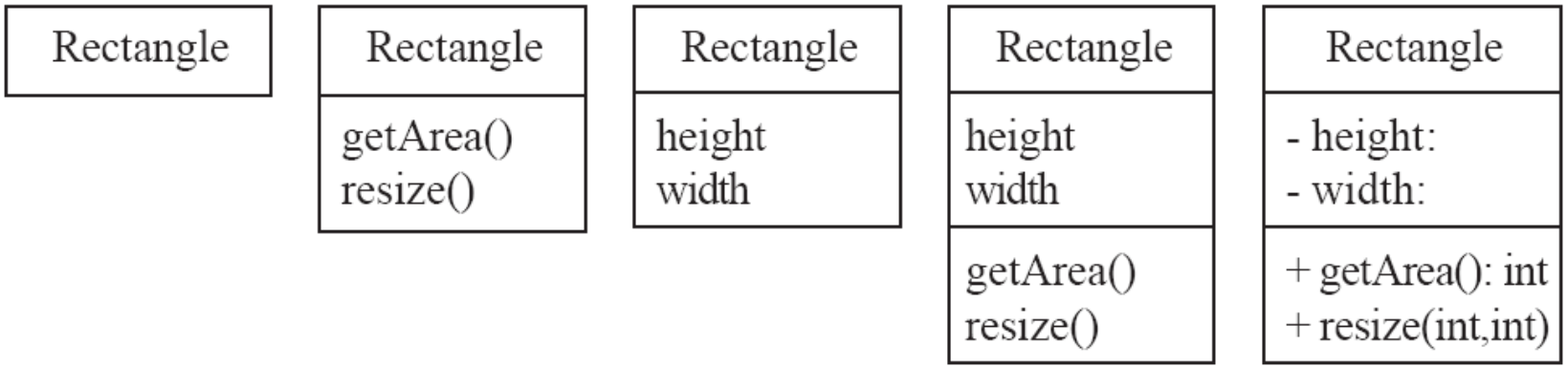
An Example


Bank
Name
Code
addBranch( )
removeBranch( )
getBranch( )
getAllBranches( )
Branch
Branch_Code
City
addAccount( )
removeAccount( )
getAccount( )
addLoan( )
removeLoan( )
getLoan( )
ZonalHeadOffice
Loan
Loan_No
Amount
Type
addCustomer( )
prepayment( )
getEMI( )
Account
Acc_No
Balance
debitAmount( )
creditAmount( )
getBalance( )
Savings_Account
Min_Balance
Date_of_Opening
addCustomer( )
removeCustomer( )
Current_Account
Interest_Rate
Date_of_Opening
addCustomer( )
removeCustomer( )
Customer
Cust_ID
Name
Address
Phone
1
*
0 ..*
0..1

Classes

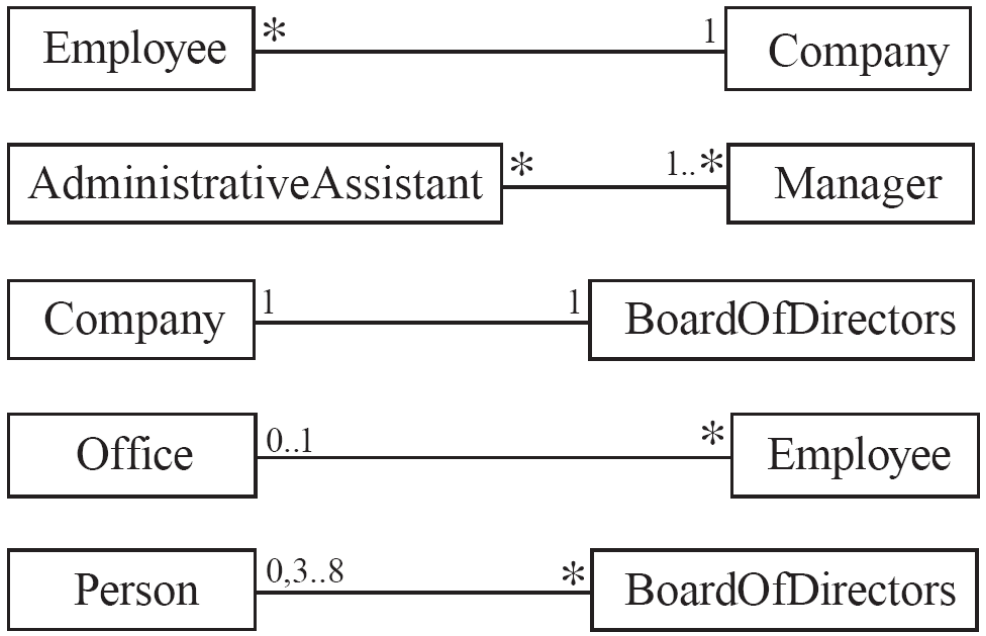
- A class is simply represented as a box with the name of the class inside
  - The diagram may also show the attributes and operations
  - The complete signature of an operation is:

    operationName(parameterName: parameterType ...): returnType



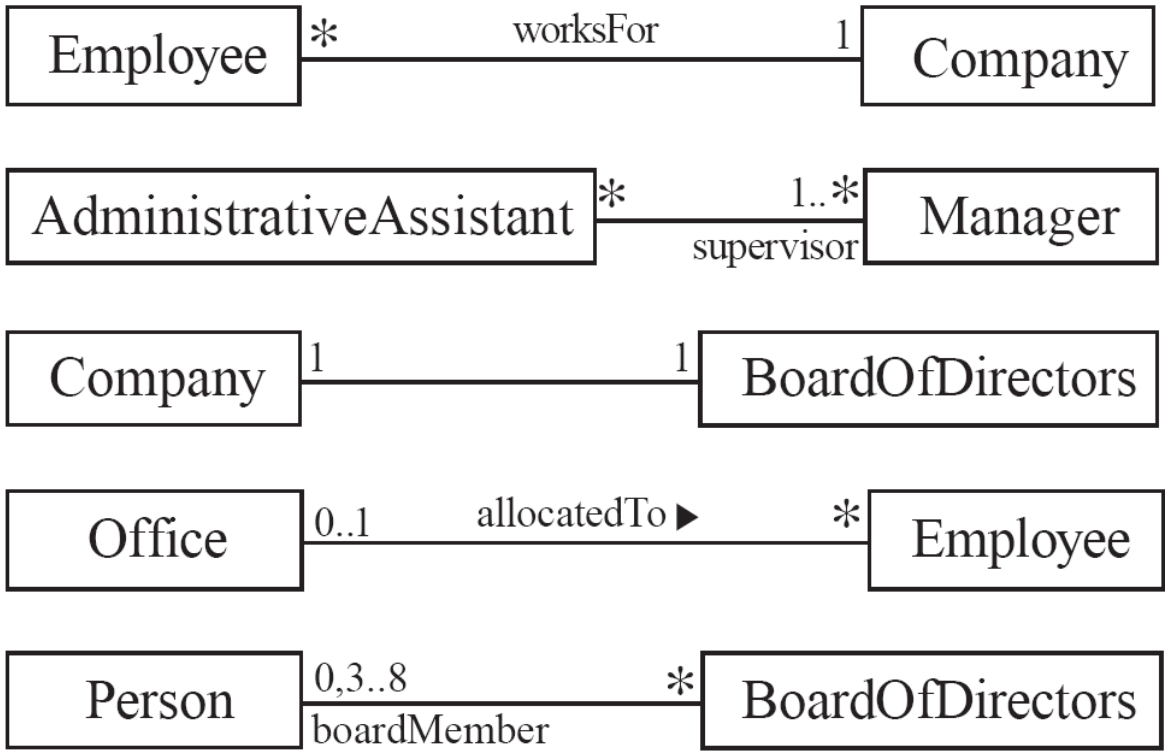
Associations and Multiplicity

- An *association* is used to show how two classes are related to each other
  - Symbols indicating *multiplicity* are shown at each end of the association



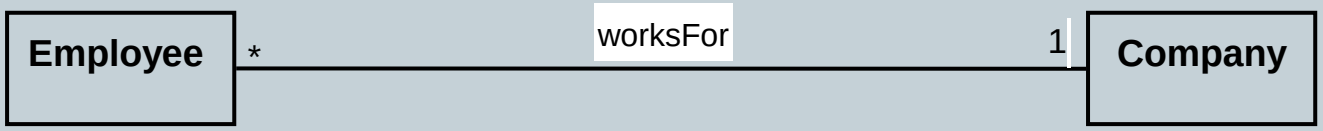
Labelling associations

- Each association can be labelled, to make explicit the nature of the association



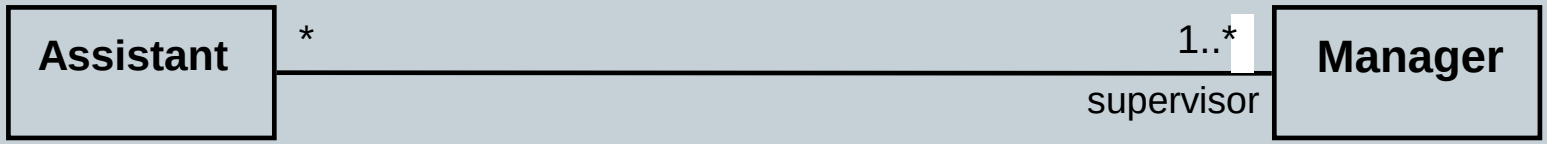
Analyzing and validating associations

- **Many-to-one**
  - A company has many employees,
  - An employee can only work for one company.
  - A company can have zero employees
    - E.g. a ‘shell’ company
  - It is not possible to be an employee unless you work for a company



Analyzing and validating associations

- **Many-to-many**
  - An assistant can work for many managers
  - A manager can have many assistants
  - Assistants can work in pools
  - Managers can have a group of assistants
  - Some managers might have zero assistants.
  - Is it possible for an assistant to have, perhaps temporarily, zero managers?



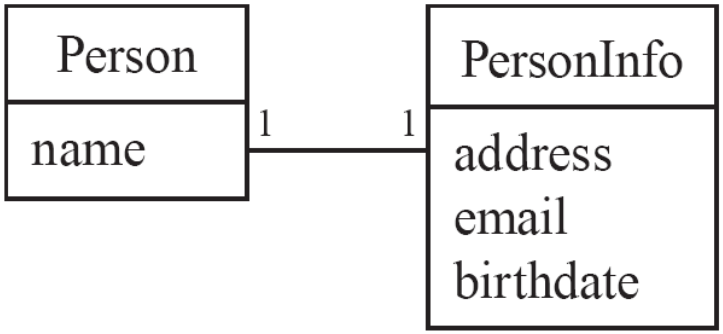
Analyzing and validating associations

- **One-to-one**
  - For each company, there is exactly one board of directors
  - A board is the board of only one company
  - A company must always have a board
  - A board must always be of some company

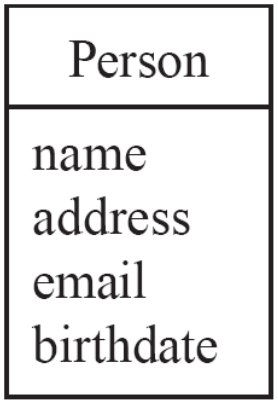
Analyzing and validating associations

- Avoid unnecessary one-to-one associations

Avoid this

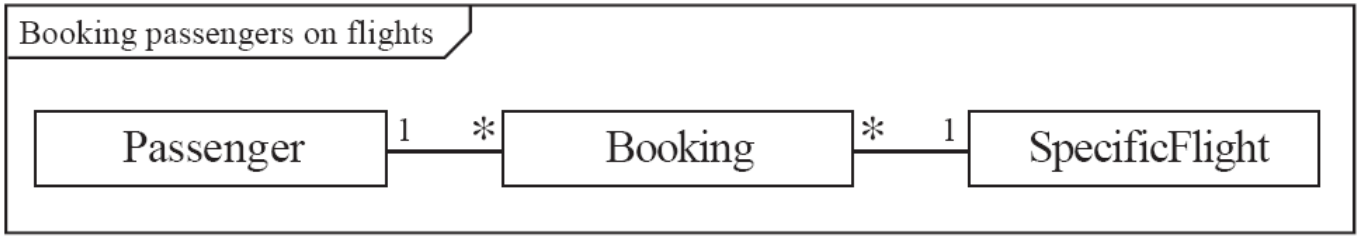


do this



A more complex example

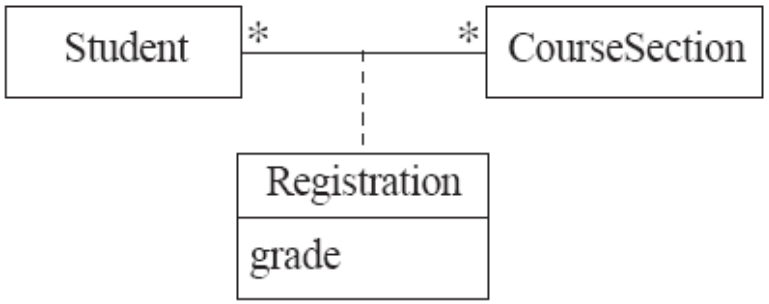
- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could *never* involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking



- The *frame* around this diagram is an optional feature that any UML 2.0 may possess.

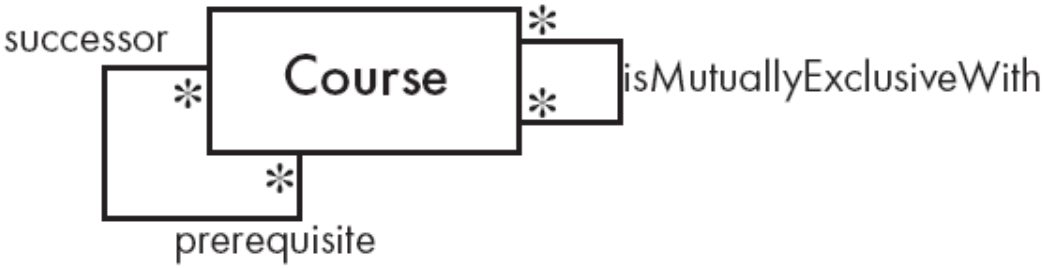
Association classes

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent



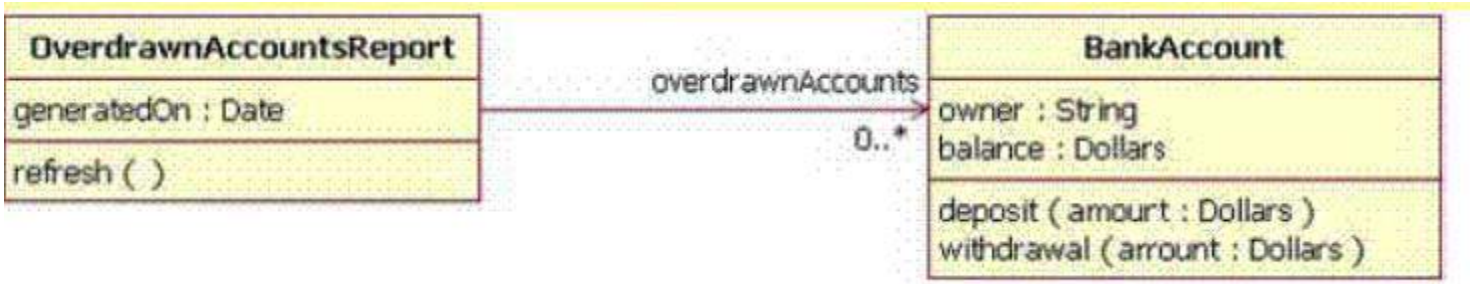
Reflexive associations

- It is possible for an association to connect a class to itself



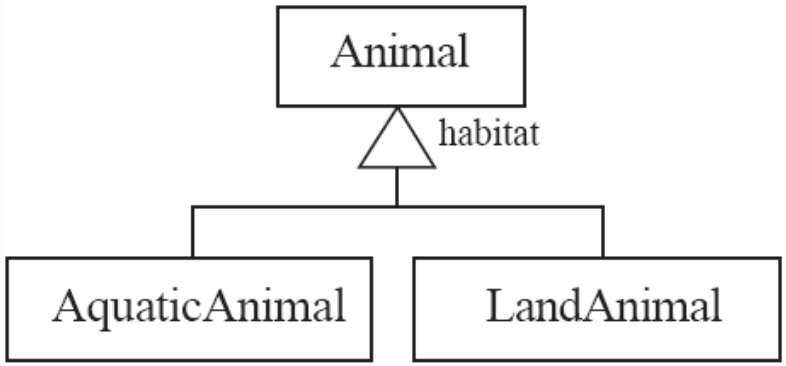
Directionality in associations

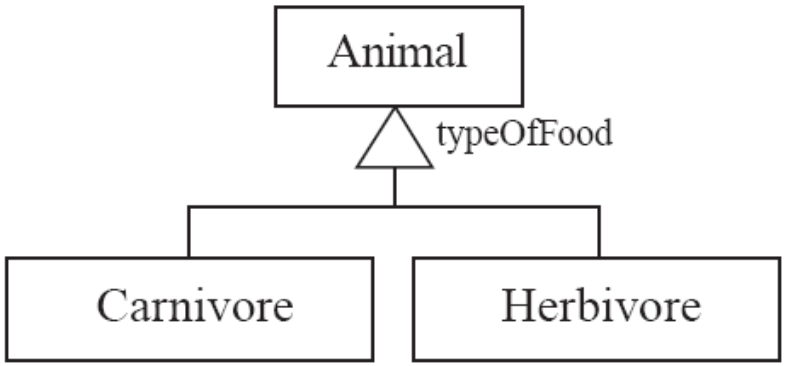
- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end



Generalization

- Specializing a superclass into two or more subclasses
  - A *generalization set* is a labeled group of generalizations with a common superclass
  - The label (sometimes called the *discriminator*) describes the criteria used in the specialization

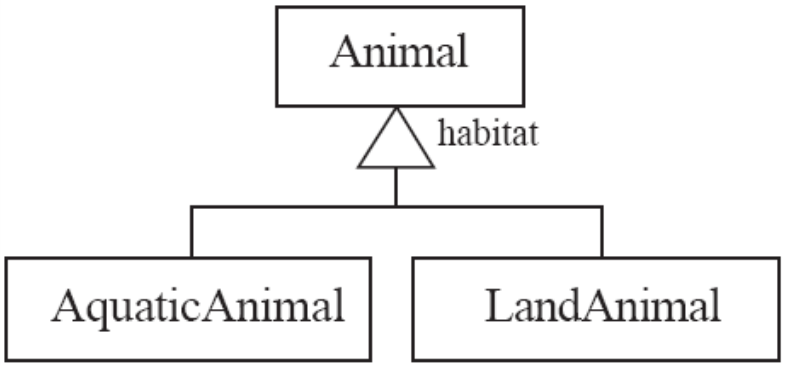


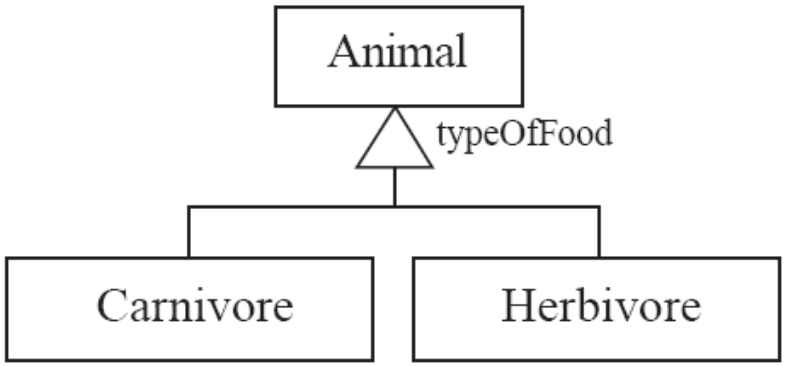


UML – Class Diagram

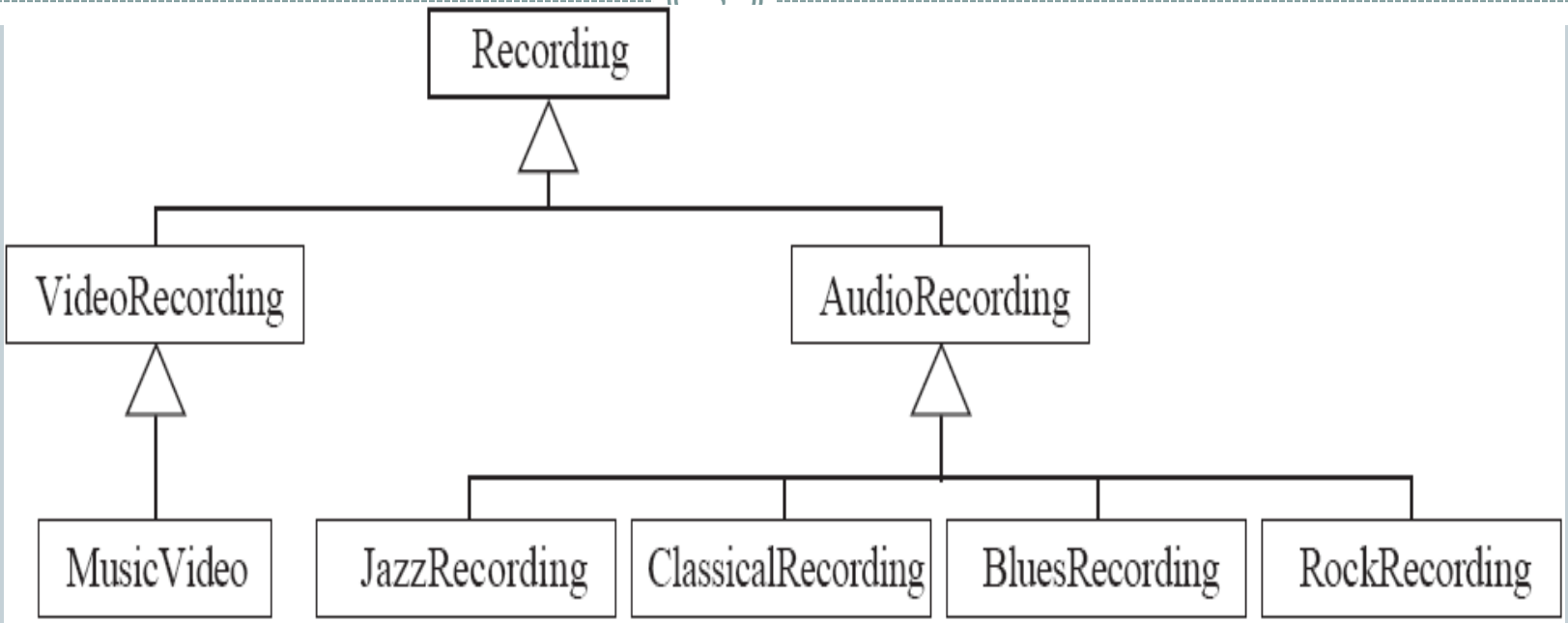
Generalization

- Specializing a superclass into two or more subclasses
  - A *generalization set* is a labeled group of generalizations with a common superclass
  - The label (sometimes called the *discriminator*) describes the criteria used in the specialization



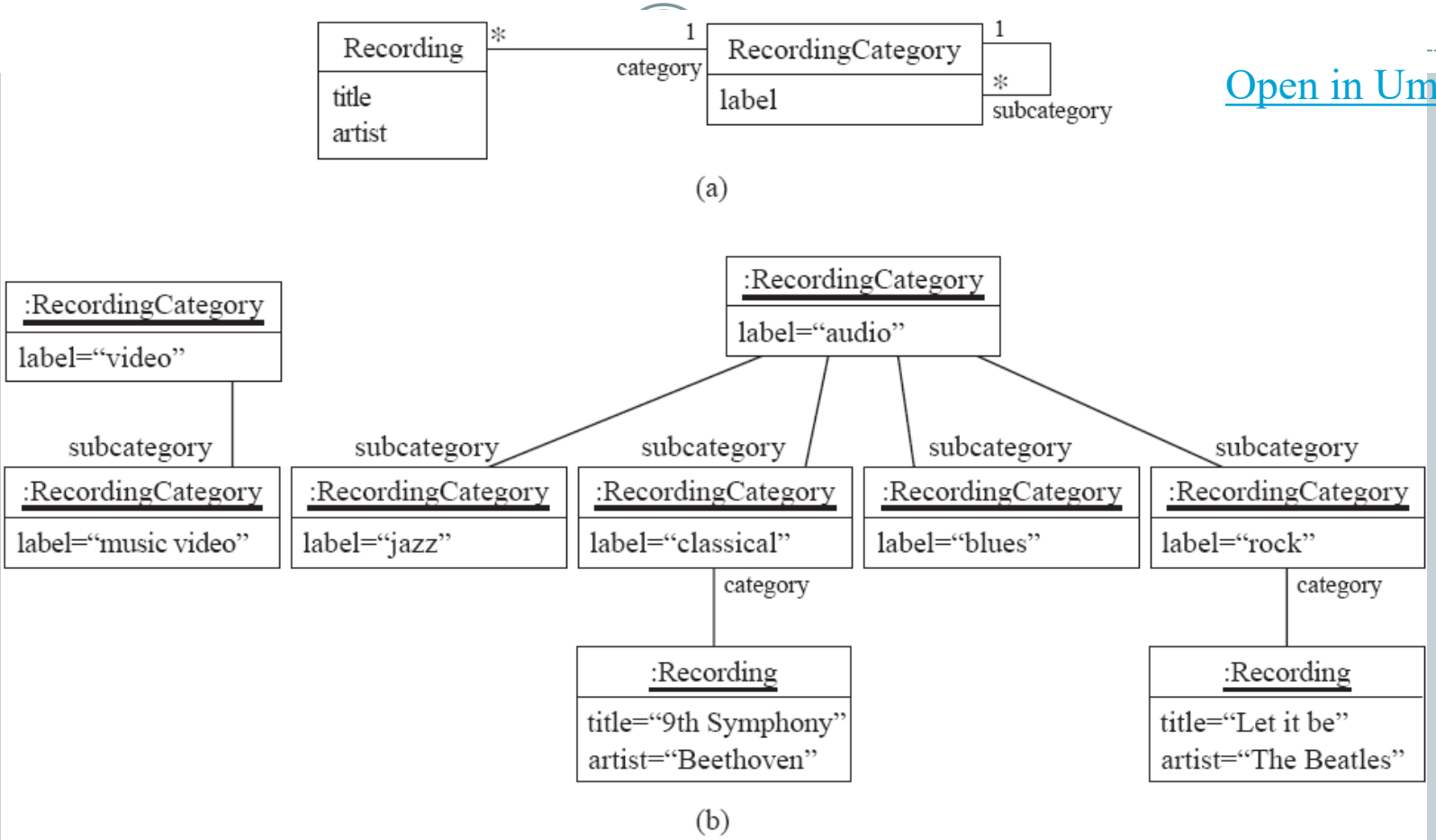


Avoiding unnecessary generalizations



Inappropriate hierarchy of classes, which should be instances

Avoiding unnecessary generalizations (cont)



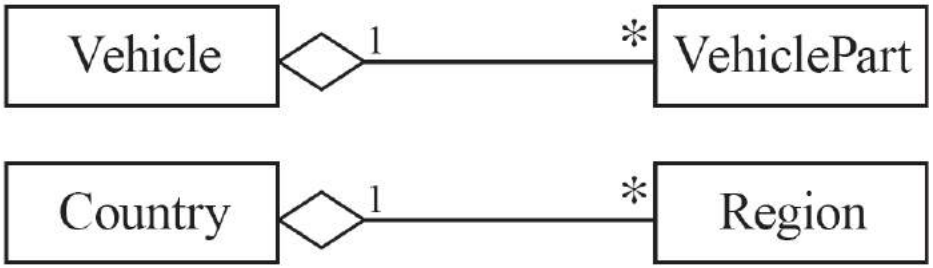
Open in Umple

Improved class diagram, with its corresponding instance diagram

Relationships

More Advanced Features: Aggregation

- Aggregations are special associations that represent 'part-whole' relationships.
  - The 'whole' side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named `isPartOf`



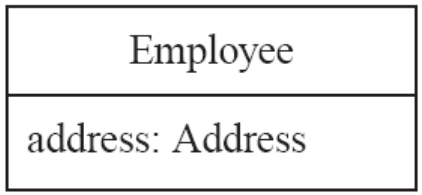
When to use an aggregation

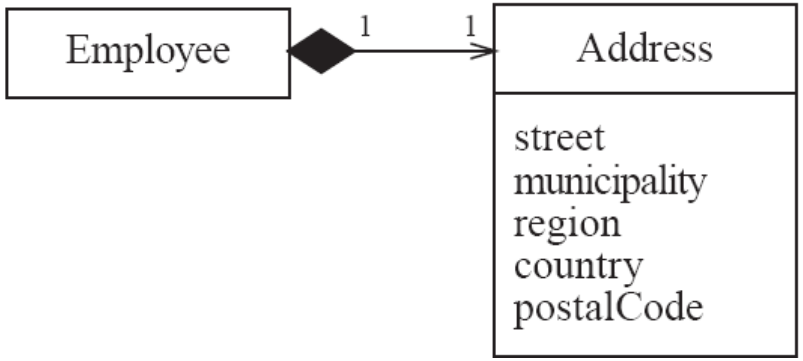
- As a general rule, you can mark an association as an aggregation if the following are true:
  - You can state that
    - the parts ‘are part of’ the aggregate
    - or the aggregate ‘is composed of’ the parts
  - When something owns or controls the aggregate, then they also own or control the parts

Composition

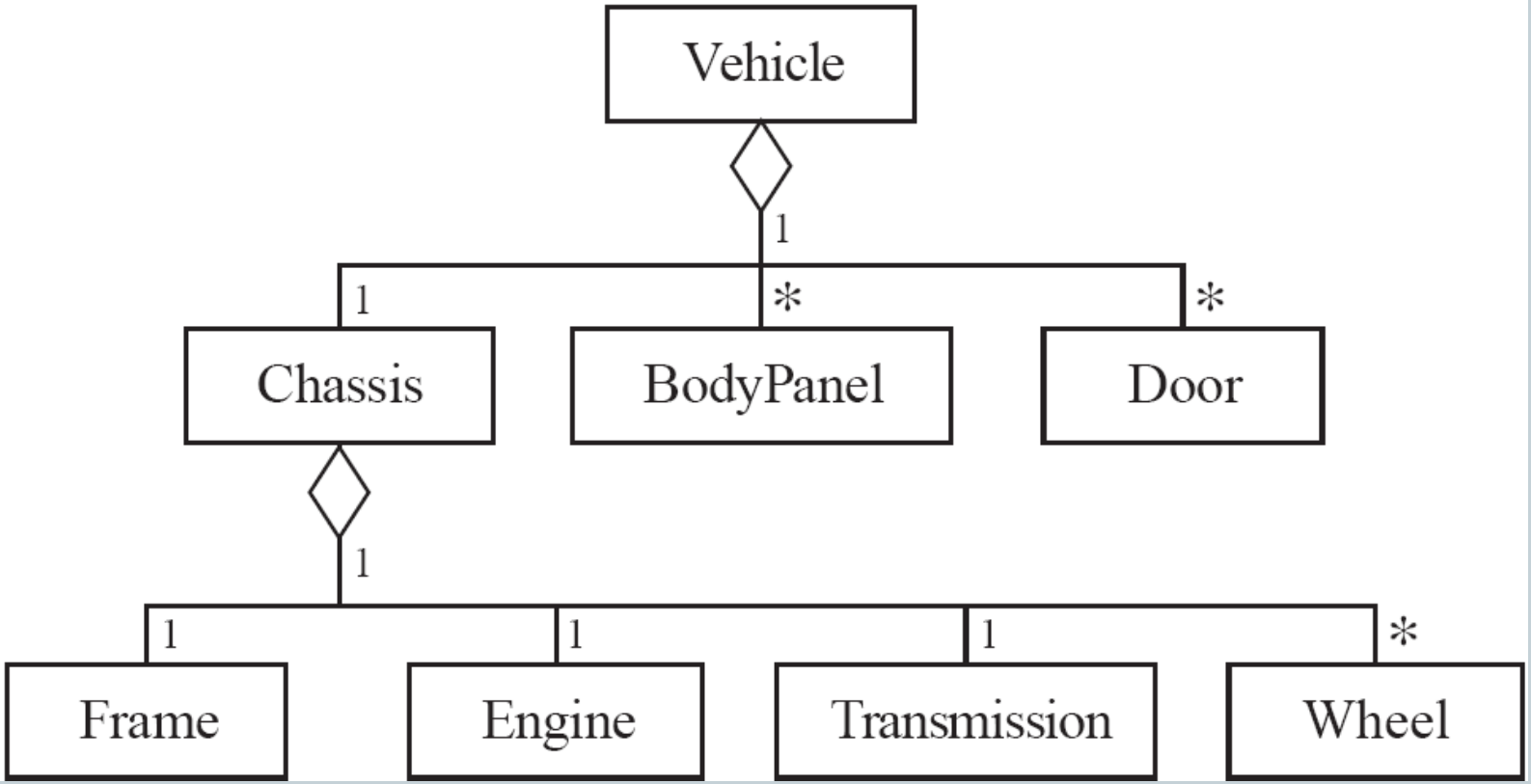
- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well

- Two alternatives for addresses





Aggregation hierarchy



Propagation

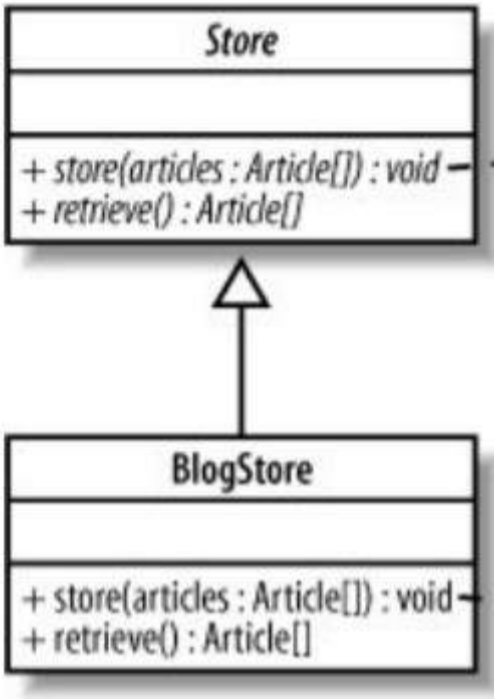
- A mechanism where an operation in an aggregate is implemented by having the aggregate perform that operation on its parts
- At the same time, properties of the parts are often propagated back to the aggregate
- Propagation is to aggregation as inheritance is to generalization.
  - The major difference is:
    - inheritance is an implicit mechanism

Abstract class

- When the concrete implementation of methods are left for the subclasses.
- Can contain both abstract and non-abstract methods

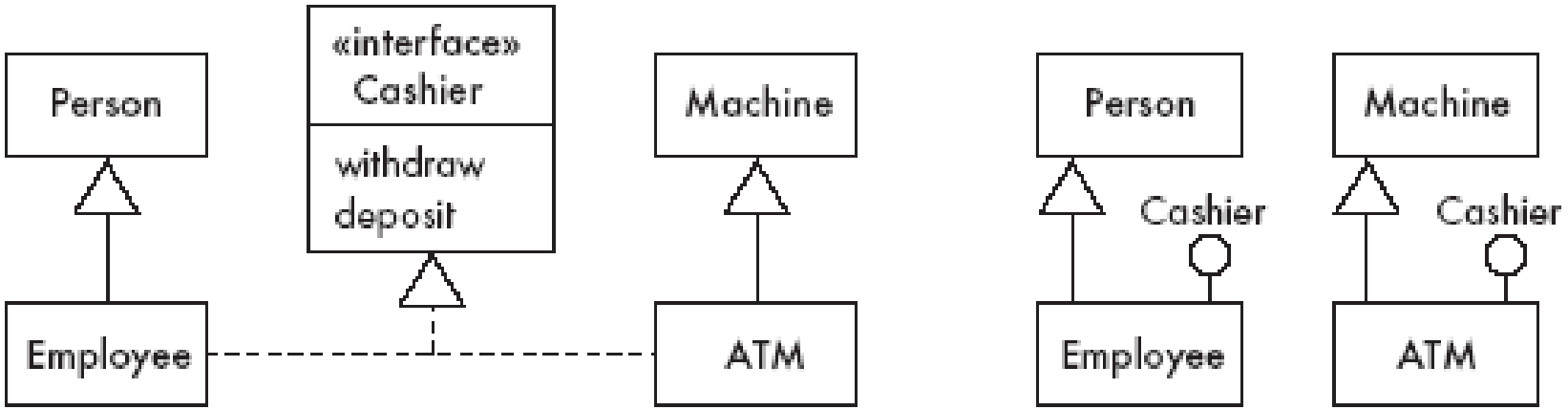

```
public abstract class Store {
    public abstract void store(Article[] articles);
    public abstract Article[] retrieve(  );
}
```

Abstract class



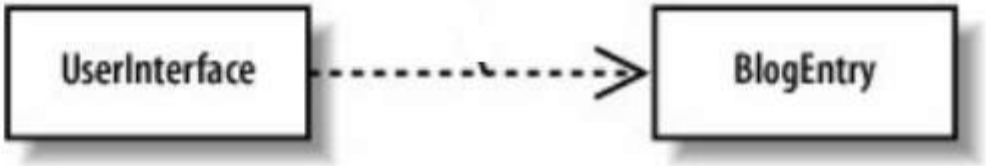
Interfaces

- An *interface* is similar to a class, except it lacks instance variables and implemented methods
- An interface describes a *portion of the visible behaviour* of a set of objects.



Dependency

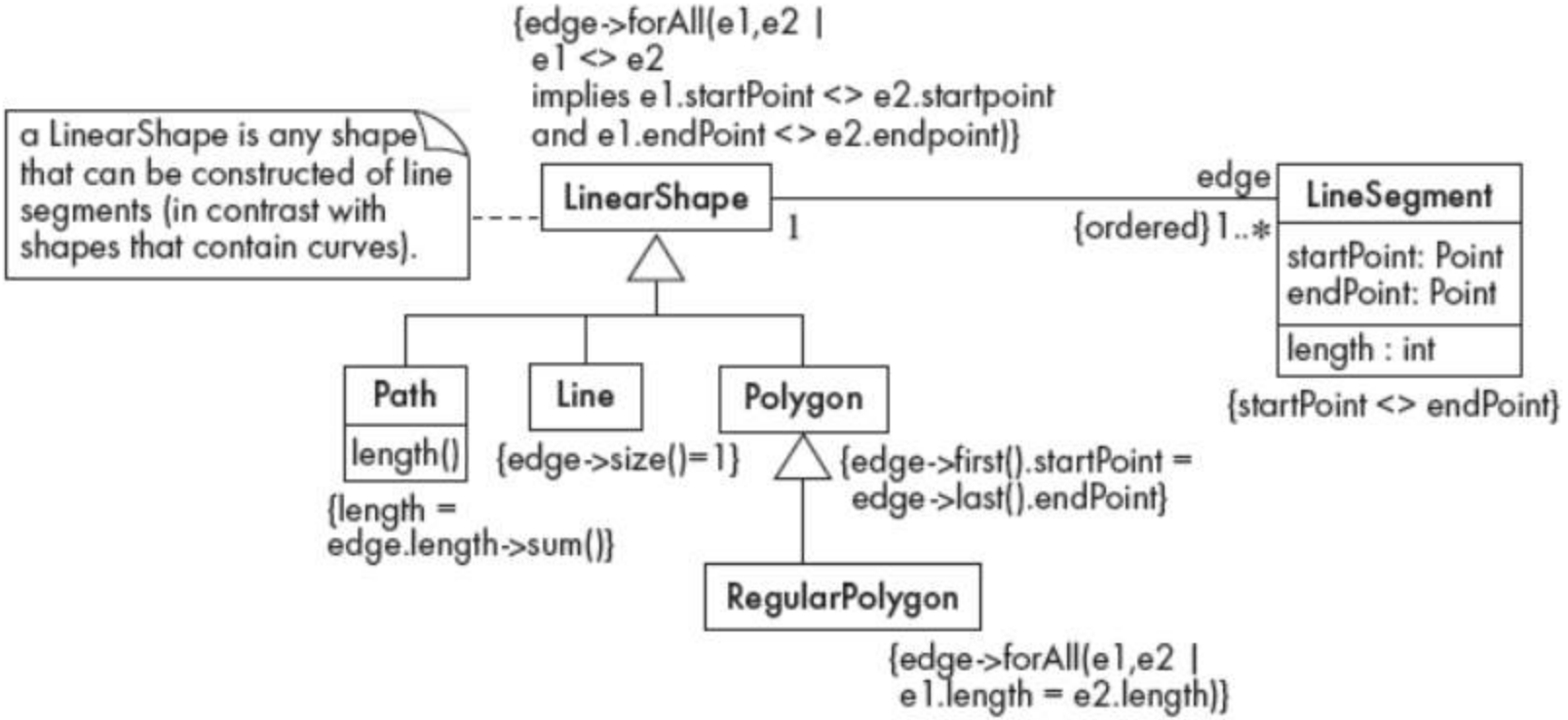
- A class needs to know about the other class in order use it's objects
- When the UserInterface wants to display, it accesses BlogEntry



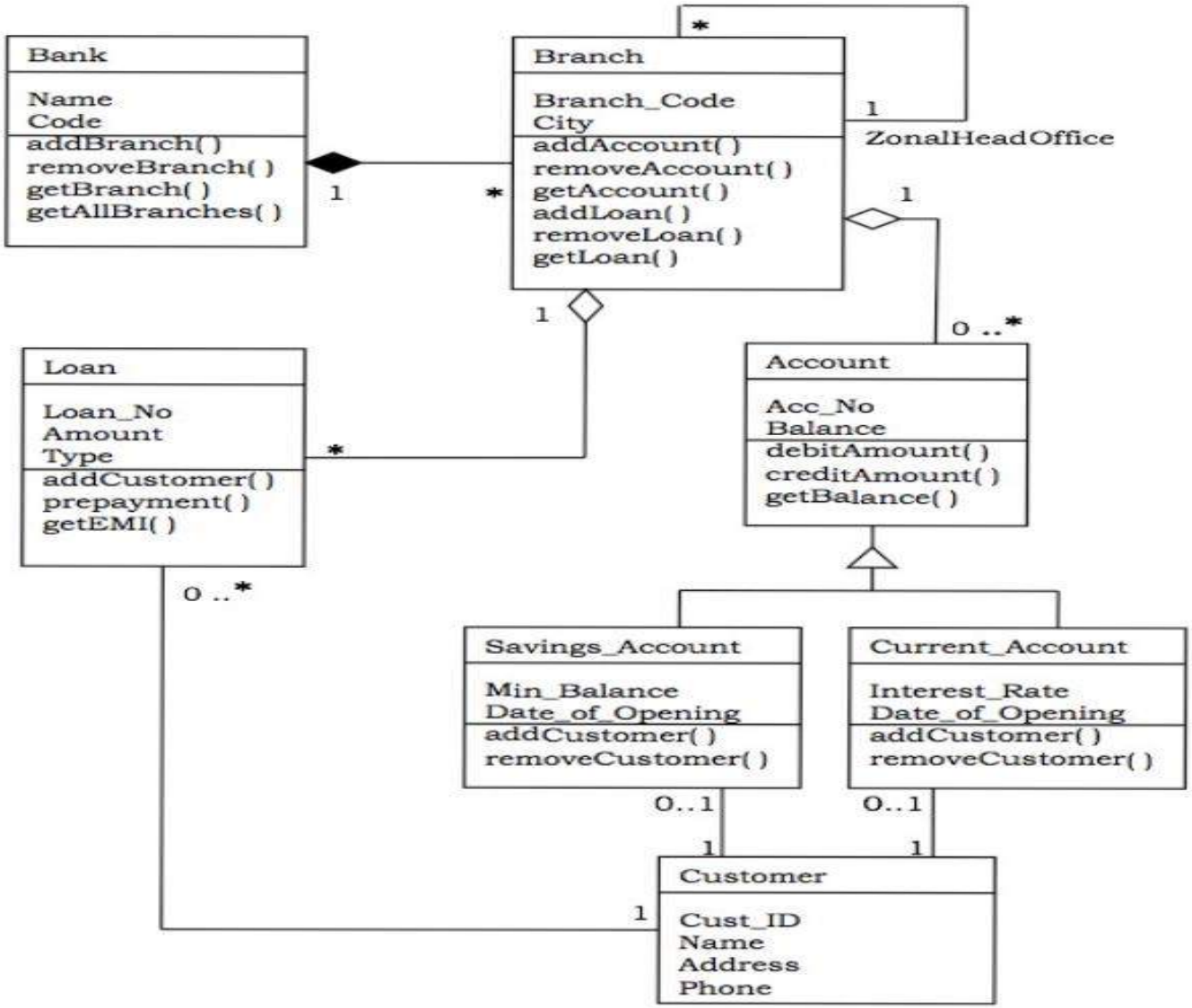
- Dependency implies only that the classes can work together, so is the weakest relationship

Notes and descriptive text

- **Descriptive text and other diagrams**
  - Embed your diagrams in a larger document
  - Text can explain aspects of the system using any notation you like
  - Highlight and expand on important features, and give rationale
- **Notes**:
  - A note is a small block of text embedded *in* a UML diagram
  - It acts like a comment in a programming language
- Constraints:
  - A constraint is like a note, except that it is written in a formal language that can be interpreted by a computer
  - Recommended language is Object Constraint Language


{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered} 1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}

An Example


Bank
Name
Code
addBranch( )
removeBranch( )
getBranch( )
getAllBranches( )
Branch
Branch_Code
City
addAccount( )
removeAccount( )
getAccount( )
addLoan( )
removeLoan( )
getLoan( )
ZonalHeadOffice
Loan
Loan_No
Amount
Type
addCustomer( )
prepayment( )
getEMI( )
Account
Acc_No
Balance
debitAmount( )
creditAmount( )
getBalance( )
Savings_Account
Min_Balance
Date_of_Opening
addCustomer( )
removeCustomer( )
Current_Account
Interest_Rate
Date_of_Opening
addCustomer( )
removeCustomer( )
Customer
Cust_ID
Name
Address
Phone
1
*
0 ..*
0..1

Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the entire process until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces

- *Don't be too disorganized. Don't be too rigid either.*

A simple technique for discovering domain classes

- Look at a source material such as a description of requirements
- Extract the *nouns* and *noun phrases*
- Eliminate nouns that:
  - are redundant
  - represent instances
  - are vague or highly general
  - not needed in the application
- Pay attention to classes in a domain model that represent *types of users* or other actors

Identifying associations and attributes

- Start with classes you think are most **central** and important
- Decide on the clear and obvious data it must contain and its relationships to other classes.
- Work outwards towards the classes that are less important.
- Avoid adding many associations and attributes to a class
  - A system is simpler if it manipulates less information

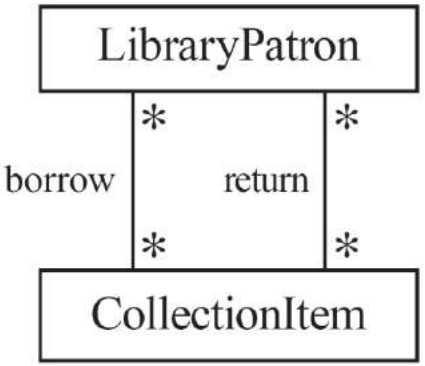
Tips about identifying and specifying valid associations

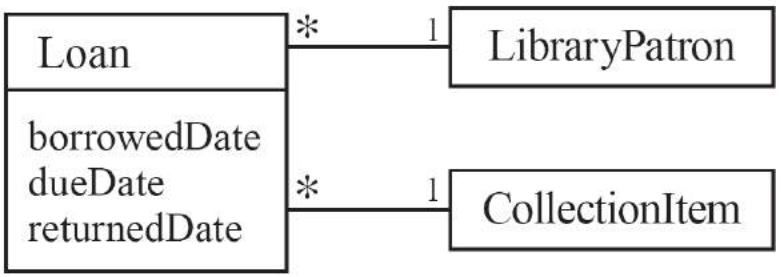
- An association should exist if a class
  - *possesses*
  - *controls*
  - *is connected to*
  - *is related to*
  - *is a part of*
  - *has as parts*
  - *is a member of*, or
  - *has as members*

  some other class in your model
- Specify the multiplicity at both ends
- Label it clearly.

Actions versus associations

- A common mistake is to represent *actions* as if they were associations



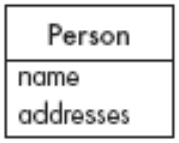


Identifying attributes

- Look for information that must be maintained about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
  - E.g. string, number

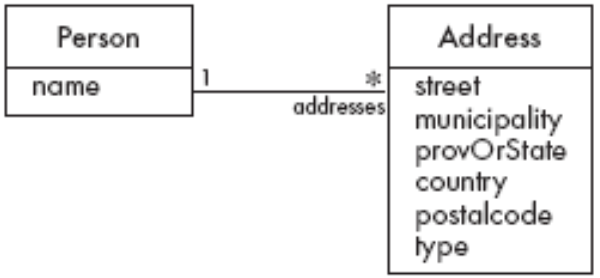
Tips about identifying and specifying valid attributes

- It is not good to have many duplicate attributes
- If a subset of a class's attributes form a coherent group, then create a distinct class containing these attributes



Bad, due to a plural attribute

Bad, due to too many attributes, and the inability to add more addresses



Good solution. The type indicates whether it is a home address, business address etc.

Identifying generalizations and interfaces

- There are two ways to identify generalizations:
  - bottom-up
    - Group together similar classes creating a new superclass
  - top-down
    - Look for more general classes first, specialize them if needed
- Create an *interface*, instead of a superclass if
  - The classes are very dissimilar except for having a few operations in common
  - One or more of the classes already have their own superclasses
  - Different implementations of the same class might be available

Allocating responsibilities to classes

- A *responsibility* is something that the system is required to do.
  - Each functional requirement must be attributed to one of the classes
    - All the responsibilities of a given class should be *clearly related*.
    - If a class has too many responsibilities, consider *splitting* it into distinct classes
    - If a class has no responsibilities attached to it, then it is probably *useless*
    - When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created
  - To determine responsibilities
    - Perform use case analysis
    - Look for verbs and nouns describing *actions* in the system description

Categories of responsibilities

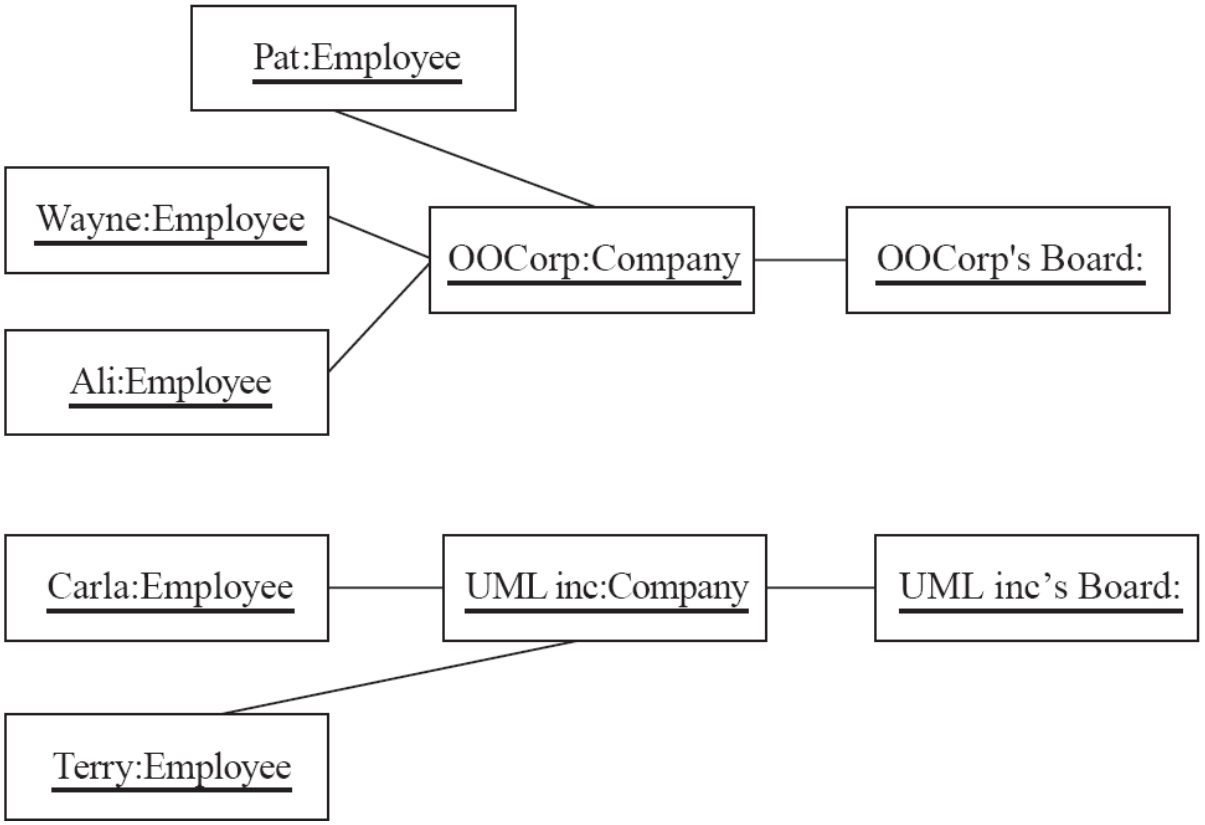
- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work

Identifying operations

- Operations are needed to realize the responsibilities of each class
  - There may be several operations per responsibility
  - The main operations that implement a responsibility are normally declared **public**
  - Other methods that collaborate to perform the responsibility must be as private as possible

Object Diagrams

- A *link* is an instance of an association
  - In the same way that we say an object is an instance of a class



Associations versus generalizations in object diagrams

- Associations describe the relationships that will exist between *instances* at run time.
  - When you show an instance diagram generated from a class diagram, there will be an instance of *both* classes joined by an association
- Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses

Modelling Interactions and Behaviour
Sequence Diagram

Interaction Diagrams

- Interaction diagrams are used to model the dynamic aspects of a software system
  - They help you to visualize how the system runs.
  - An interaction diagram is often built from a use case and a class diagram.
    - The objective is to show how a set of objects accomplish the required interactions with an actor.

Interactions and messages

- Interaction diagrams show how a set of actors and objects communicate with each other to perform:
  - The steps of a use case, or
  - The steps of some other piece of functionality.
- The set of steps, taken together, is called **an *interaction***.
- Interaction diagrams can show several different types of communication.
  - E.g. method calls, messages send over the network
  - These are all referred to as *messages*.

Elements found in interaction diagrams

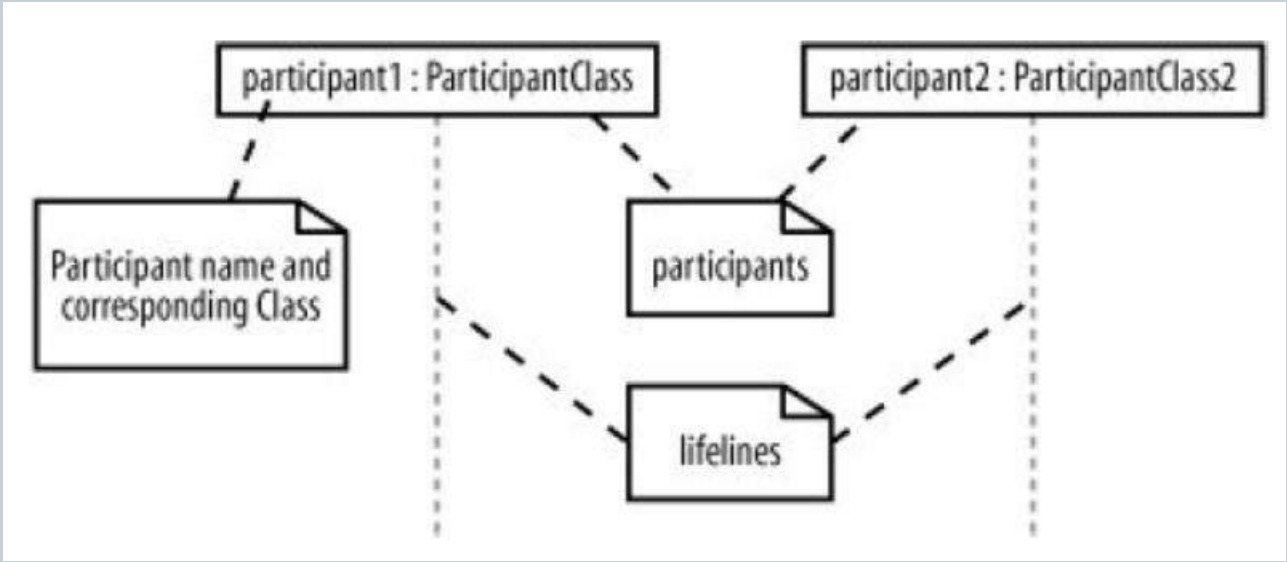
- Instances of classes
  - Shown as boxes with the class and object identifier underlined
- Actors
  - Use the stick-person symbol as in use case diagrams
- Messages
  - Shown as arrows from actor to object, or from object to object

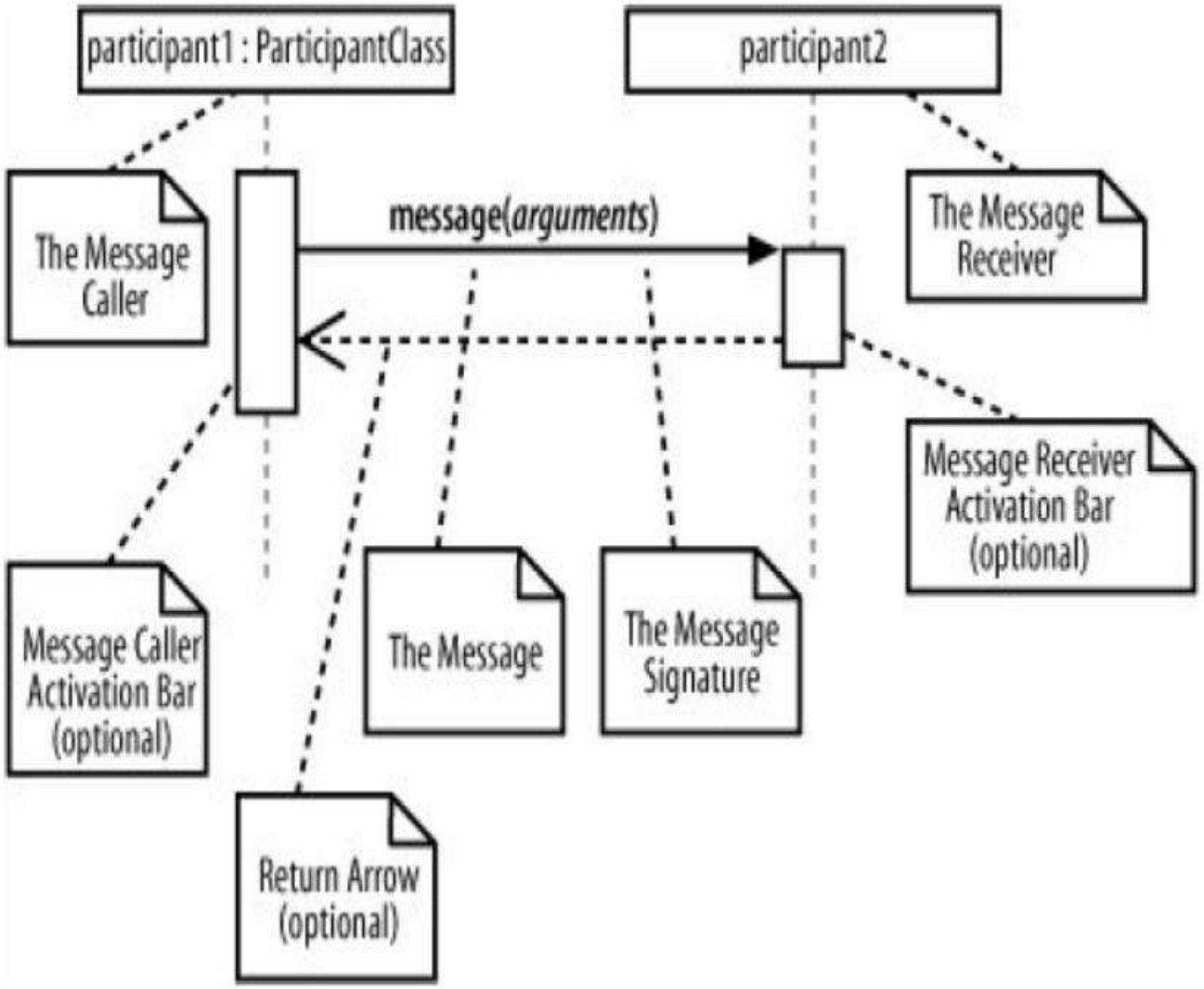
Creating interaction diagrams

- You should develop a class diagram and a use case model before starting to create an interaction diagram.
  - Important interaction diagrams:
    - *Sequence diagrams*
    - *Communication diagrams*
    - *Timing diagrams*

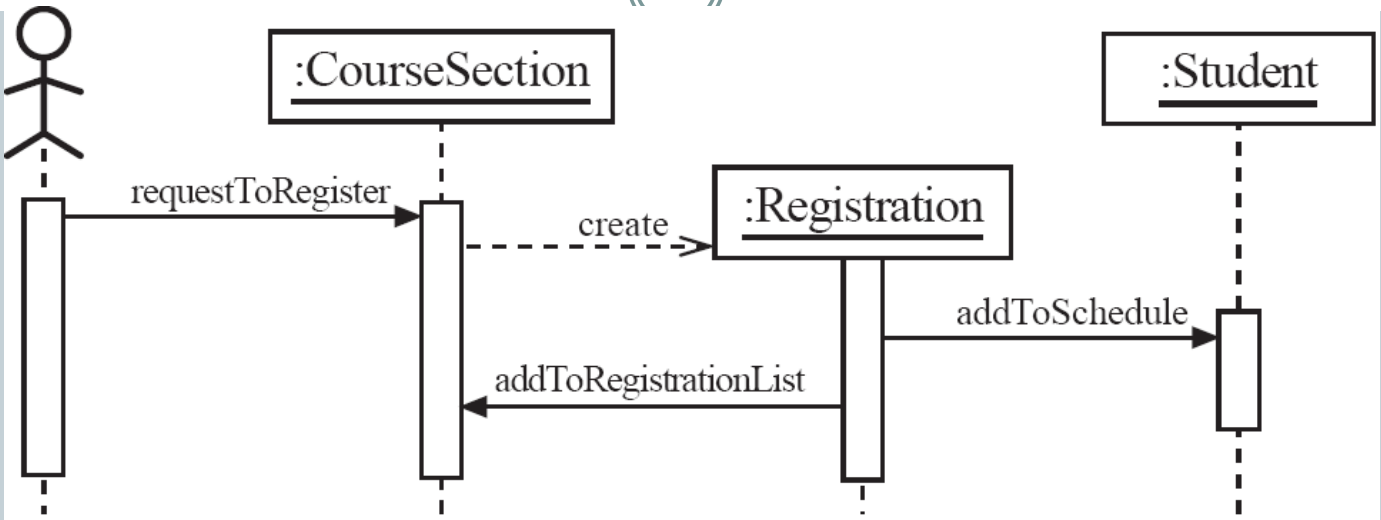
Sequence diagrams

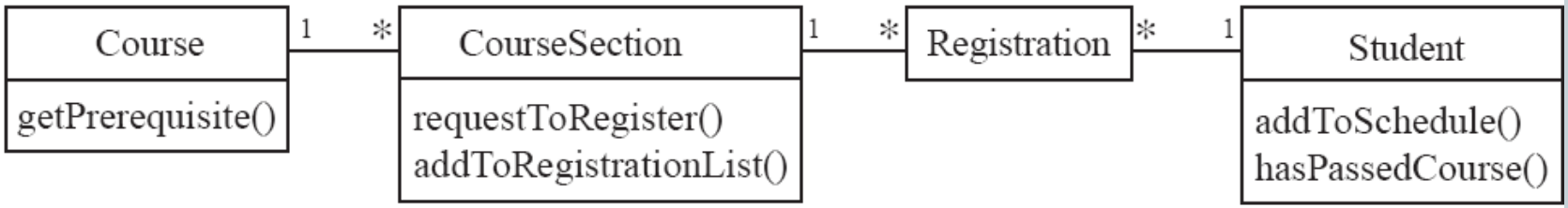
- A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task
  - The objects are arranged horizontally across the diagram.
  - An actor that initiates the interaction is often shown on the left.
  - The vertical dimension represents time.
  - A vertical line, called a *lifeline*, is attached to each object or actor.
  - The lifeline becomes a broad box, called an *activation box* during the *live activation* period.
  - A message is represented as an arrow between activation boxes of the sender and receiver.
    - A message is labelled and can have an argument list and a return value.


participant1 : ParticipantClass
participant2 : ParticipantClass2
Participant name and corresponding Class
participants
lifelines

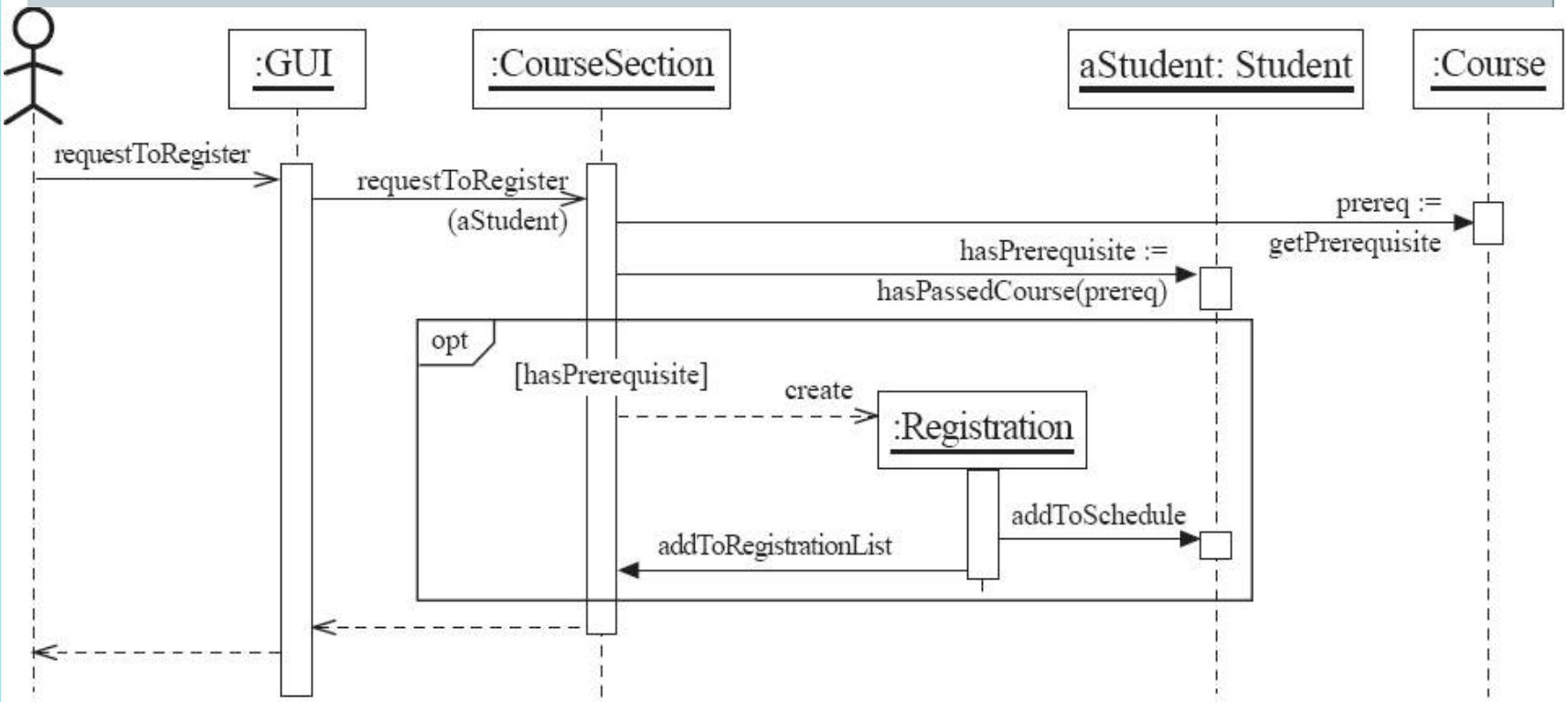

participant1 : ParticipantClass
participant2
The Message Caller
message(arguments)
The Message Receiver
Message Receiver Activation Bar (optional)
Message Caller Activation Bar (optional)
The Message
The Message Signature
Return Arrow (optional)

Sequence diagrams – an example

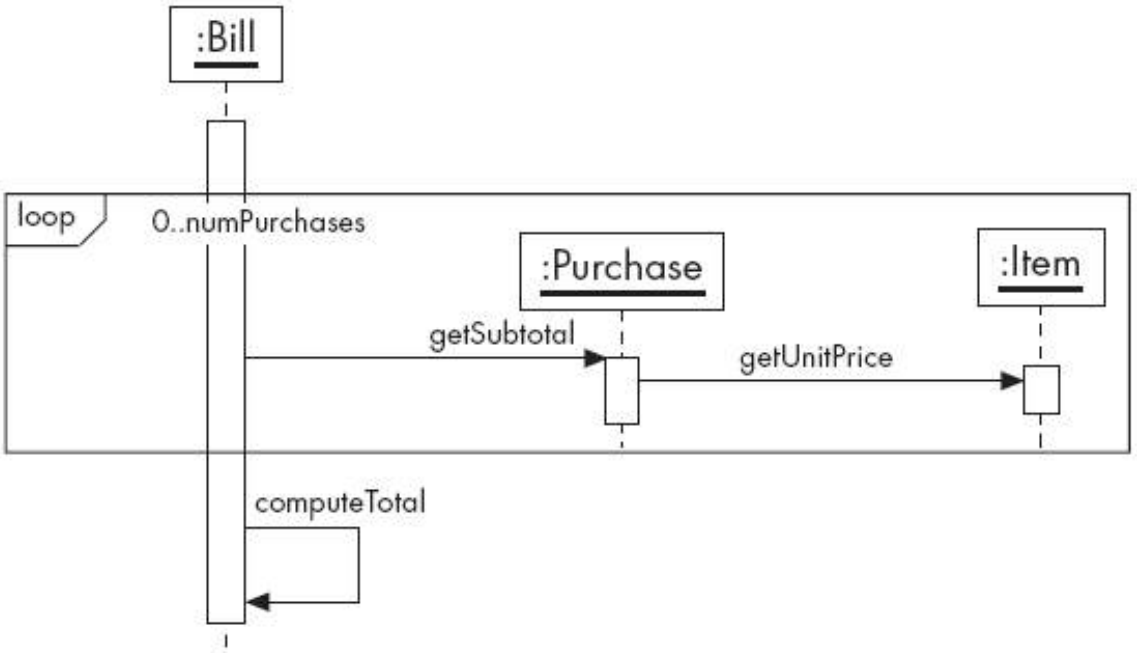


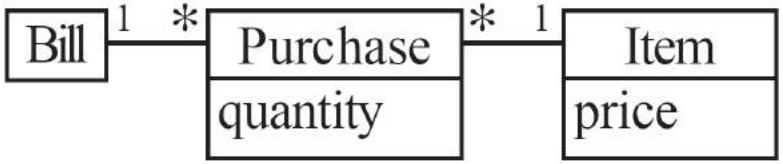


Sequence diagrams – same example, more details



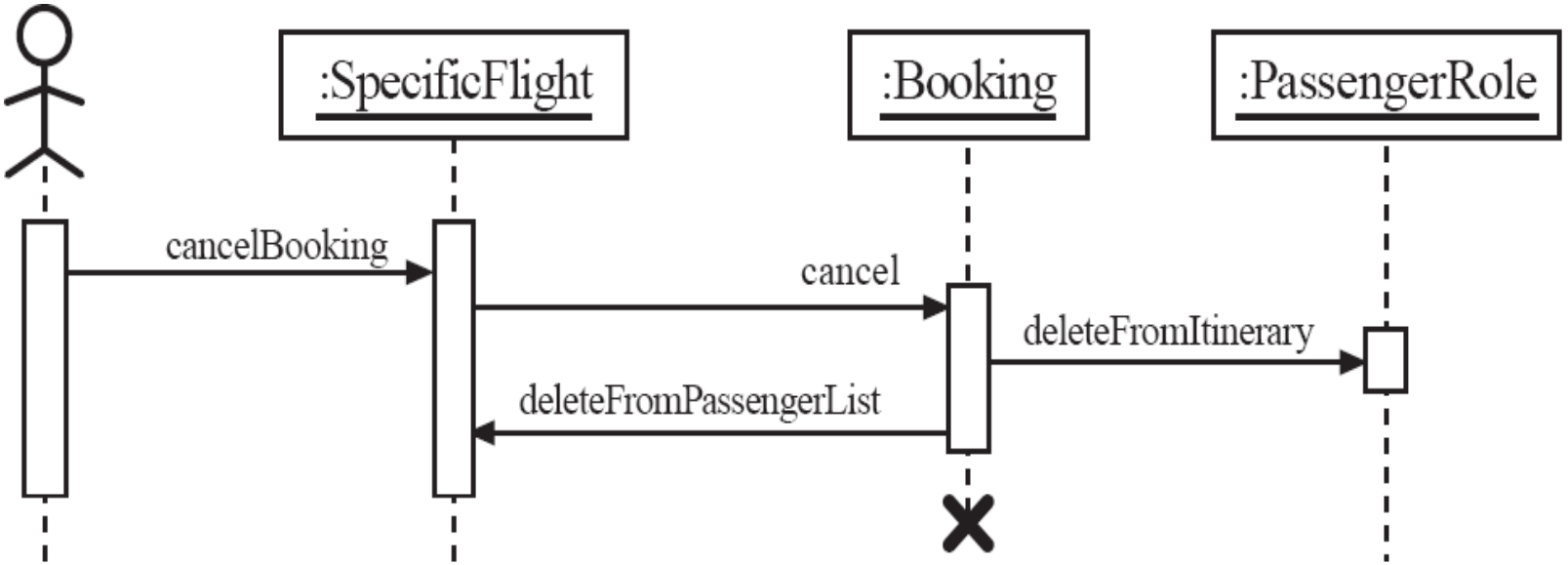
Sequence diagrams – an example with replicated messages





Sequence diagrams – an example with object deletion

- If an object's life ends, this is shown with an X at the end of the lifeline



Quiz

- On 2nd March 2021, after the lecture
  - Class Diagram
  - Sequence Diagram

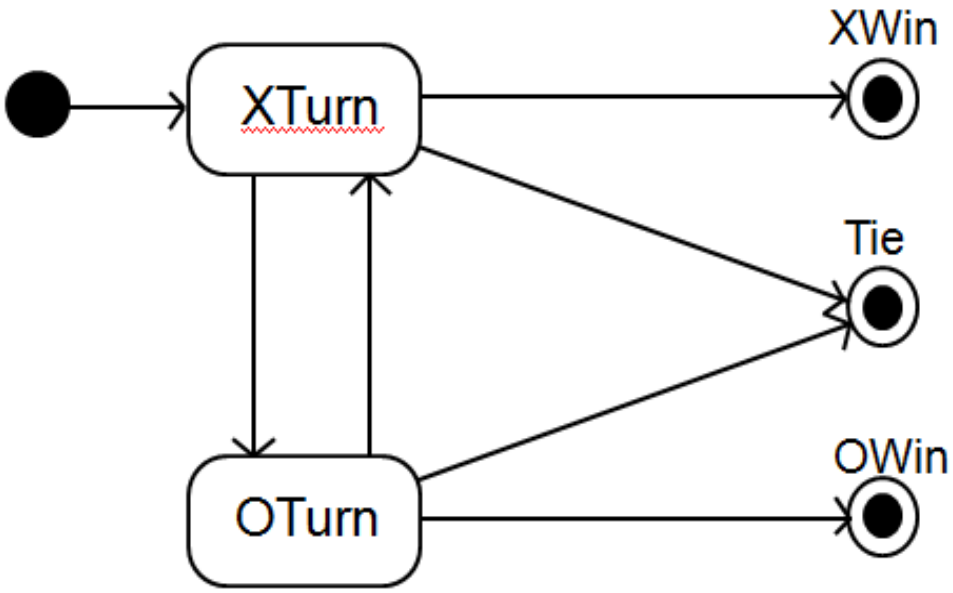
STATE DIAGRAM AND ACTIVITY DIAGRRAMS

State Diagrams

- **A state diagram describes the behaviour of a *system*, some *part* of a system, or an *individual object*.**
  - At any given point in time, the system or object is in a certain ***state***.
    - Being in a state means that it is will behave in a *specific way* in response to any events that occur.
  - Some events will cause the system to change state.
    - In the new state, the system will behave in a different way to events.
  - A state diagram is a directed graph where the nodes are states and the arcs are transitions.

State diagrams – an example

- tic-tac-toe game



States

- At any given point in time, the system is in one state.
- It will remain in this state until an event occurs that causes it to change state.
- A state is represented by a **rounded rectangle** containing the name of the state.
- Special states:
  - A black circle represents the ***start state***
  - A circle with a ring around it represents an ***end state***

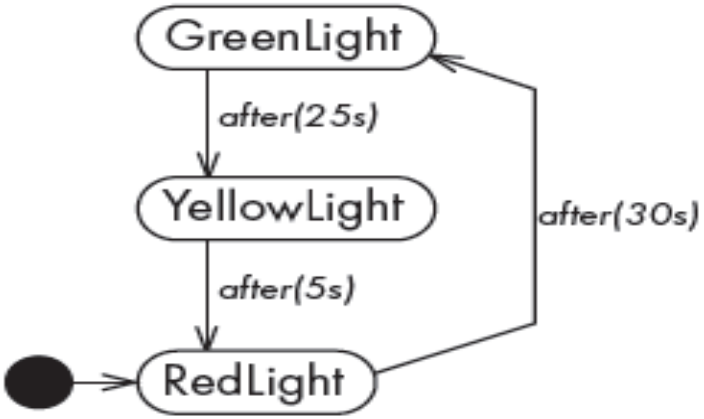
Transitions

- A transition represents a change of state in response to an event.
  - It is considered to occur instantaneously.

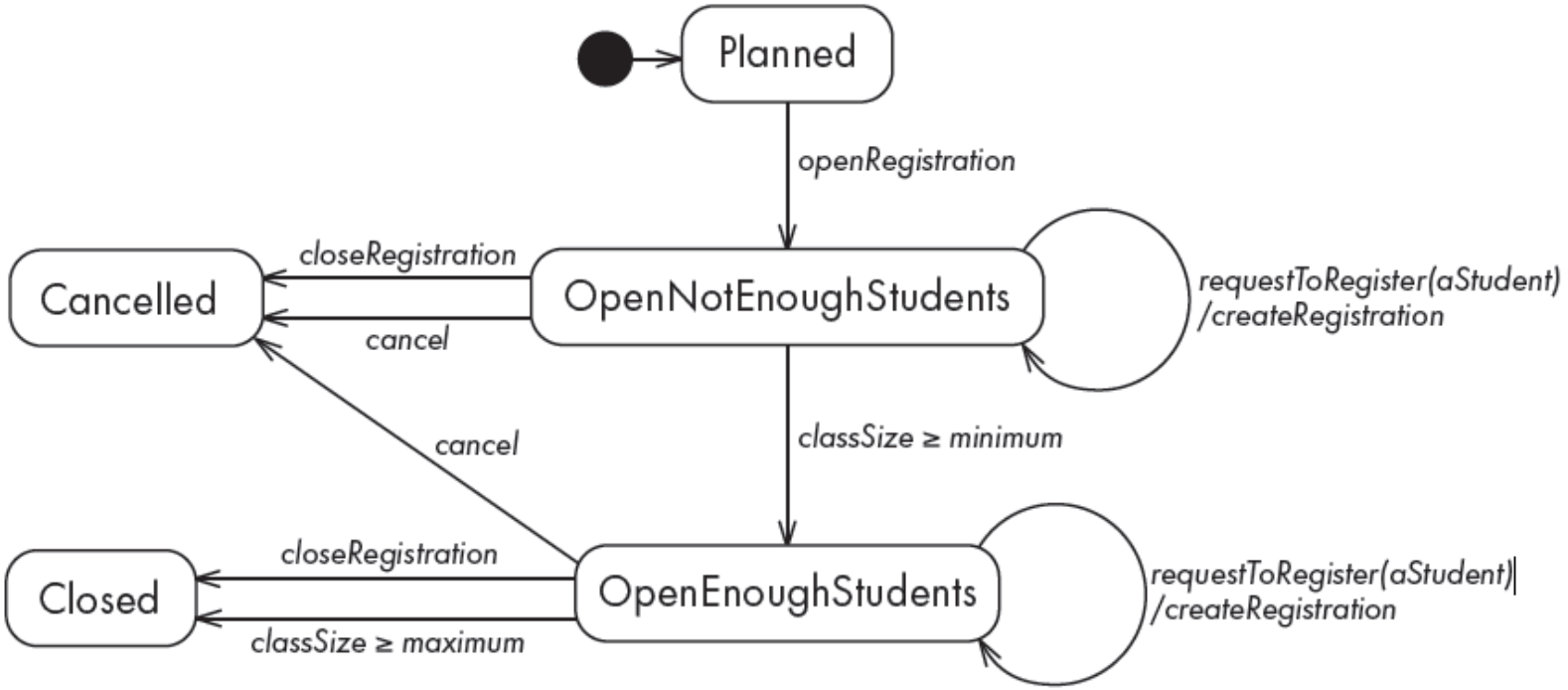
- The label on each transition is the event that causes the change of state.
- A transition is rendered as a **solid directed line**.

State diagrams – an example of transitions with time-outs and conditions

State diagrams of a simple traffic light, illustrating elapsed-time transitions



State diagrams – an example with conditional transitions



State diagram of a CourseSection class

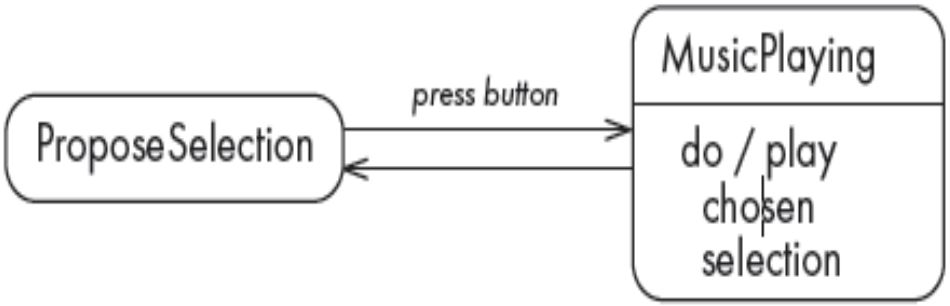
Activities in state diagrams

- An *activity* is something that takes place while the system is *in* a state.
  - It takes a period of time.
  - The system may take a transition out of the state in response to completion of the activity,
  - Some other outgoing transition may result in:
    - The interruption of the activity, and
    - An early exit from the state.

Activity representation

- An activity is shown textually within a state box by the word '**do**' followed by a '**/**' symbol, and a ***description*** of what is to be done.
- When you have details such as actions in a state, you draw a horizontal line above them to separate them from the state name.

State diagram – an example with activity



State diagram for a jukebox, illustrating an activity in a state

Actions in state diagrams

- An *action* is something that takes place effectively *instantaneously*
  - When a particular transition is taken,
  - Upon entry into a particular state, or
  - Upon exit from a particular state

- ***An action should consume no noticeable amount of time***

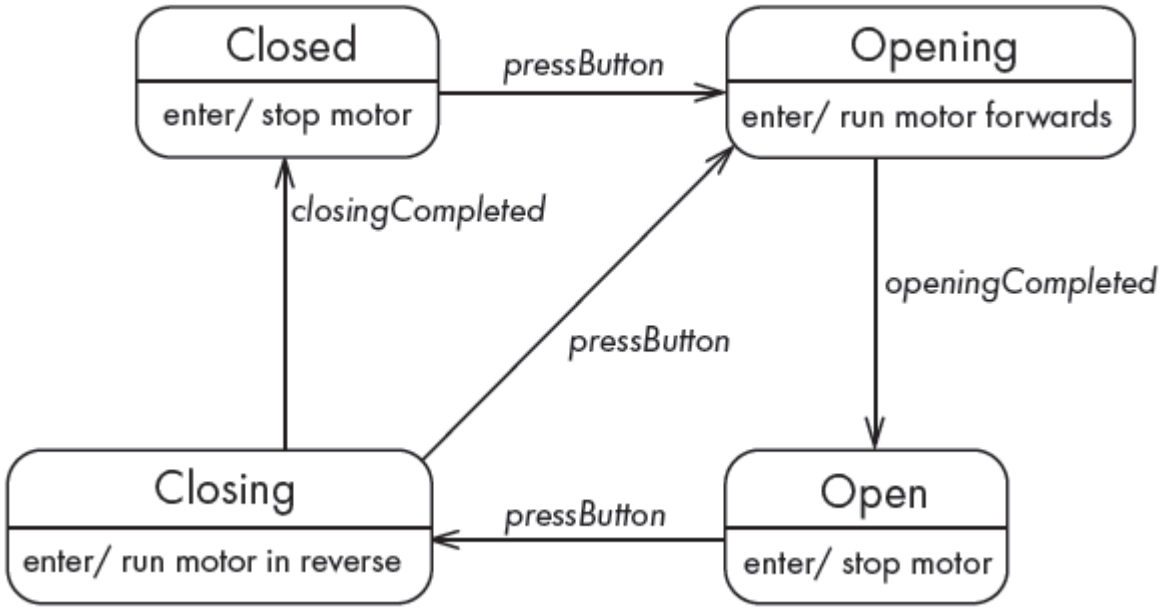
- It should be something simple, such as sending a message, starting a hardware device or setting a variable.

Representation of action

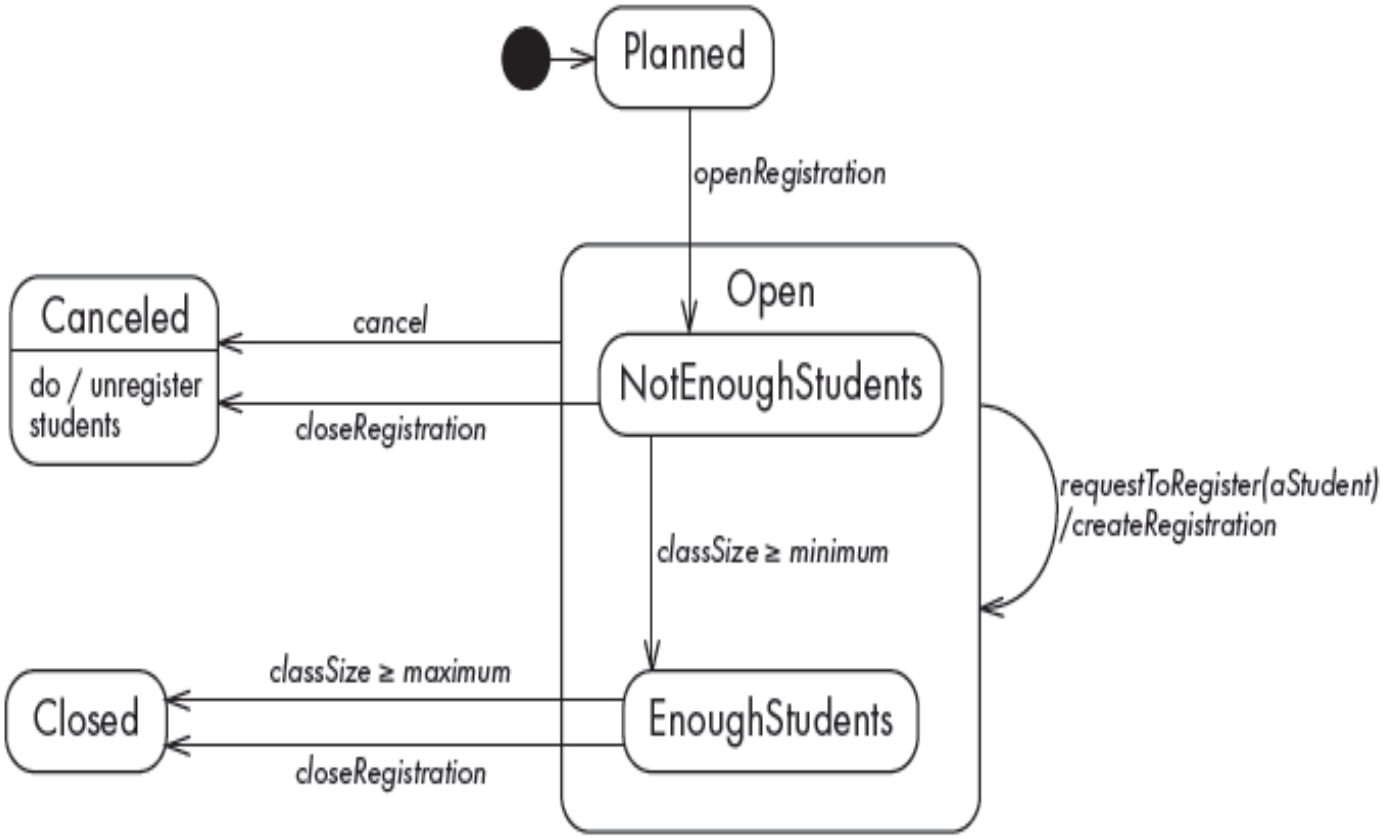
- An action is always shown preceded by a slash (‘/’) symbol.
- If the action is to be performed during a transition, then the syntax is **event/action**.
- If the action is to be performed when entering or exiting a state, then it is written in the state box with the notation **enter/action** or **exit/action**.

State diagram – an example with actions

- **State diagram for a garage door opener, showing actions triggered by entry into a state**



State diagram – an example with substates



A version of the course section example from Figure 8.14, showing the effect of nested states

Activity Diagrams

- **Activity diagram** is another important behavioural diagram in **UML** diagram to describe dynamic aspects of the system.
- The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario.
  - Activity diagram is essentially an advanced version of flow chart that modelling the flow from one activity to another activity.
  - An *activity diagram* is like ***a state diagram***.
    - Except most transitions are caused by *internal* events, such as the completion of a computation.

Activity Diagrams.....

- An activity diagram
  - Can be used to understand the flow of work that an object or component performs.
  - Can also be used to visualize the interrelation and interaction between different use cases.
  - Is most often associated with several classes.

- One of the strengths of activity diagrams is the representation of ***concurrent*** **activities**.

Activity Diagram Notation

- Activity diagram uses ***rounded rectangles*** to imply a specific system function
- ***Arrows*** to represent flow through the system.
- ***Decision diamonds*** to depict a branching decision (each arrow emanating from the diamond is labelled).
- ***Solid horizontal lines*** to indicate that parallel activities are occurring.

Activity Diagram Notation .....

Activity

Is used to represent a set of actions

Control Flow

Shows the sequence of execution

Initial Node

Portrays the beginning of a set of actions or activities

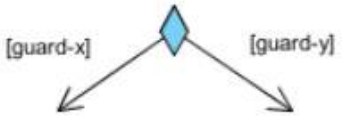
Activity Final Node

Stop all control flows and object flows in an activity (or action)

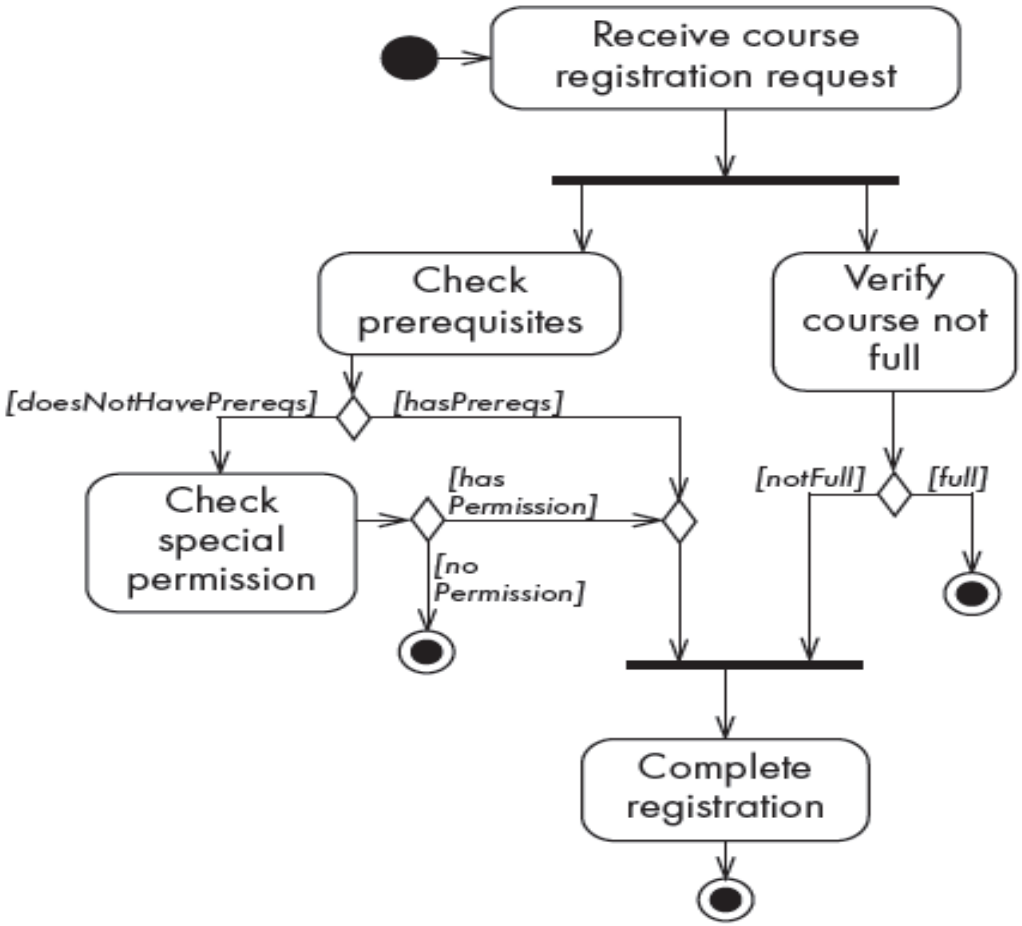
Decision nodes and merge nodes

- An activity diagram has two types of nodes for branching within a single thread. These are represented as small ***diamonds***:
  - ***Decision node***
    - *has one incoming transition and multiple outgoing transitions* each with a Boolean guard in square brackets. Exactly one of the outgoing transitions will be taken.
  - ***Merge node***
    - *has two incoming transitions and one outgoing transition. It is* used to bring together paths that had been split by decision nodes.

Activity Diagram Notation .....

Decision Node Represent a test condition to ensure that the control flow or object flow only goes down one path	
Merge Node Bring back together different decision paths that were created using a decision-node.	

Activity diagrams – an example



Activity diagram of the registration process

Representing concurrency

- Concurrency is shown using **forks, joins and rendezvous.**
  - A ***fork*** has one incoming transition and multiple outgoing transitions.
    - The execution splits into two concurrent threads.
  - A ***rendezvous*** has multiple incoming and multiple outgoing transitions.
    - Once all the incoming transitions occur all the outgoing transitions may occur.

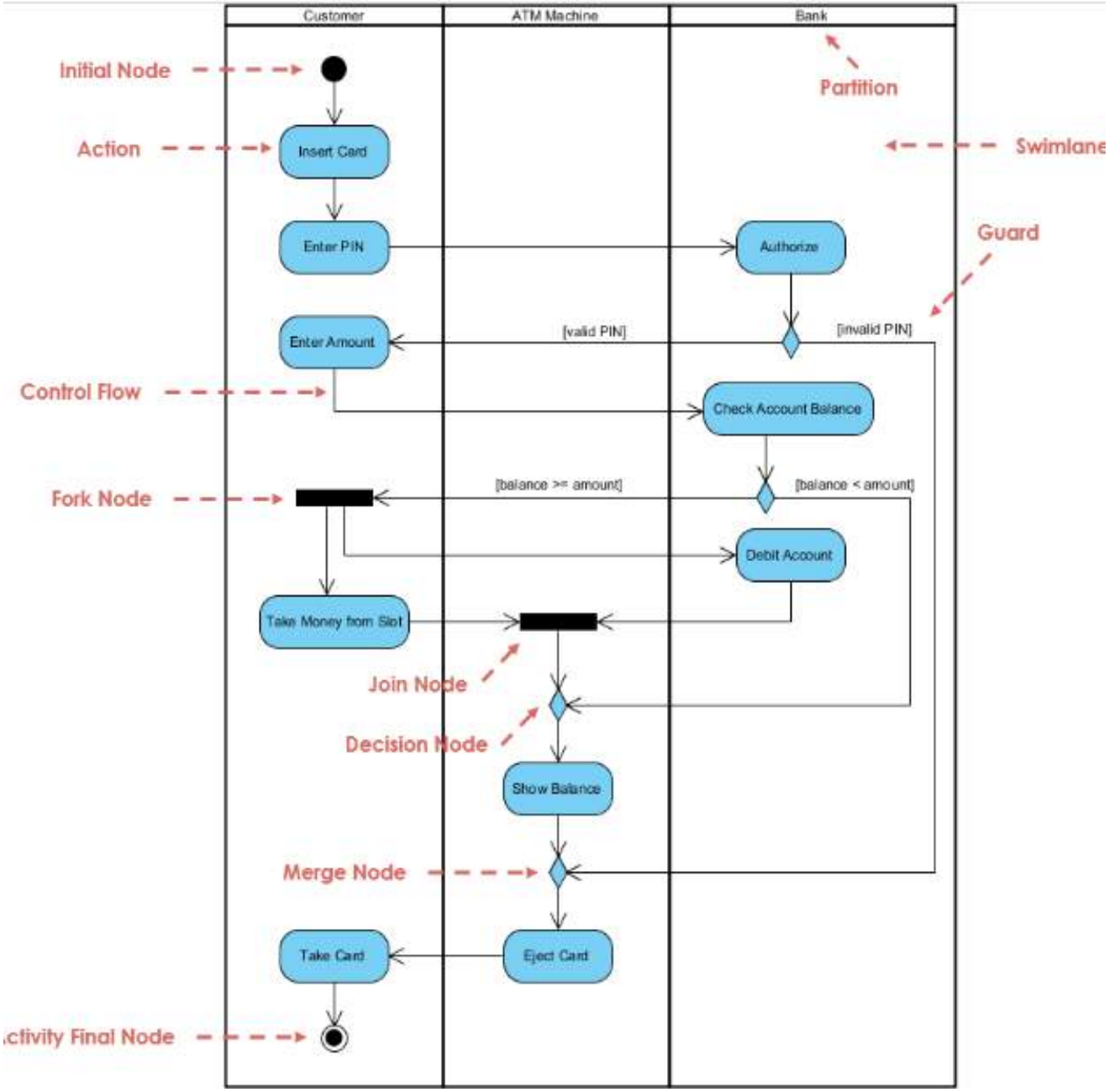
Representing concurrency

- A ***join*** has multiple incoming transitions and one outgoing transition.
  - The outgoing transition will be taken when all incoming transitions have occurred.
  - The incoming transitions must be triggered in separate threads.
  - If one incoming transition occurs, a wait condition occurs at the join until the other transitions occur.

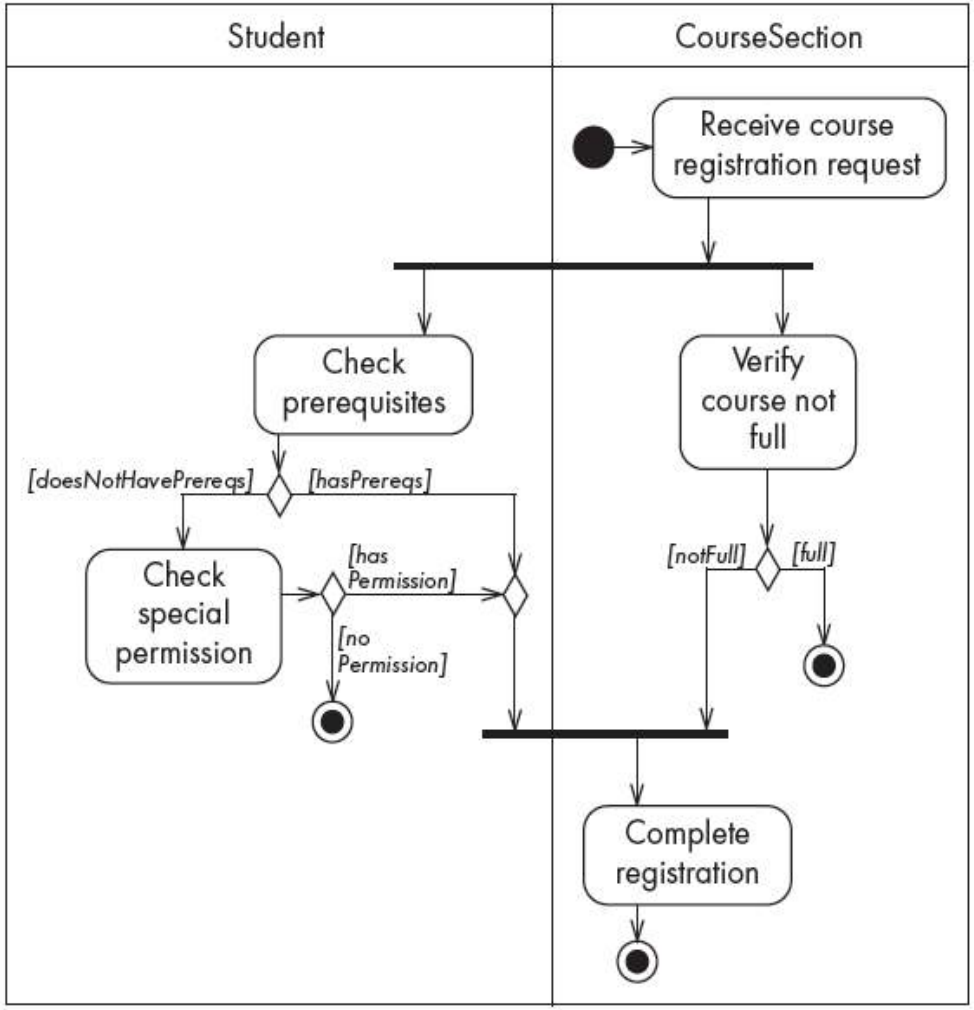
Swimlanes

- Activity diagrams are most often associated with several classes.
- The partition of activities among the existing classes can be explicitly shown in an activity diagram by the introduction of ***swimlanes.***
  - *Allows* you to represent the flow of activities described by the use case
  - indicate which actor or analysis class has responsibility for the action described by an activity rectangle.

- Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.


Customer
ATM Machine
Bank
Initial Node
Partition
Action
Swimlane
Insert Card
Enter PIN
Authorize
Guard
Enter Amount
[valid PIN]
[invalid PIN]
Control Flow
Check Account Balance
Fork Node
[balance >= amount]
[balance < amount]
Debit Account
Take Money from Slot
Join Node
Decision Node
Show Balance
Merge Node
Take Card
Eject Card
ctivity Final Node

Activity diagrams – an example with swimlanes



Activity diagram with swimlanes

Design Patterns
Architectural Patterns

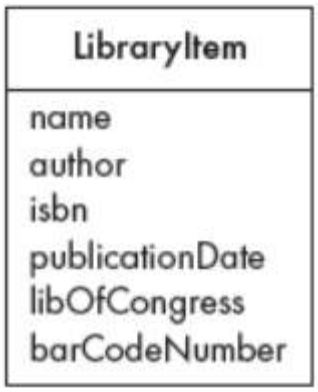
Introduction to Patterns

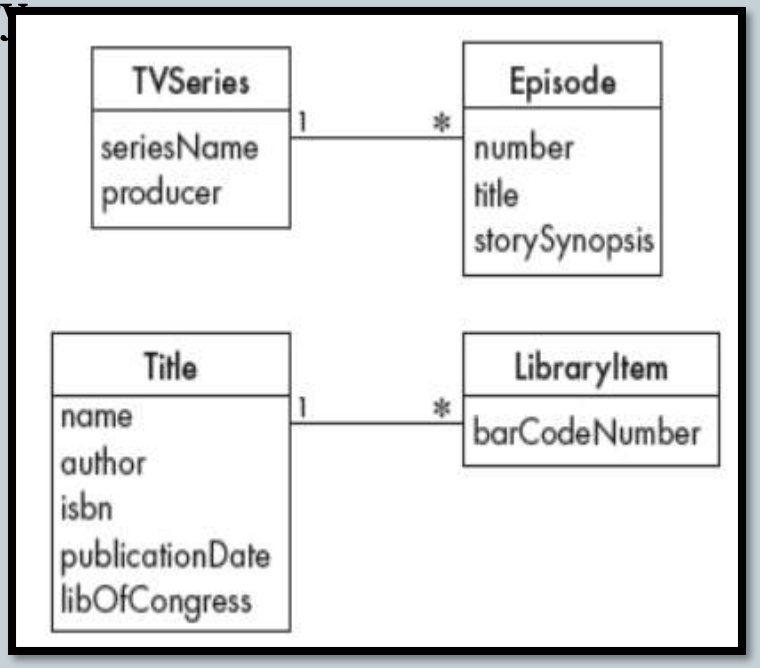
- The recurring aspects of designs are called *design patterns*.
  - A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context
  - Many of them have been systematically documented for all software developers to use
  - A good pattern should
    - Be as general as possible
    - Contain a solution that has been proven to effectively solve the problem in the indicated context.

Studying patterns is an effective way to learn from the experience of others

An Example

- All the episodes of a television series.
- The flights that leave at the same time every day for the same destination.
- All the copies of the same book in a library.





Pattern description

Context:
- The general situation in which the pattern applies

Problem:
- A short sentence or two raising the main difficulty.

Forces:
- The issues or concerns to consider when solving the problem

Solution:
- The recommended way to solve the problem in the given context.
  - 'to balance the forces'

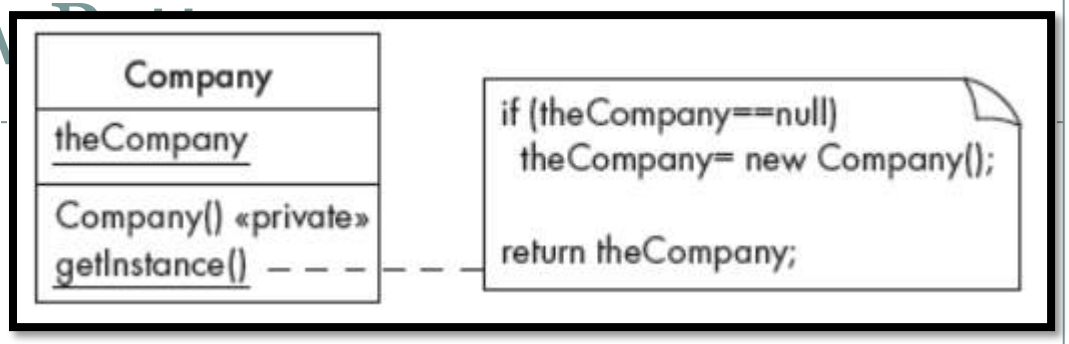
Antipatterns: (Optional)
- Solutions that are inferior or do not work in this context.

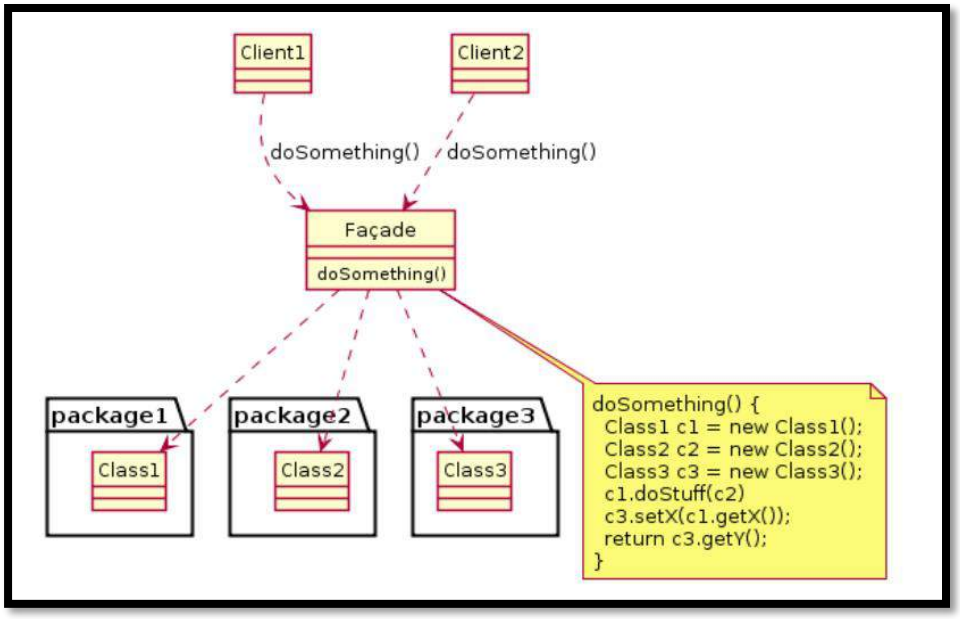
Related patterns: (Optional)
- Patterns that are similar to this pattern.

References:
- Who developed or inspired the pattern.

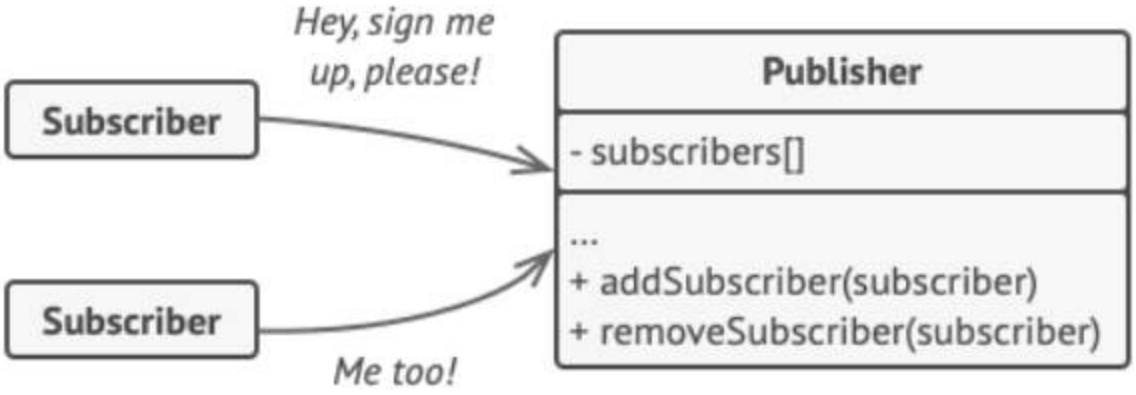
Few P

- Creational
  - Eg : Singleton
- Structural
  - Eg : Facade
- Behavioral
  - Eg : Observer





The Observer Pattern

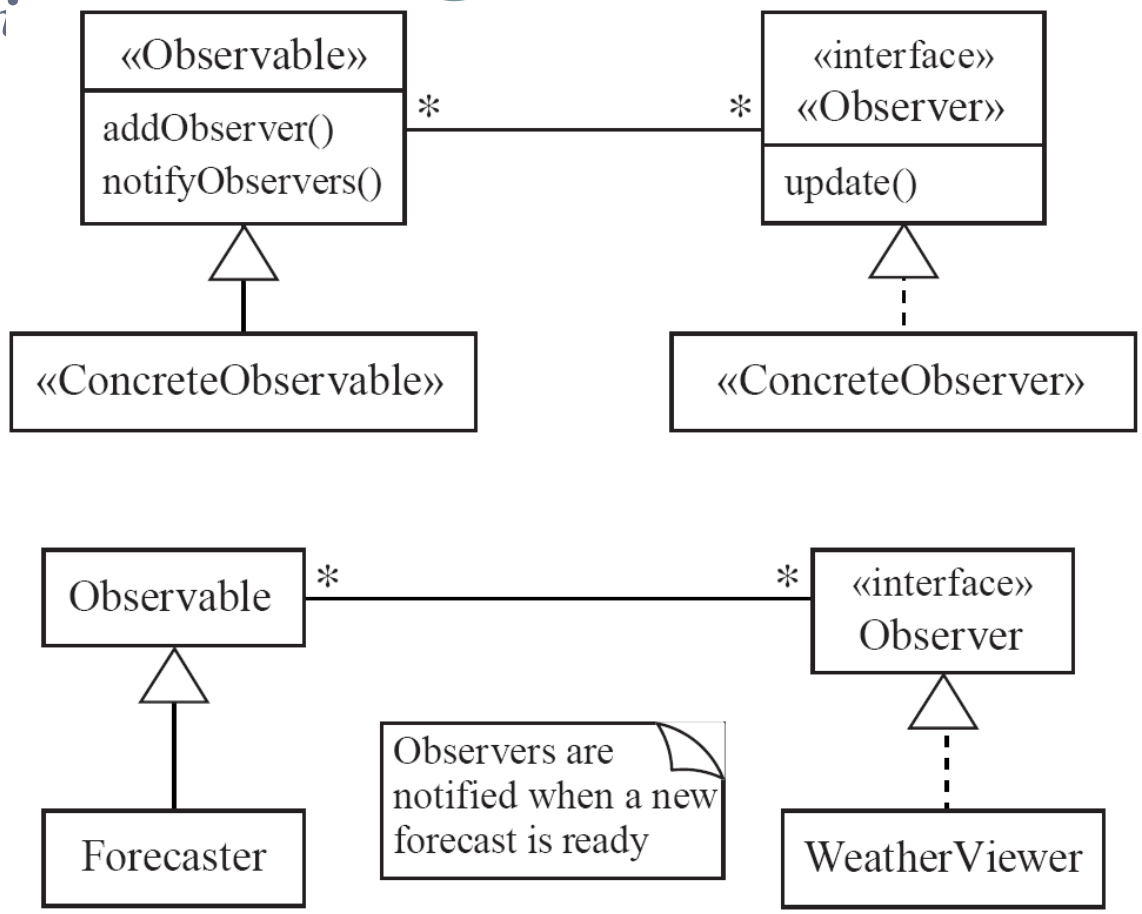


The Observer Pattern

- ***Context***:
  - When an association is created between two classes, the code for the classes becomes inseparable.
  - If you want to reuse one class, then you also have to reuse the other.
- ***Problem***:
  - How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?
- ***Forces***:
  - You want to maximize the flexibility of the system to the greatest extent possible

Observer

- ***Soluti***



Observer

- Antipatterns:
  - Connect an observer directly to an observable so that they both have references to each other.
  - Make the observers *subclasses* of the observable.

- References
- The Observer pattern is one of the ‘Gang of Four’ patterns. It is also widely known as Publish-and-Subscribe (the observers are Subscribers and the
- observable is a Publisher).

Software Architecture

- *Software architecture* is process of designing the global organization of a software system, including:
  - Dividing software into subsystems.
  - Deciding how these will interact.
  - Determining their interfaces.
    - The architecture is the **core of the design**, so all software engineers need to understand it.
    - The architecture will often constrain the overall efficiency, reusability and maintainability of the system.

The importance of software architecture

- Why you need to develop an architectural model:
  - To enable everyone to better understand the system
  - To allow people to work on individual pieces of the system in isolation
  - To prepare for extension of the system
  - To facilitate reuse and reusability

Contents of a good architectural model

- A system's architecture will often be expressed in terms of several different *views*
  - The logical breakdown into subsystems
  - The interfaces among the subsystems
  - The dynamics of the interaction among components at run time
  - The data that will be shared among the subsystems
  - The components that will exist at run time, and the machines or devices on which they will be located

Design *stable* architecture

- To ensure the maintainability and reliability of a system, an architectural model must be designed to be *stable*.
  - Being stable means that the new features can be easily added with only small changes to the architecture

Developing an architectural model

- Start by sketching an outline of the architecture
  - Based on the principal requirements and use cases
  - Determine the main components that will be needed
  - Choose among the various architectural patterns
    - Discussed next
  - *Suggestion*: have several different teams independently develop a first draft of the architecture and merge together the best ideas

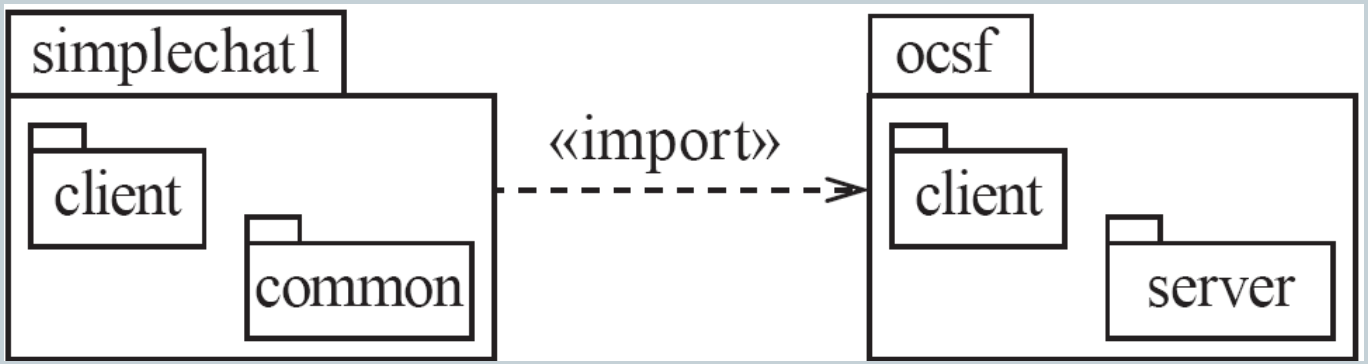
Developing an architectural model

- Refine the architecture
  - Identify the main ways in which the components will interact and the interfaces between them
  - Decide how each piece of data and functionality will be distributed among the various components
  - Determine if you can re-use an existing framework, if you can build a framework
- Consider each use case and adjust the architecture to make it realizable
- Mature the architecture

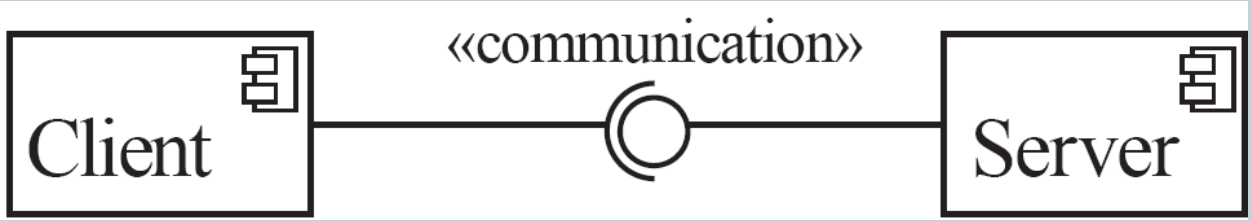
Describing an architecture using UML

- All UML diagrams can be useful to describe aspects of the architectural model
- UML diagrams are particularly suitable for architecture modelling:
  - Package diagrams
  - Component diagrams
  - Deployment diagrams

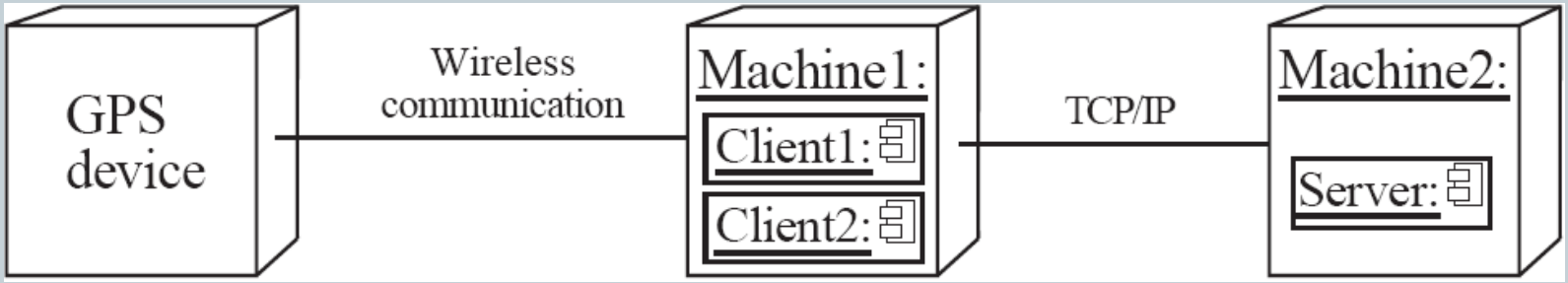
Package diagrams



Component diagrams



Deployment diagrams



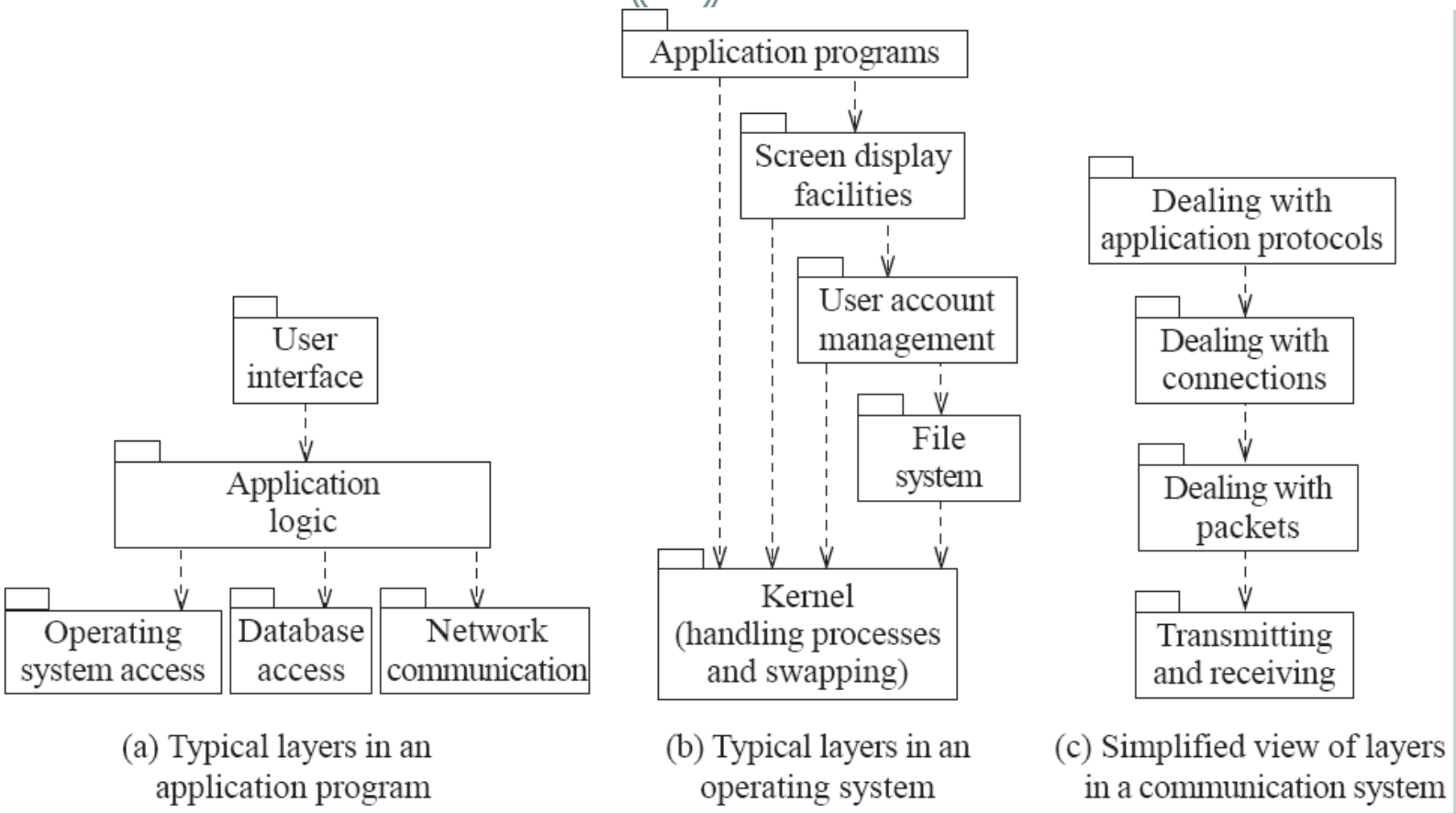
Architectural Patterns

- The notion of patterns can be applied to software architecture.
  - These are called *architectural patterns* or *architectural styles*.
  - Each allows you to design flexible systems using components
    - The components are as independent of each other as possible.

The Multi-Layer architectural pattern

- In a layered system, each layer communicates only with the layer immediately below it.
  - Each layer has a well-defined interface used by the layer immediately above.
    - The higher layer sees the lower layer as a set of *services*.
  - A complex system can be built by superposing layers at increasing levels of abstraction.
    - Layers immediately below the UI layer provide the application functions determined by the use-cases.
    - Bottom layers provide general services.
      - e.g. network communication, database access

Example of multi-layer systems



(a) Typical layers in an application program

(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system

The multi-layer architecture and design principles

1. *Divide and conquer*: The layers can be independently designed.
2. *Increase cohesion*: Well-designed layers have layer cohesion.
3. *Reduce coupling*: Well-designed lower layers do not know about the higher layers and the only connection between layers is through the API.
4. *Increase abstraction*: you do not need to know the details of how the lower layers are implemented.
5. *Increase reusability*: The lower layers can often be designed generically.

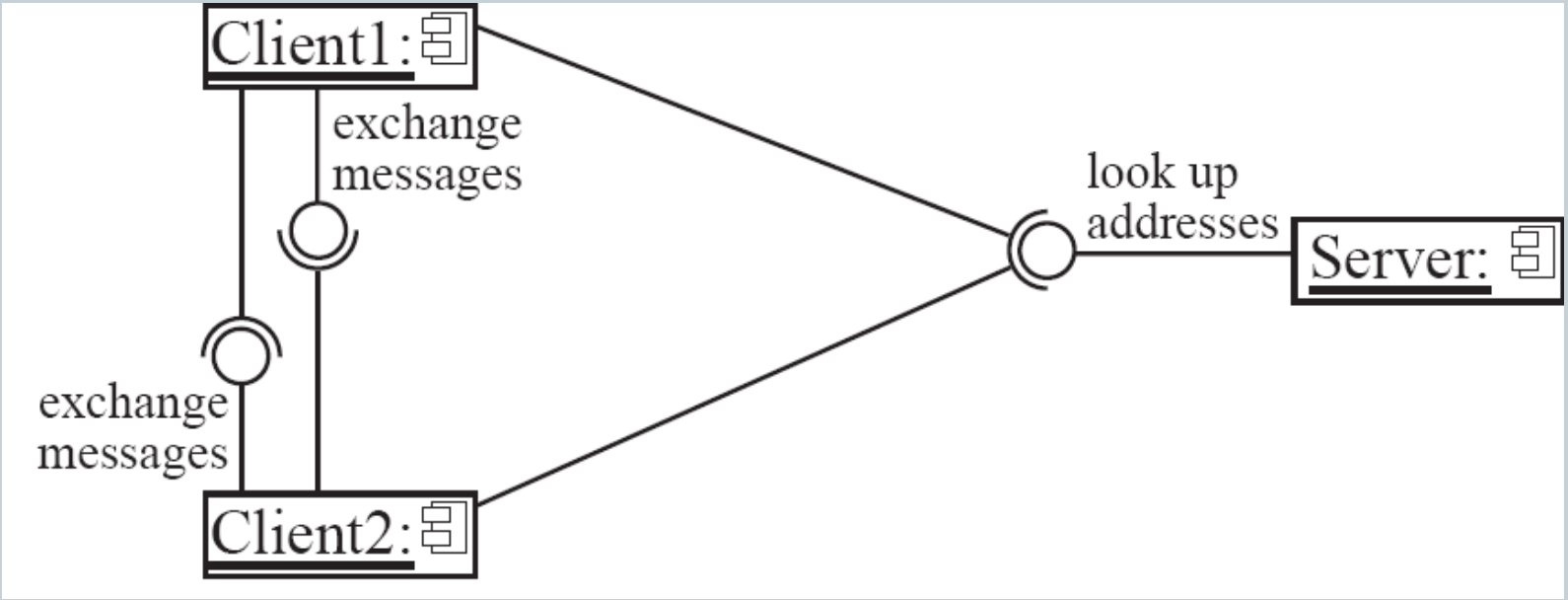
The multi-layer architecture and design principles

6. *Increase reuse*: You can often reuse layers built by others that provide the services you need.
7. Increase flexibility: you can add new facilities built on lower-level services, or replace higher-level layers.
8. *Anticipate obsolescence*: By isolating components in separate layers, the system becomes more resistant to obsolescence.
9. *Design for portability*: All the dependent facilities can be isolated in one of the lower layers.
10. *Design for testability*: Layers can be tested independently.
11. *Design defensively*: The APIs of layers are natural places to build in rigorous assertion-checking.

The Client-Server and other distributed architectural patterns

- There is at least one component that has the role of *server*, waiting for and then handling connections.
- There is at least one component that has the role of *client*, initiating connections in order to obtain some service.
- A further extension is the Peer-to-Peer pattern.
  - A system composed of various software components that are distributed over several hosts.

An example of a distributed system



The distributed architecture and design principles

1. *Divide and conquer*: Dividing the system into client and server processes is a strong way to divide the system.
   - Each can be separately developed.
2. *Increase cohesion*: The server can provide a cohesive service to clients.
3. *Reduce coupling*: There is usually only one communication channel exchanging simple messages.
4. *Increase abstraction*: Separate distributed components are often good abstractions.
6. *Increase reuse*: It is often possible to find suitable frameworks on which to build good distributed systems
   - However, client-server systems are often very application specific.

The distributed architecture and design principles

7. *Design for flexibility*: Distributed systems can often be easily reconfigured by adding extra servers or clients.
9. *Design for portability*: You can write clients for new platforms without having to port the server.
10. *Design for testability*: You can test clients and servers independently.
11. *Design defensively*: You can put rigorous checks in the message handling code.

The Broker architectural pattern

- Transparently distribute aspects of the software system to different nodes
  - An object can call methods of another object without knowing that this object is remotely located.
  - CORBA is a well-known open standard that allows you to build this kind of architecture.

Example of a Broker system

The broker architecture and design principles

1. *Divide and conquer*: The remote objects can be independently designed.

5. *Increase reusability*: It is often possible to design the remote objects so that other systems can use them too.

6. *Increase reuse*: You may be able to reuse remote objects that others have created.

7. *Design for flexibility*: The brokers can be updated as required, or the proxy can communicate with a different remote object.

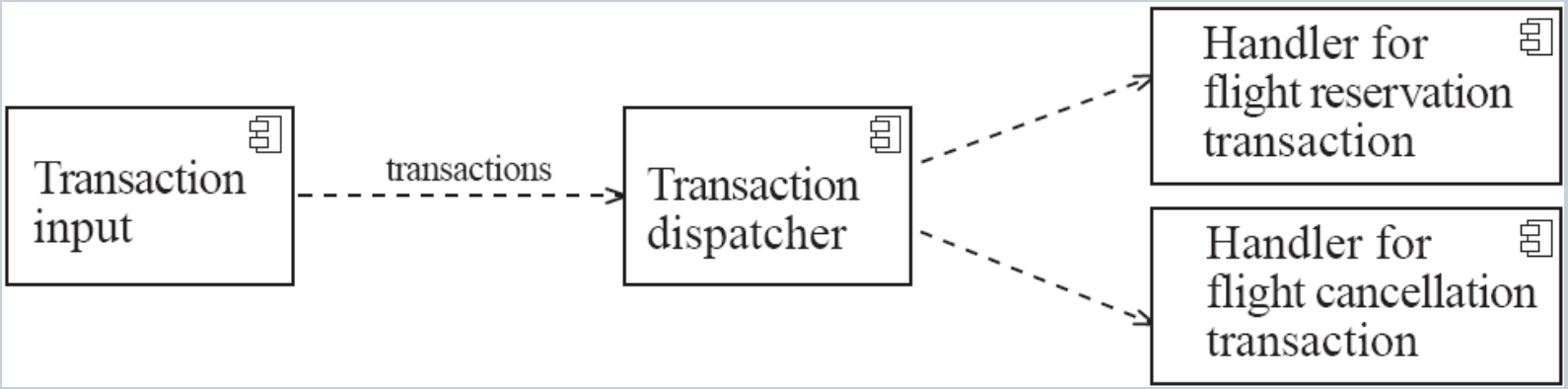
9. *Design for portability*: You can write clients for new platforms while still accessing brokers and remote objects on other platforms.

11. *Design defensively*: You can provide careful assertion checking in the remote objects.

The Transaction-Processing architectural pattern

- A process reads a series of inputs one by one.
  - Each input describes a *transaction* – a command that typically some change to the data stored by the system
  - There is a transaction *dispatcher* component that decides what to do with each transaction
  - This dispatches a procedure call or message to one of a series of component that will *handle* the transaction

Example of a transaction-processing system



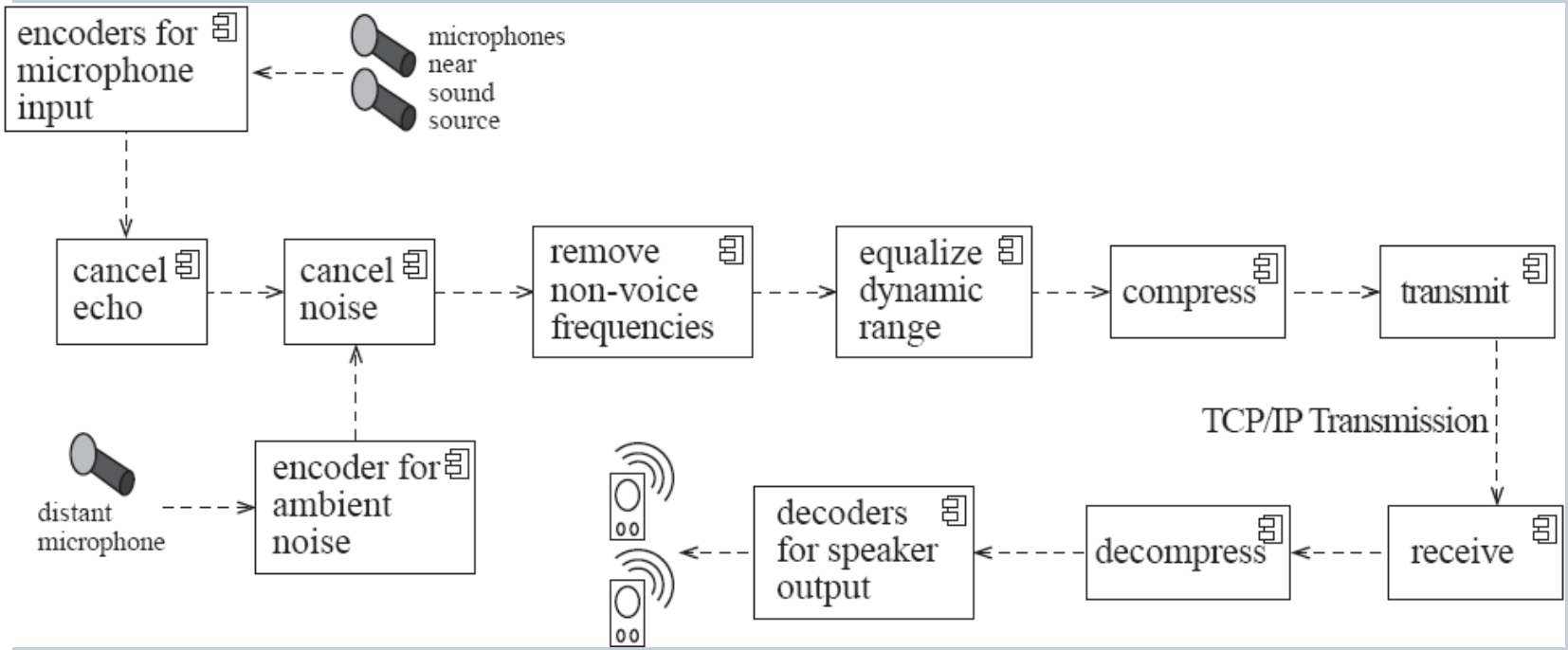
The transaction-processing architecture and design principles

1. *Divide and conquer*: The transaction handlers are suitable system divisions that you can give to separate software engineers.
2. *Increase cohesion*: Transaction handlers are naturally cohesive units.
3. *Reduce coupling*: Separating the dispatcher from the handlers tends to reduce coupling.
7. *Design for flexibility*: You can readily add new transaction handlers.
11. *Design defensively*: You can add assertion checking in each transaction handler and/or in the dispatcher.

The Pipe-and-Filter architectural pattern

- A stream of data, in a relatively simple format, is passed through a series of processes
  - Each of which transforms it in some way.
  - Data is constantly fed into the pipeline.
  - The processes work concurrently.
  - The architecture is very flexible.
    - Almost all the components could be removed.
    - Components could be replaced.
    - New components could be inserted.
    - Certain components could be reordered.

Example of a pipe-and-filter system



The pipe-and-filter architecture and design principles

1. *Divide and conquer*: The separate processes can be independently designed.
2. *Increase cohesion*: The processes have functional cohesion.
3. *Reduce coupling*: The processes have only one input and one output.
4. *Increase abstraction*: The pipeline components are often good abstractions, hiding their internal details.
5. *Increase reusability*: The processes can often be used in many different contexts.
6. *Increase reuse*: It is often possible to find reusable components to insert into a pipeline.

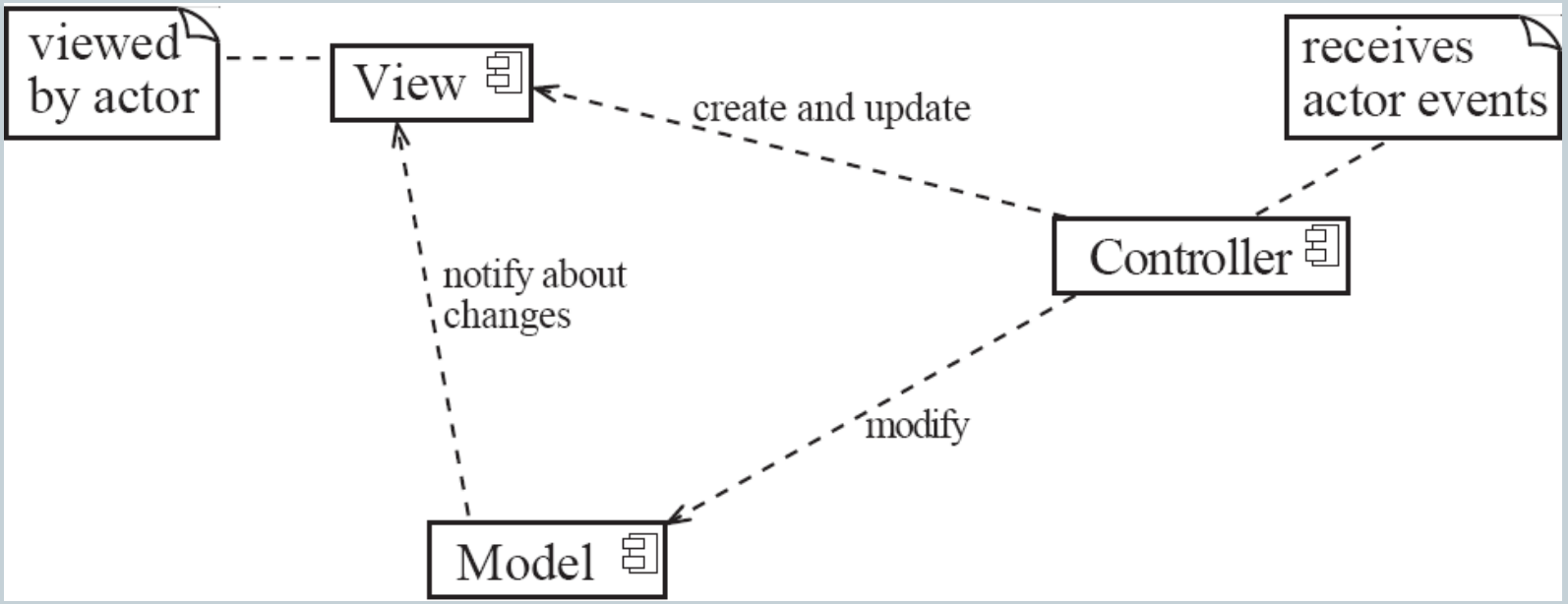
The pipe-and-filter architecture and design principles

7. *Design for flexibility*: There are several ways in which the system is flexible.
10. *Design for testability*: It is normally easy to test the individual processes.
11. *Design defensively*: You rigorously check the inputs of each component, or else you can use design by contract.

The Model-View-Controller (MVC) architectural pattern

- An architectural pattern used to help separate the user interface layer from other parts of the system
  - The *model* contains the underlying classes whose instances are to be viewed and manipulated
  - The *view* contains objects used to render the appearance of the data from the model in the user interface
  - The *controller* contains the objects that control and handle the user's interaction with the view and the model

Example of the MVC architecture for the UI



Example of MVC in Web architecture

- The *View* component generates the HTML code to be displayed by the browser.
- The *Controller* is the component that interprets 'HTTP post' transmissions coming back from the browser.
- The *Model* is the underlying system that manages the information.

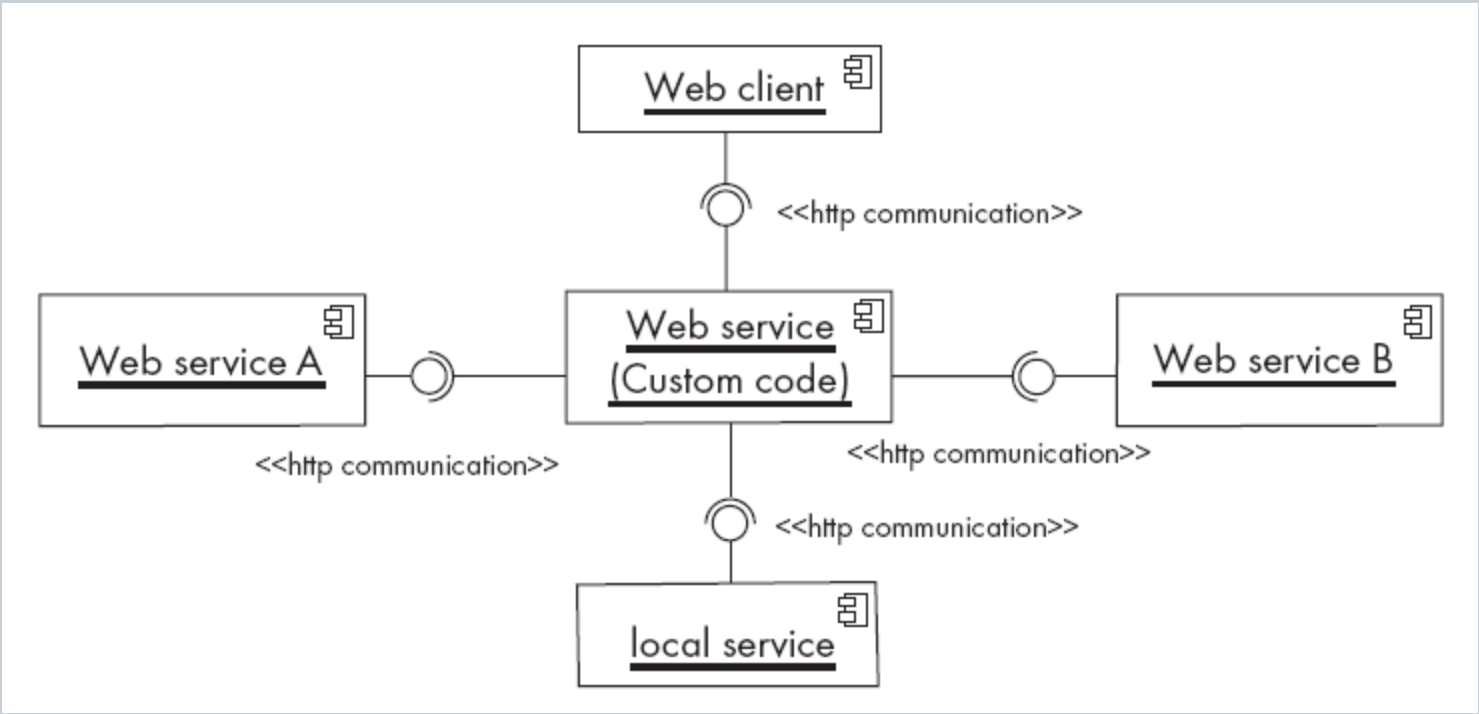
The MVC architecture and design principles

1. *Divide and conquer*: The three components can be somewhat independently designed.
2. *Increase cohesion*: The components have stronger layer cohesion than if the view and controller were together in a single UI layer.
3. *Reduce coupling*: The communication channels between the three components are minimal.
6. *Increase reuse*: The view and controller normally make extensive use of reusable components for various kinds of UI controls.
7. *Design for flexibility*: It is usually quite easy to change the UI by changing the view, the controller, or both.
10. *Design for testability*: You can test the application separately from the UI.

The Service-oriented architectural pattern

- This architecture organizes an application as a collection of services that communicates using well-defined interfaces
  - In the context of the Internet, the services are called *Web services*
  - A web service is an application, accessible through the Internet, that can be integrated with other services to form a complete system
  - The different components generally communicate with each other using open standards such as XML.

Example of a service-oriented application



The Service-oriented architecture and design principles

1. *Divide and conquer*: The application is made of independently designed services.
2. *Increase cohesion*: The Web services are structured as layers and generally have good functional cohesion.
3. *Reduce coupling*: Web-based applications are loosely coupled built by binding together distributed components.
5. *Increase reusability*: A Web service is a highly reusable component.
6. *Increase reuse*: Web-based applications are built by reusing existing Web services.
8. *Anticipate obsolescence*: Obsolete services can be replaced by new implementation without impacting the applications that use them.

The Service-oriented architecture and design principles

9. *Design for portability*: A service can be implemented on any platform that supports the required standards.
10. *Design for testability*: Each service can be tested independently.
11. *Design defensively*: Web services enforce defensive design since different applications can access the service.

Writing a Good Design Document

Writing a Good Design Document

- Design documents as an aid to making better designs
  - They force you to be explicit and consider the important issues before starting implementation.
  - They allow a group of people to review the design and therefore to improve it.
  - Design documents as a means of communication.
    - To those who will be *implementing* the design.
    - To those who will need, in the future, to *modify* the design.
    - To those who need to create systems or subsystems that *interface* with the system being designed.

Structure of a design document

A. Purpose:

- What system or part of the system this design document describes.
- Make reference to the requirements that are being implemented by this design (*traceability)* .

B. General priorities:

- Describe the priorities used to guide the design process.

C. Outline of the design:

- Give a high-level description of the design that allows the reader to quickly get a general feeling for it.

D. Major design issues:

- Discuss the important issues that had to be resolved.
- Give the possible alternatives that were considered, the final decision and the rationale for the decision.

E. Other details of the design:

- Give any other details the reader may want to know that have not yet been mentioned.

When writing the document

- Avoid documenting information that would be *readily obvious* to a skilled programmer or designer.
- Avoid writing details in a design document that would be better placed as *comments* in the code.
- Avoid writing details that can be *extracted automatically* from the code, such as the list of public methods.

Difficulties and Risks in Design

- Like modelling, design is a skill that requires considerable experience
  - *Individual software engineers should not attempt the design of large systems*
  - *Aspiring software architects should actively study designs of other systems*
- Poor designs can lead to expensive maintenance
  - *Ensure you follow the principles discussed in this chapter*

Difficulties and Risks in Design

- It requires constant effort to ensure a software system's design remains good throughout its life
  - *Make the original design as flexible as possible so as to anticipate changes and extensions.*
  - *Ensure that the design documentation is usable and at the correct level of detail*
  - *Ensure that change is carefully managed*

STRUCTURED PROGRAMMING

Structured Programming

- Programming style to improve the quality, clarity and development time by make use of block structures.
- Better way to program as it involves systematic organization of programs.
- Structured Programming started in 70s,primarly against control constructs like gotos
- The goal was to simplify the programming structure
- Is now well established and followed.

Structured Programming.....

```
Main()
{
..............
.................
  goto;
.................
..............

goto;

.................
}
```

Structured programming

- Structured programming is concerned with the structures used in computer program.
- Generally structures of computer program comprise decisions, sequences and loops

  - Sequence
    - *implements* processing steps that are essential in the specification of any algorithm.
  - Condition (decision)
    - selected processing based on some logical occurrence
  - Repetition(loops)
    - allows for looping

Structured programming....

- In structured programming, we sub-divide the whole program into small modules so that the program becomes easy to understand.
- Linearize control flow through a computer program so that the execution sequence follows the sequence in which the code is written.
- This linear flow of control can be managed by restricting the set of allowed applications construct to a single entry, single exit formats.

Structured programming.....

- Primarily structured programming focus on reducing the following statements from our program
  - GOTO statements
  - Break or continue
  - Multiple exit points to a function ,procedure or subroutine.(ex. Multiple return statements)
  - Multiple entry points to a function ,procedure or subroutine
- The key point of structured programming : **single entry and single exit points.**

Why we use Structured Programming?

- The use of the structured constructs reduces program complexity
- Enhances readability, testability, and maintainability.
- The intent of structured programming is to assist the designer in defining algorithms that are less complex and therefore easier to read, test, and maintain.
- If a program consists of thousands of instructions and an error occurs then it is complicated to find that error in the whole program.
- but in structured programming, we can easily detect the error and then go to that location and correct it.
- This saves a lot of time.

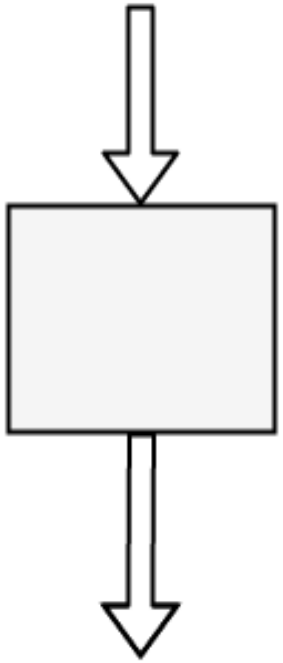
Rules in structured programming:

- **Structured Rule One: Code Block**
  - If the entry conditions are correct, but the exit conditions are wrong, the error must be in the block.

  #This is not true if the execution is allowed to jump into a block. The error might be anywhere in the program. Debugging under these circumstances is much harder.

  - In flow-charting condition, a box with a single entry point and single exit point are structured.
  - Structured programming is a method of making it evident that the program is correct.

Rules in structured programming....

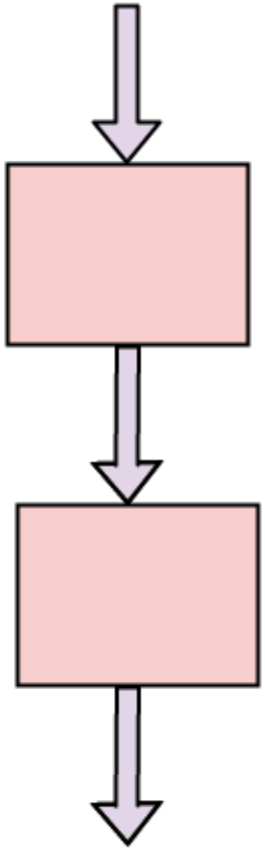


Rule1: Code block is structured

Rules in structured programming....

- **Structure Rule Two: Sequence**
- A sequence of blocks is correct if the exit conditions of each block match the entry conditions of the following block.
- Execution enters each block at the block's entry point and leaves through the block's exit point.
- The whole series can be regarded as a single block, with an entry point and an exit point.

Rules in structured programming.....

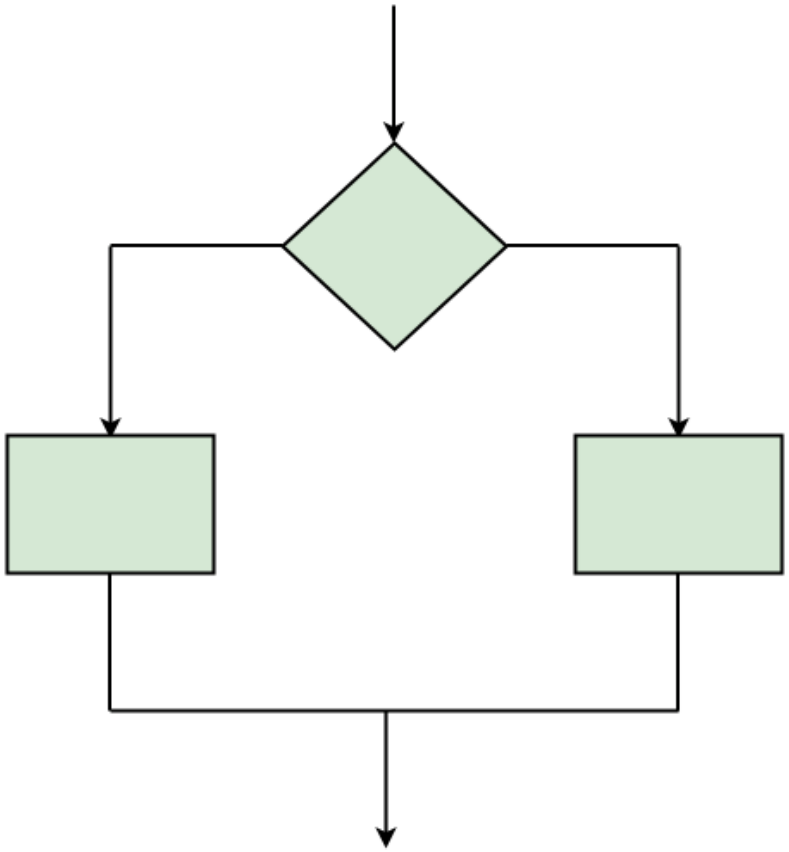


Rule2: A sequence of code blocks is structured

Rules in structured programming....

- **Structured Rule Three: Alternation**
- If-then-else is frequently called alternation (because there are alternative options).
- In structured programming, each choice is a code block.
- If alternation is organized as in the flowchart at right, then there is one entry point (at the top) and one exit point (at the bottom).
- The structure should be coded so that if the entry conditions are fulfilled, then the exit conditions are satisfied (just like a code block).

Rules in structured programming....

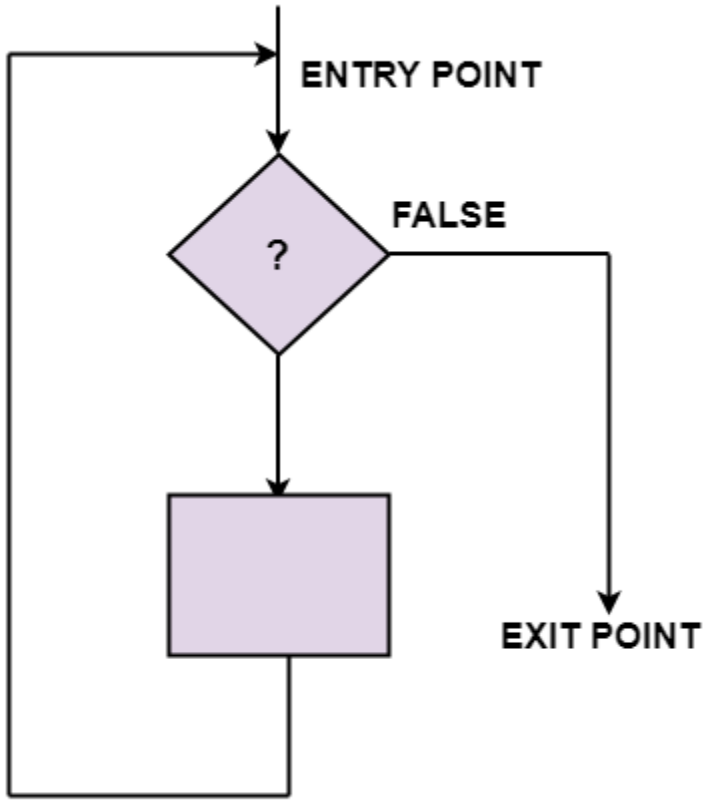


Rule 3: An alternation of code blocks is structured

Rules in structured programming.....

- **Structured Rule 4: Iteration**
- Iteration (while-loop) is organized as at right.
- It also has one entry point and one exit point.
- The entry point has conditions that must be satisfied, and the exit point has requirements that will be fulfilled.
- There are no jumps into the form from external points of the code.

Rules in structured programming....



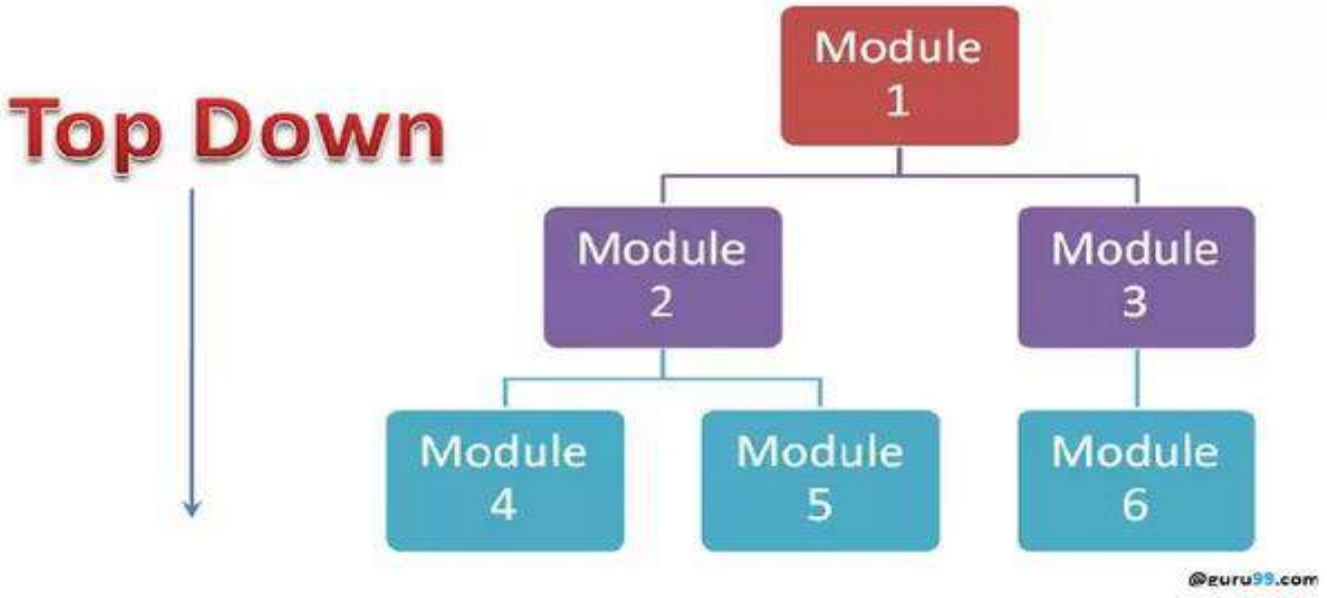
Rule 4: Iteration of code blocks is structured

Structured Programming

- Structured programming involves 3 phases
  - Top down analysis
  - Modular programming
  - Structured code

Top down analysis

- **Top down analysis** is a problem solving mechanism whereby a given problem is successively broken **down** into smaller and smaller sub-problems or operations until a set of easily solvable (by computer) sub-problems is arrived at.



Modular programming

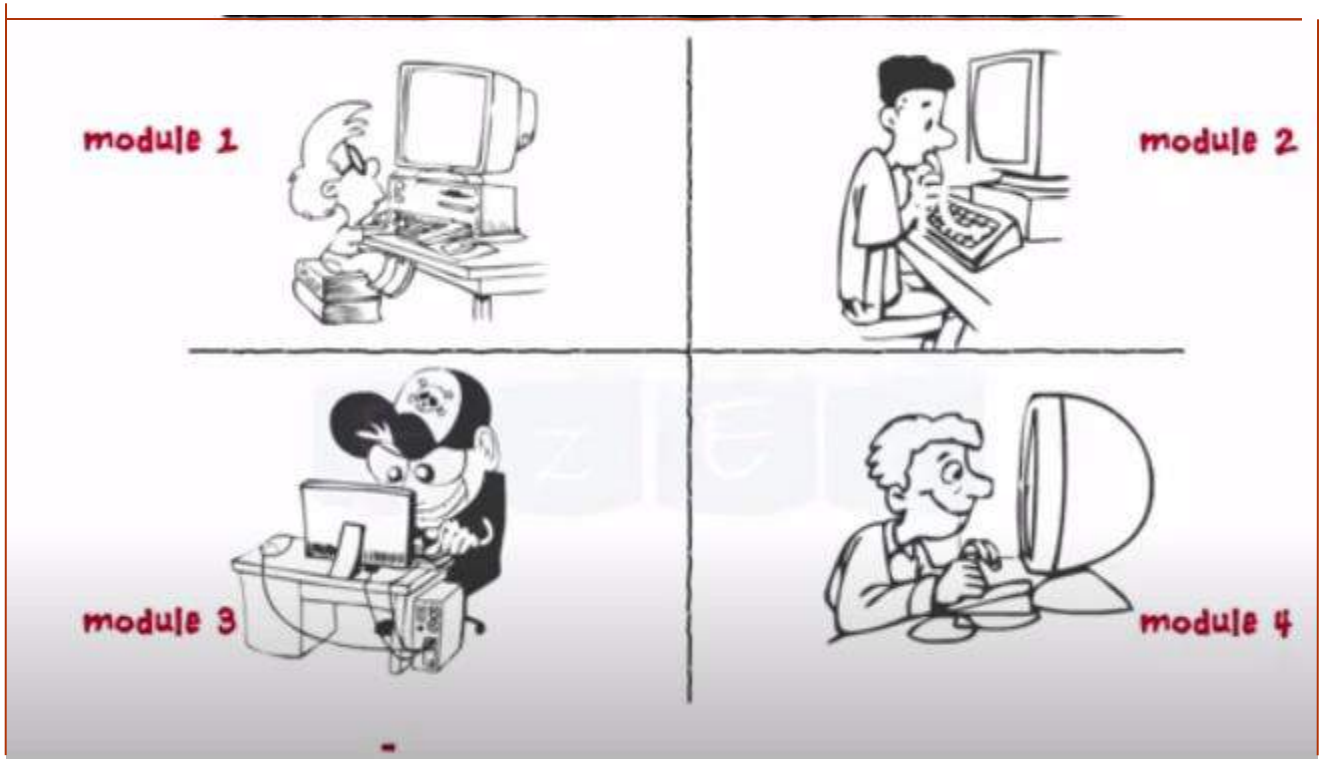
- Software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality
- Modular programming will be helpful for large programs which are difficult to debug.
- Leads to reuse the code which can be used in other programs
- Debugging and finding errors is easy

Modular programming

```
main()
 {
  ……….
  ……….
}
```

```
Main()
{
   ……….
   ……….
}
Module 1                                  independent
{
   ……….
}
Module 2                                  independent
{
   ……….
}
```

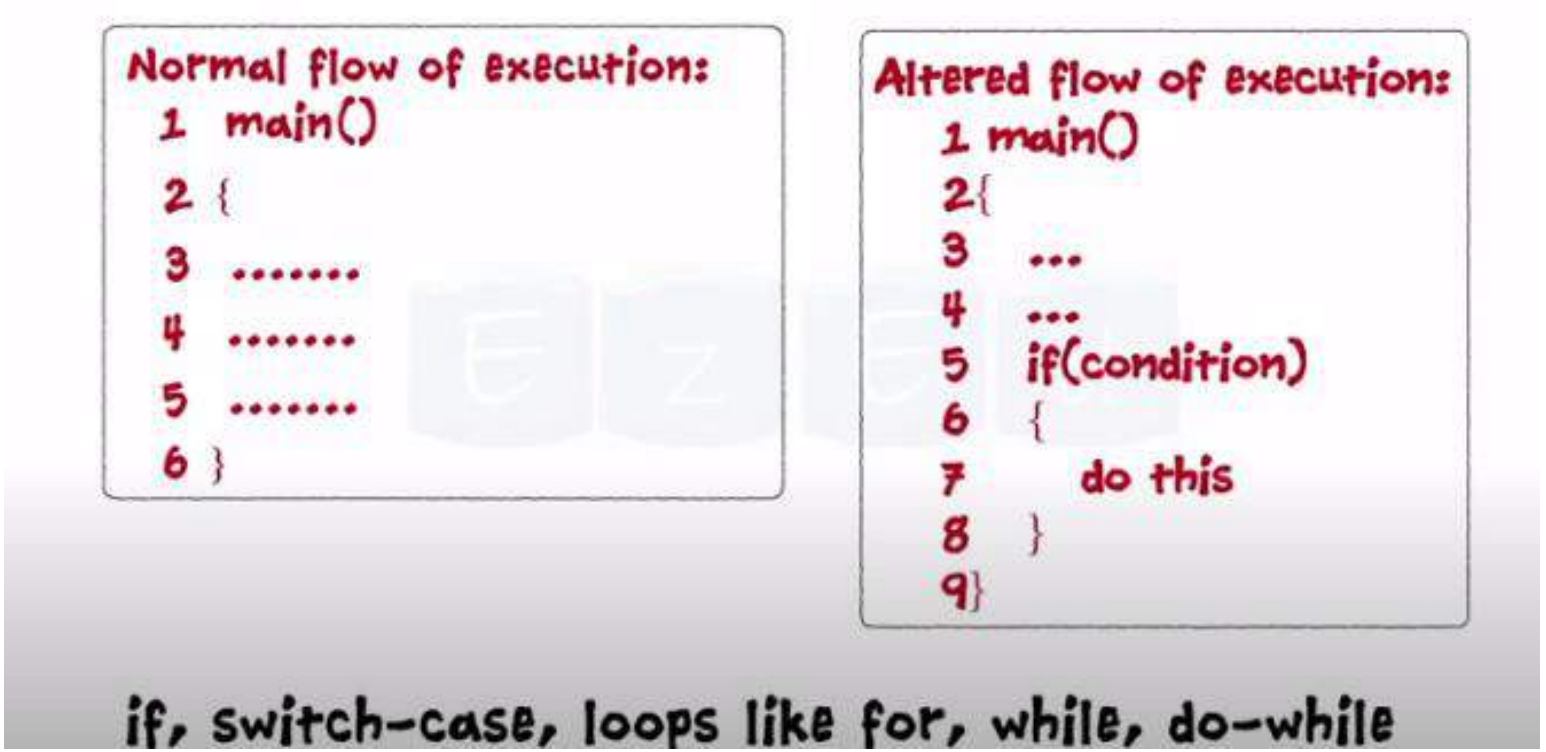
Advantage of modular programming



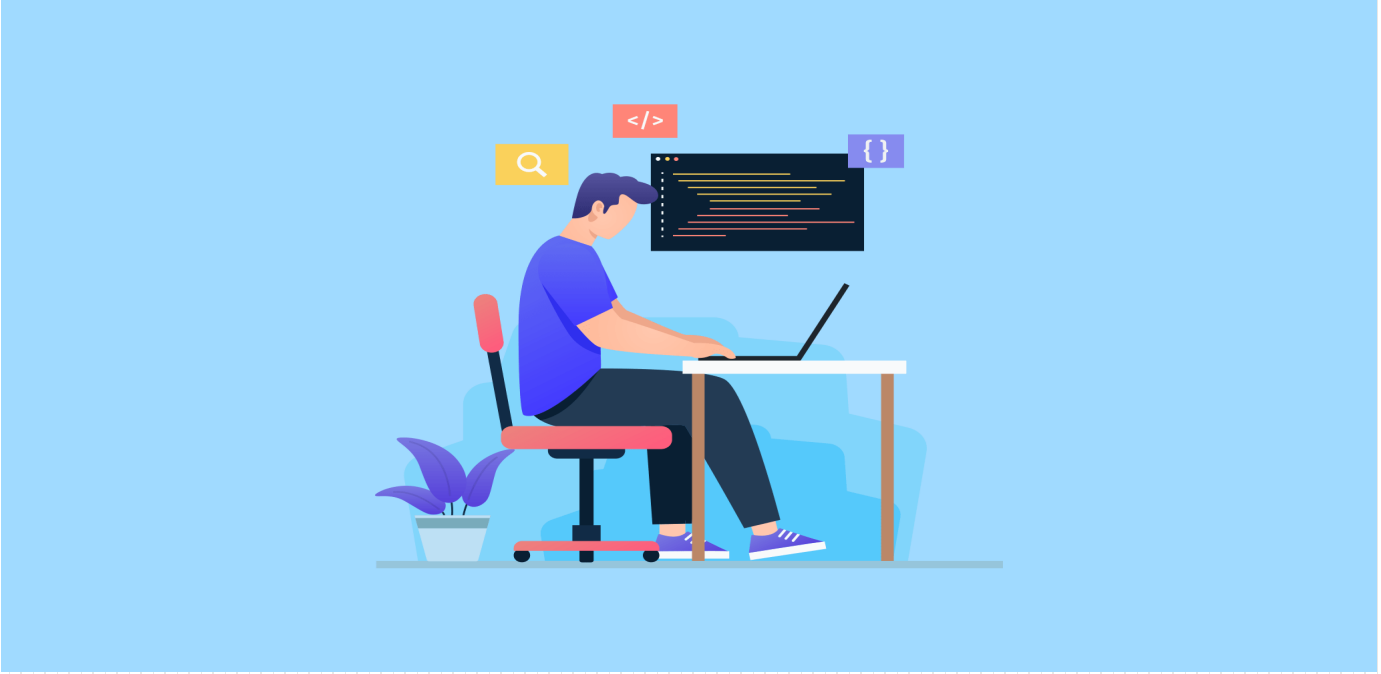
Structured Code

use of the structured control flow constructs of selection (if/then/else) and remove goto

Structured code



CODING STANDARDS



CODING STANDARDS

- Standardization has a positive impact on any business.
- There are certain coding standards that are needed for successful software development.
- A coding standard makes sure that all the developers working on the project are following certain specified guidelines.
- The code can be easily understood and proper consistency is maintained.

Coding Standards....

- General coding standards refers to how the developer writes code, so here we will discuss some essential standards regardless of the programming language being used.

CODING STANDARDS.....

- **Indentation:**
  - Proper and consistent indentation is essential in producing easy to read and maintainable programs.
    Indentation should be used to:
    - Emphasize the body of a control structure such as a loop or a select statement.
    - Emphasize the body of a conditional statement
    - Emphasize a new scope block
- **Inline comments:**
  - Inline comments analyze the functioning of the subroutine, or key aspects of the algorithm shall be frequently used.
- **Rules for limiting the use of global:**
  - These rules file what types of data can be declared global and what cannot.

CODING STANDARDS.....

- **Structured Programming:**
  - Structured (or Modular) Programming methods shall be used.
    - "GOTO" statements shall not be used as they lead to "spaghetti" code, which is hard to read and maintain.
- **Naming conventions for global variables, local variables, and constant identifiers:**
  - A possible naming convention can be that global variable names always begin with a capital letter,
  - local variable names are made of small letters, and constant names are always capital letters.
- **Error return conventions and exception handling system:**
  - Different functions in a program report the way error conditions are handled should be standard within an organization.
  - Error handling is an essential aspect of computer programming. This does not only include adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

CODING STANDARDS.....

- **Line Length:**
  - It is considered a good practice to keep the length of source code lines at or below 80 characters. Lines longer than this may not be visible properly on some terminals and tools. Some printers will truncate lines longer than 80 columns.
- **Spacing:**
  - The appropriate use of spaces within a line of code can improve readability.

Example:

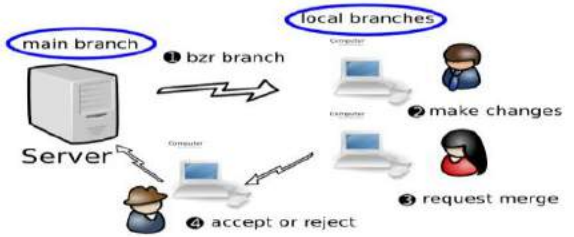
Bad:
```
cost=price+(price*sales_tax)
fprintf(stdout ,"The total cost is %5.2f\n",cost);
```

Better:
```
cost = price + ( price * sales_tax )
fprintf (stdout,"The total cost is %5.2f\n",cost);
```

VERSION CONTROL SYSTEMS

Version Control Systems(VCS)

- A *version control system* allows programmers to keep track of every revision of all source code files
- Version control is important to keep track of changes and keep every team member working off the latest version.
  - The main element of the version control system is the *repository*, a database or directory which contains each of the files contained in the system.
  - A programmer can pick a point at any time in the history of the project and see exactly what those files looked like at the time.
  - Latest version of any file can be retrieved from the repository.
  - Changing a file will not unexpectedly overwrite any previous changes to that file; any change can be rolled back.



Version Control Systems(VCS)

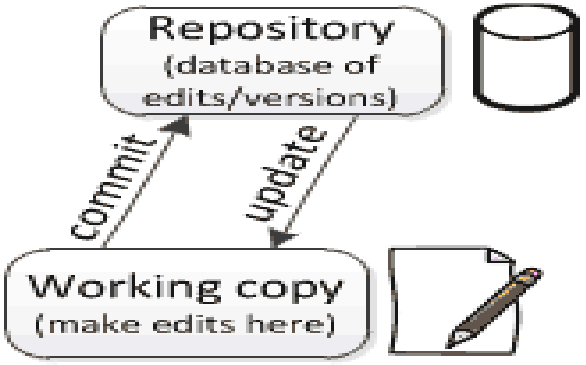
- Version Control Systems (VCS) have seen great improvements over the past few decades and some are better than others.
- VCS are sometimes known as SCM (Source Code Management) tools or RCS (Revision Control System).
- Some popular version control systems are *Git* (distributed), *Mercurial* (distributed), and *Subversion* (centralized).

Purpose of Version Control

- Version control enables multiple people to simultaneously work on a single project.
- integrates work done simultaneously by different team members
- Version control gives access to historical versions of your project.

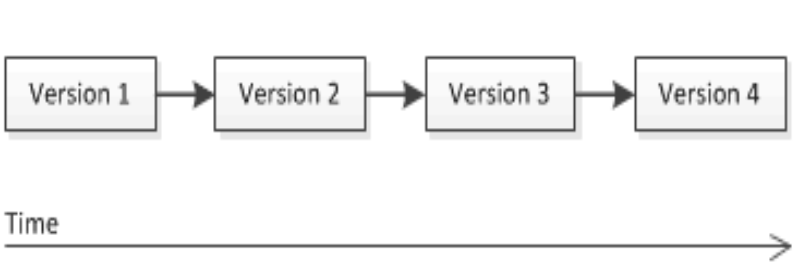
Repositories and working copies

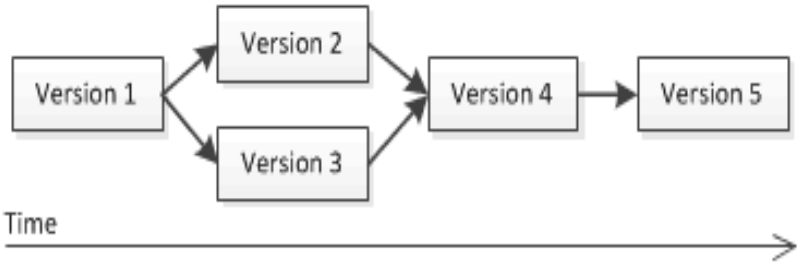
- Version control uses a *repository* (a database of changes) and a *working copy* where you do your work.
  - A repository is a database of all the edits to, and/or historical versions (snapshots) of, your project.
  - Your *working copy* (sometimes called a *checkout*) is your personal copy of all the files in the project.



Branching in version control

- In the simplest case, the database contains a linear history: *each change is made after the previous one.*
- Another possibility is that different users made edits simultaneously (this is sometimes called "*branching*").
  - version history splits and then merges again.





Version Control Systems(VCS)

- There are two common models for version control systems
  - In a copy-modify-merge system, multiple people can work on a single file at a time.
    - When a programmer wants to update the repository with his changes, he retrieves all changes which have occurred to the checked out files and reconciles any of them which conflict with changes he made before updating the repository.
  - In a lock-modify-unlock system, only one person can work on any file at a time.
    - A programmer must check a file out of the repository before it can be modified. The system prevents anyone else from modifying any file until it is checked back in.
    - On large projects, the team can run into delays because one programmer is often stuck waiting for a file to be available.

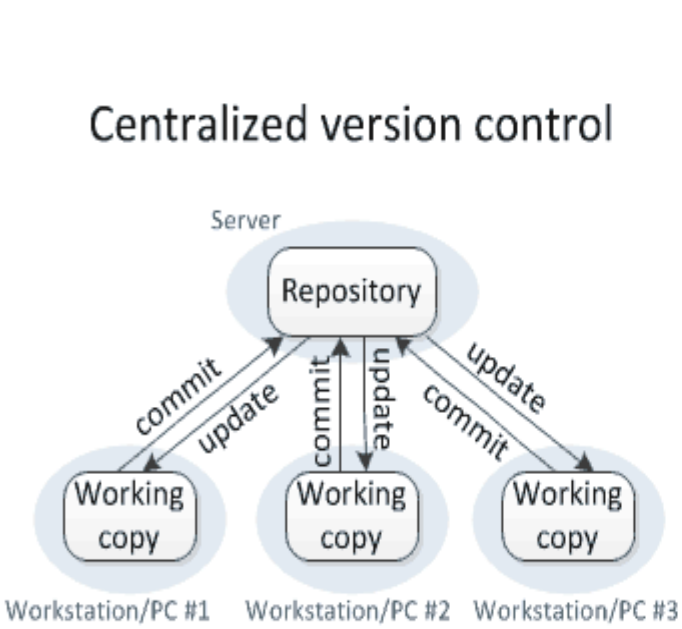
INDUSTRY JARGON

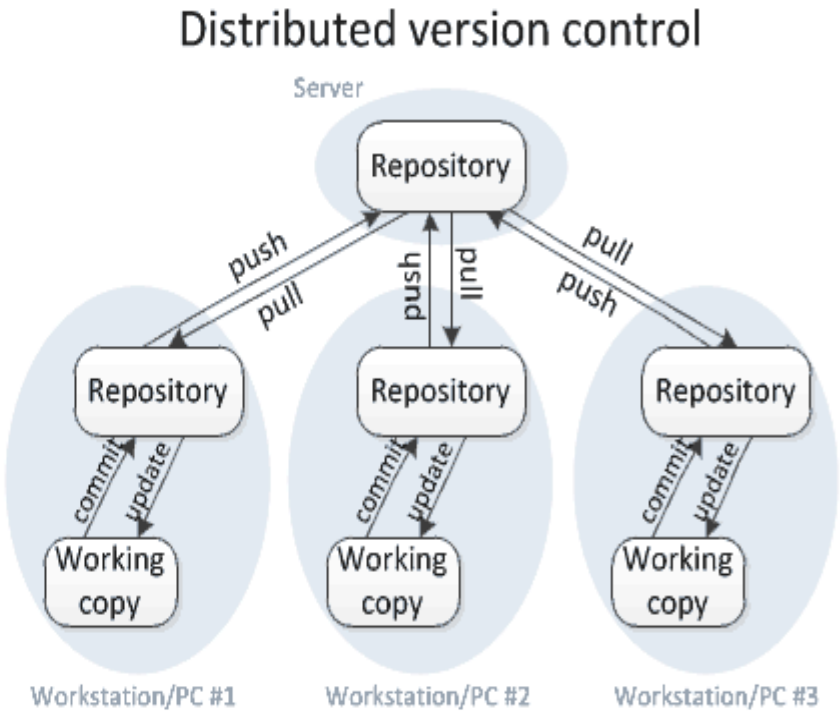
- **Repository (repo):** Database where files are stored
- **Server:** The computer storing the repository
- **Client:** The computer connecting to the repository
- **Working Copy:** Your local directory of files
- **Trunk / Main:** Master location for code in the repository
- **Head:** The latest revision in the repository

BASIC ACTIONS

- **Add:** Place a file under version control
- **Check in:** Send local changes to the repo
- **Check out:** Download from a repo to your working copy
- **Ignore:** Allows files to exist in your working copy but not in the repo
- **Revert:** Throw away your working copy and restore last revision
- **Update / Sync:** Update your working copy to the latest revision

Centralized Vs Distributed VCS





Centralized Version Control System

- server and a client
- server is the master repository which contains all of the versions of the code
- To work on any project, firstly user or client needs to get the code from the master repository(take an *update* from the master repository)
- After making the changes ,need to commit those changes straight forward into the master repository.
- There will be just one repository and that will contain all the history or version of the code and different branches of the code.
- *Platforms*: **SVN**,ClearCase etc.
- *Software*:Tortoise SVN,SmartSVN

Distributed Version Control System

- every single developer or client has their own server and they will have a copy of the entire history or version of the code and all of its branches in their local server or machine
- every client or user can work locally and disconnected which is more convenient than centralized source control

Distributed Version Control System….

- when you start working on a project,
  - you clone the code from the master repository in your own hard drive,
  - then you get the code from your own repository to make changes
  - after doing changes, you commit your changes to your local repository
  - At this point your local repository will have '*change sets*' but it is still disconnected with master repository (master repository will have different '***sets of changes***' from each and every individual developer's repository)

Distributed Version Control System…

- To communicate with it, you issue a request to the master repository and push your local repository code to the master repository.
- Getting the new change from a repository is called "**pulling**" and merging your local repository 'set of changes' is called "**pushing**".
- *Platforms*: **GIT**,mercurail,Bazaar
- *Softwares*:GITHub,GITLab,bBtbucket etc.

TOOLS & APPS - MAC

Git

- gityapp.com
- git-tower.com
- sourcetreeapp.com
- gitboxapp.com
- GitHub Desktop

Subversion

- versionsapp.com
- kaleidoscopeapp.com
- Cornerstone
- Beyond Compare
- Xcode (FileMerge)

TOOLS & APPS - WIN

Git

- TortoiseGit
- sourcetreeapp.com
- GitHub Desktop
- IDE integration

Subversion

- TortoiseSVN
- TortoiseIDiff (image diff)
- IDE integration
- WinMerge

Subversion(SVN)

- Apache Subversion, abbreviated as SVN aims at to be a best-matched successor to the widely used **CVS**
  - Has a benefit of good GUI tools like TortoiseSVN.
  - Supports empty directories.
  - Have better windows support as compared to Git.
  - Easy to set up and administer.
  - Integrates well with Windows, leading IDE and Agile tools.
  - Free and opensource
  - Official website: https://subversion.apache.org/

Subversion(SVN)

- **SVN checkout**
  - Downloads repository contents to a local folder
- **Update**
  - Merges new changes from repository to the local working copy. Updates before every commit are important .
- **Commit**
  - Merges changes from local copy to SVN repository.
- **Resolved**
  - Tells SVN that the conflicted file is now OK.
- **Add**
  - Schedules new file to be saved to repository with next commit.
- **Delete**
  - Schedules file to be deleted from repository with next commit. Accidental deletes can be recovered.
- **Revert**
  - Undoes all changes to the local copy since last commit. Throw away the changes and start over.
- **Log**
  - Shows log of changes sorted by date

GIT

- Git is one of the best version control tools that is available in the present market.
  - Super-fast and efficient performance.
  - Cross-platform
  - Code changes can be very easily and clearly tracked.
  - Easily maintainable and robust.
  - Offers an amazing command line utility known as git bash.
  - Also offers GIT GUI where you can very quickly re-scan, state change, sign off, commit & push the code quickly with just a few clicks.
  - **Open Source and free**
- Official website : https://git-scm.com/

Software Testing

Introduction

- What is it?
- Why is it important?
- What is the work product?

A Test case

TEST CASE ID	TEST SCENARIO	TEST CASE	PRE-CONDITION	TEST STEPS	TEST DATA	EXPECTED RESULT	POST CONDITION	ACTUAL RESULT	STATUS (PASS/ FAIL)
TC_LOGIN_001	Verify the login of Gmail	Enter valid User Name and valid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Valid User Name>	Successful login	Gmail inbox is shown		
				2. Enter Password	<Valid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter valid User Name and invalid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Valid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Invalid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter invalid User Name and valid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Invalid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Valid Password>				
				3. Click "Login" button					
TC_LOGIN_001	Verify the login of Gmail	Enter invalid User Name and invalid Password	1. Need a valid Gmail Account to do login	1. Enter User Name	<Invalid User Name>	A message "The email and password you entered don't match" is shown			
				2. Enter Password	<Invalid Password>				
				3. Click "Login" button					

Test Characteristics

- *A good test has a high probability of finding an error.*
- *A good test is not redundant*
- *A good test should be "best of breed"*
- *A good test should be neither too simple nor too complex*

"It does not guarantee absence of errors"

Testing Principles

- All tests should be *traceable to customer/user requirements*
- Tests should be *planned long before* testing begins
- The *Pareto principle* applies to software testing
- Testing *should begin "in the small" and progress toward testing "in the large"*
- *Exhaustive testing* is not possible
- To be most effective, testing should be conducted by an *independent third party*

What criteria to use?

- Should the testing be done based on externally observable behavior ?

Or

- Should the code be seen to design testcases?

External Vs Internal View

-Black box testing

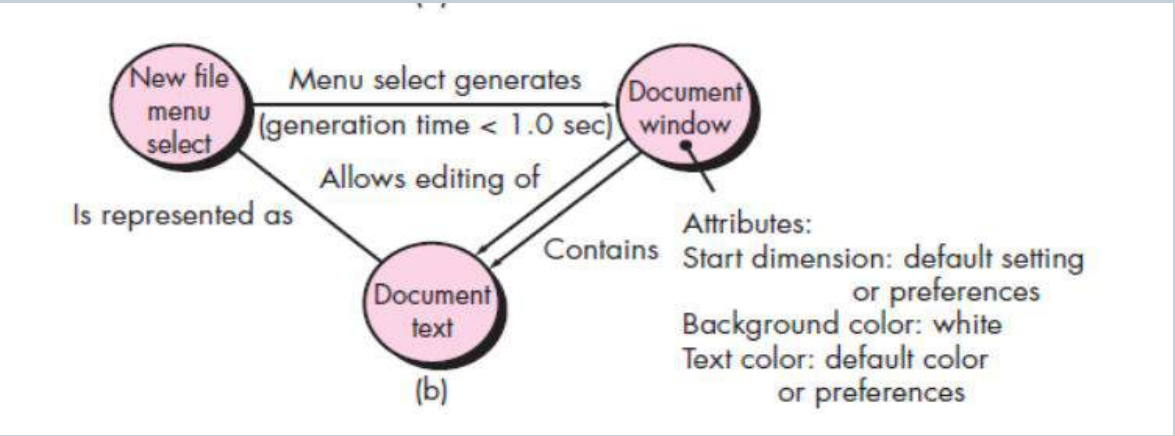
- White box testing

Black box testing

- Also called as *functional testing* or *behavioral testing*,
- Tests in the external point of view
- Specifications are used to generate testcases
- Tests for absence of features
- No programming knowledge is required

Black box testing techniques

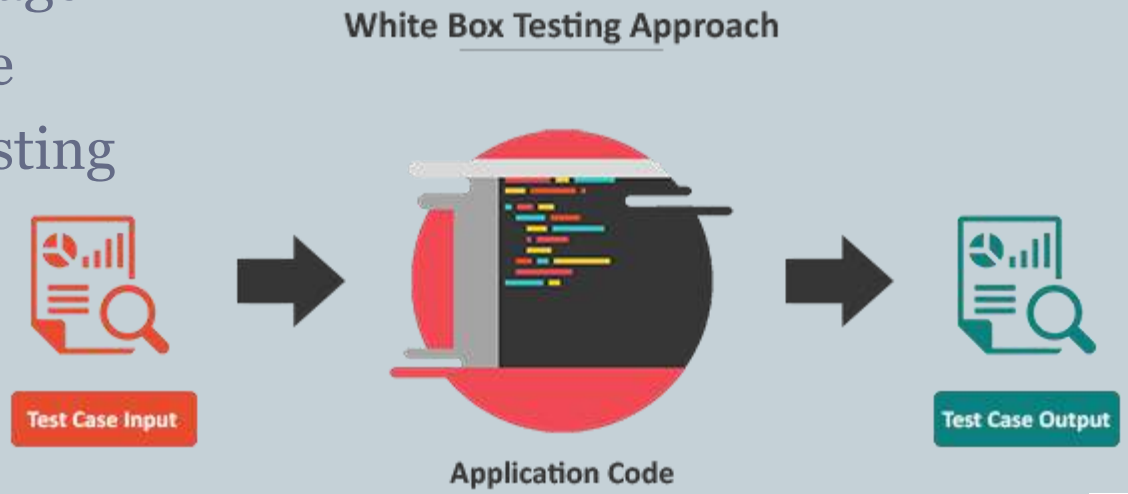
- Graph-Based Testing Methods



- Equivalence Partitioning
- Boundary Value Analysis

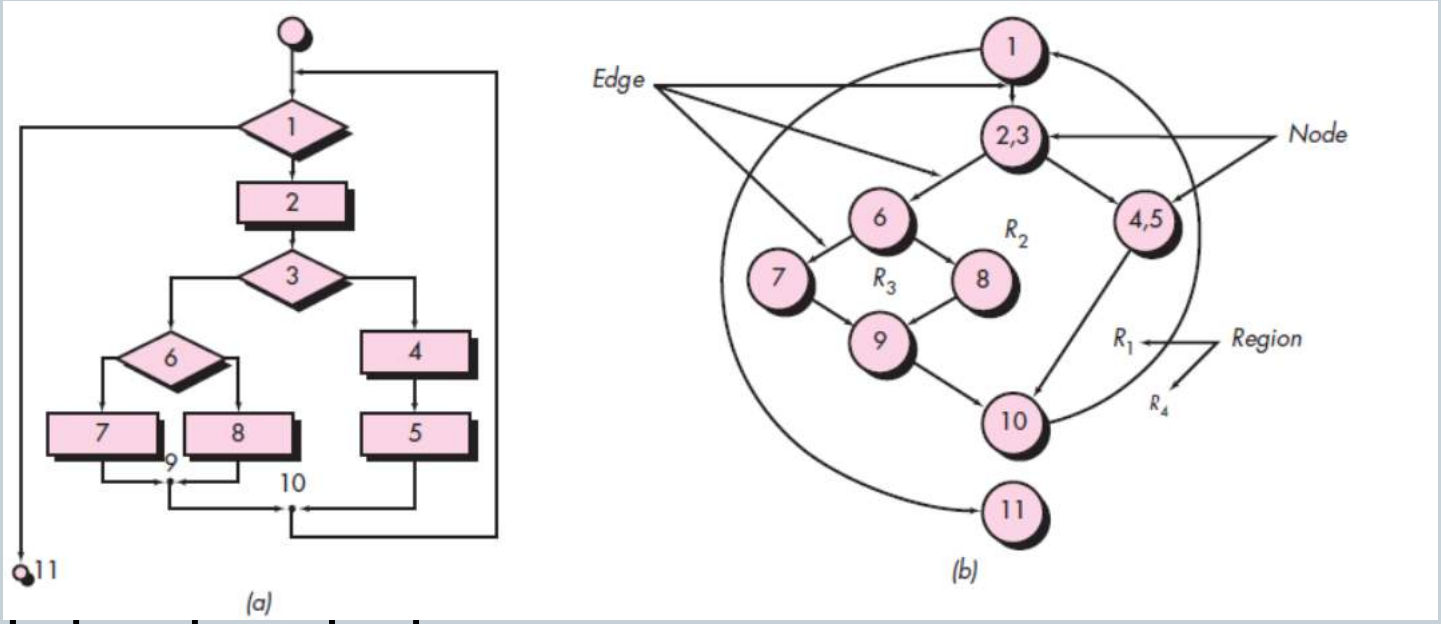
White box testing

- Also called as *glass box testing* or *structural testing*
- Tests the internal point of view or implementation
- Cannot detect absence of features
- Coverage measures are used
  - Statement coverage
  - Branch Coverage
  - Path oriented testing



White box testing techniques

- Basis Path Testing



- Control structure testing
  - Condition testing
  - Loop testing

Testing Strategies for Software Process

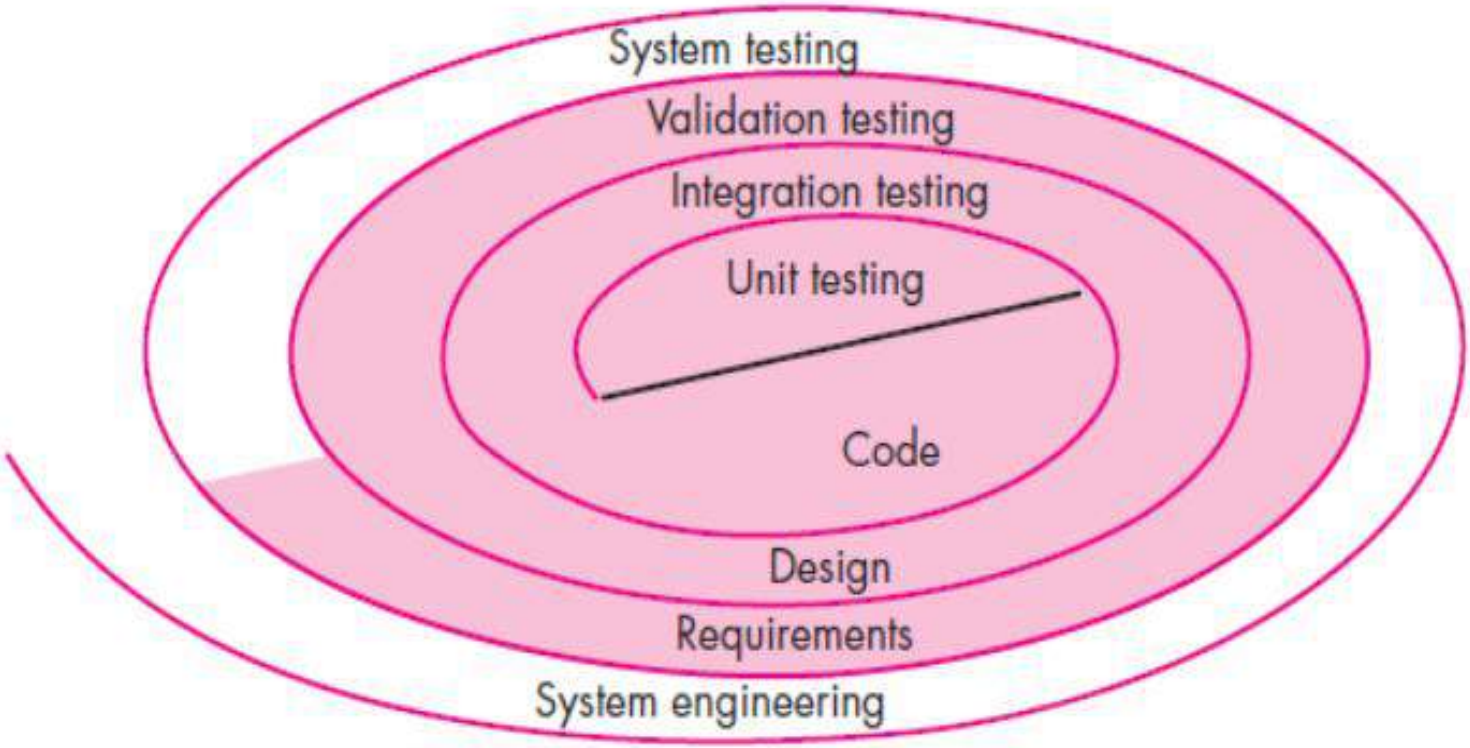
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.

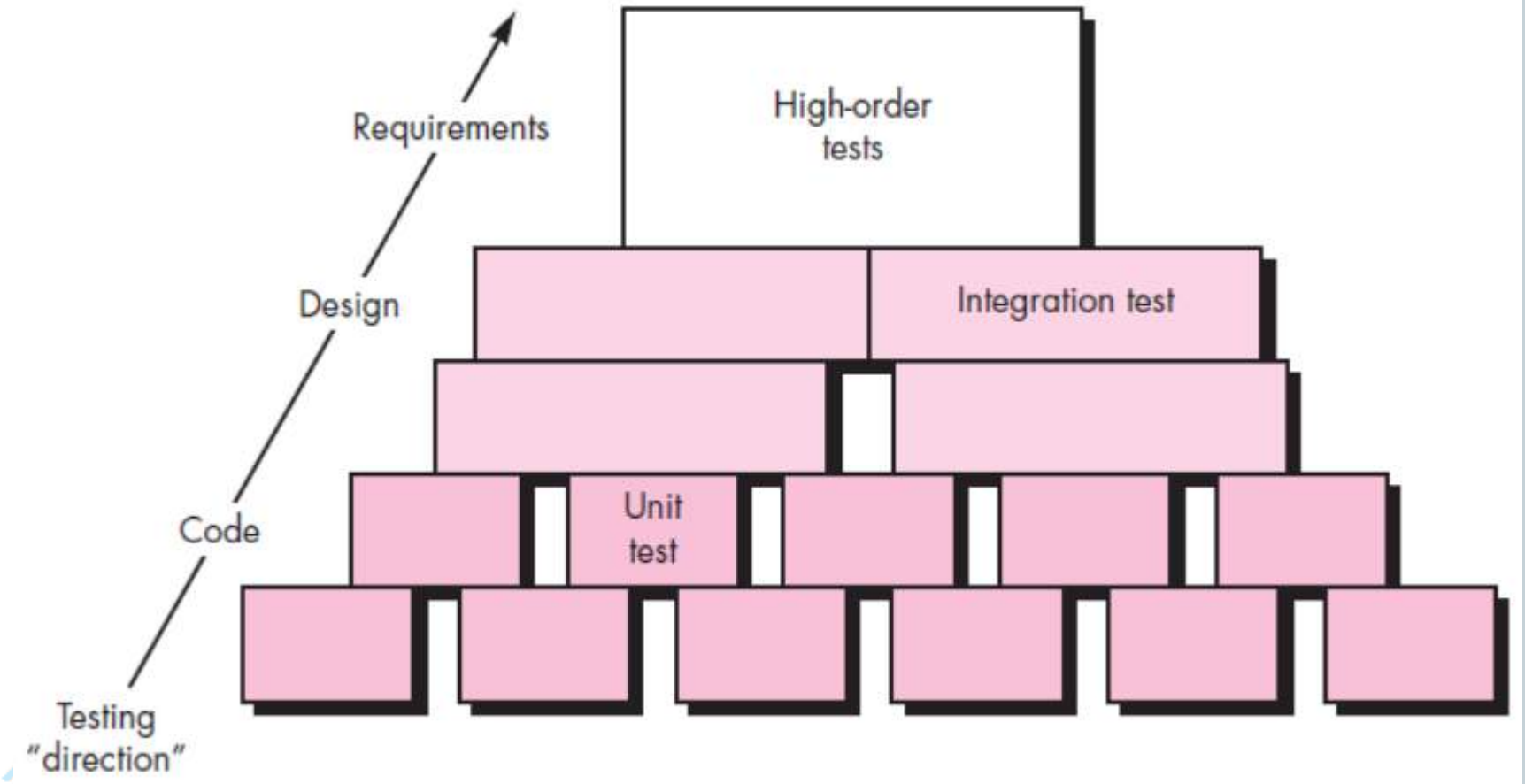
Verification and Validation (V&V)

Verification: “Are we building the product right?”

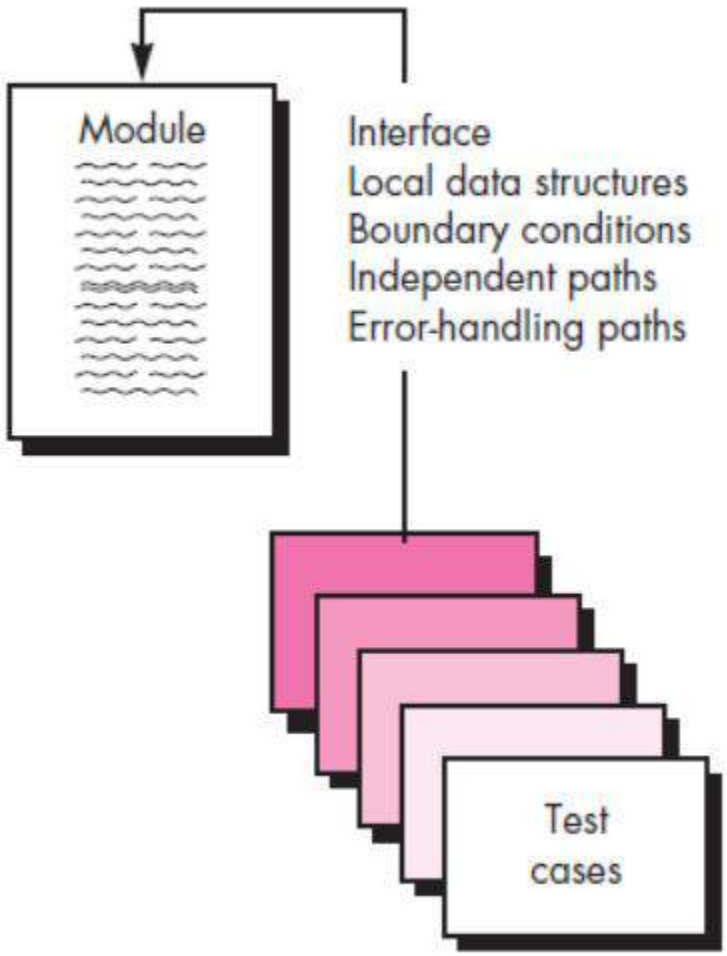
Validation: “Are we building the right product?”

Software Testing Strategy—The Big Picture

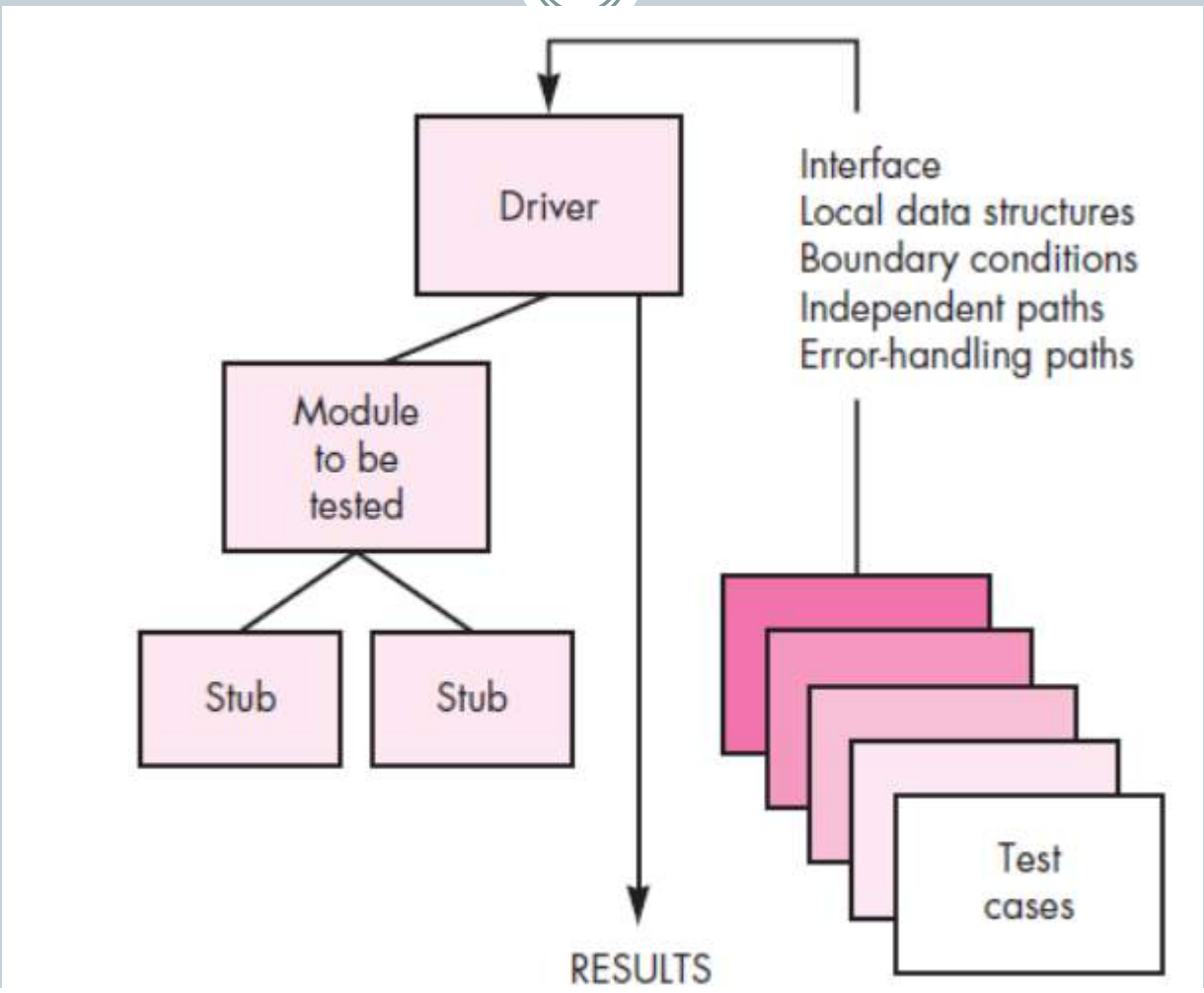



Requirements
Design
Code
Testing
"direction"
High-order
tests
Integration test
Unit
test

Unit Testing



Unit Test Procedure



Integration Testing

Once all modules have been unit tested:

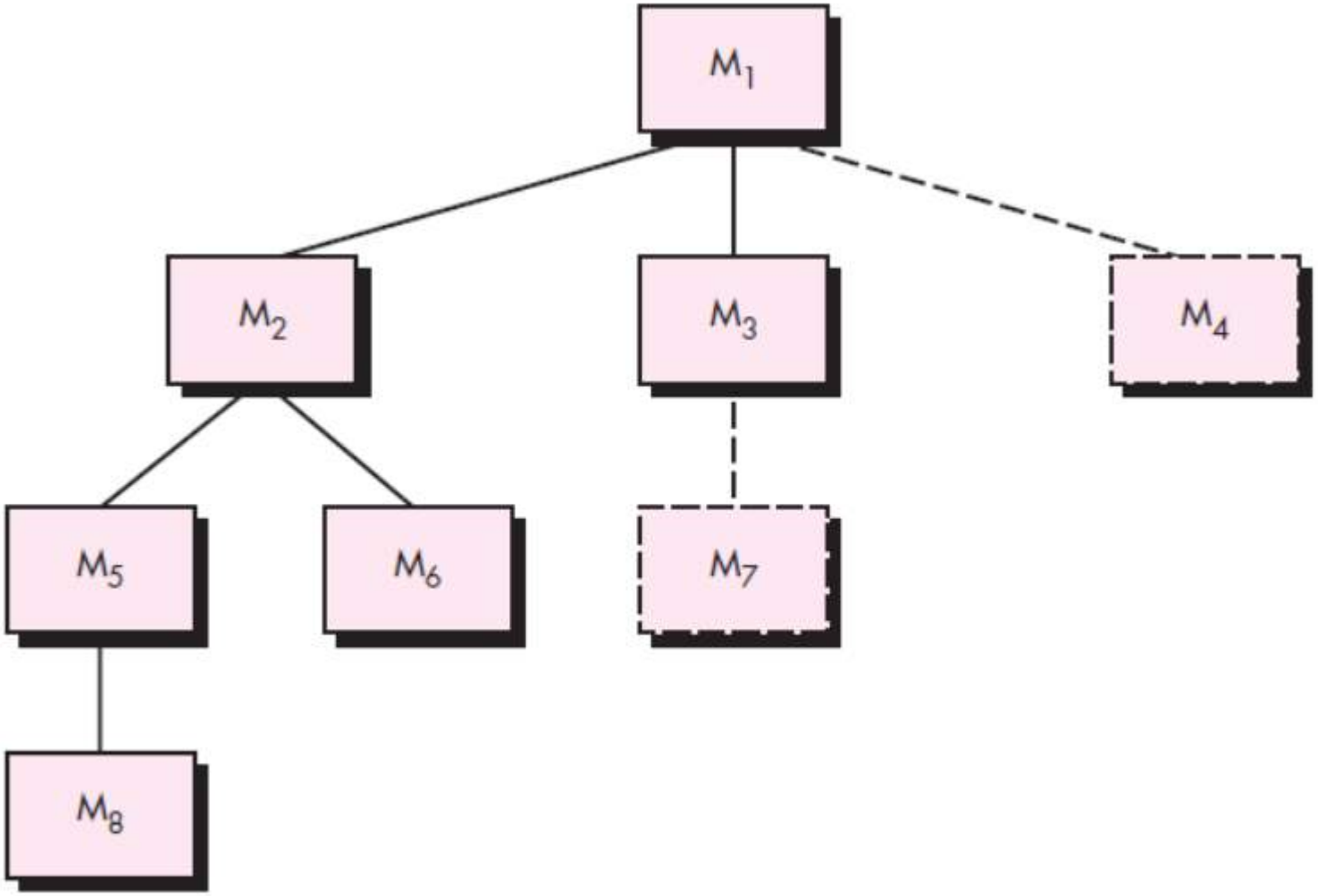
"If they all work individually, why do you doubt that they'll work when we put them together?"

The problem, of course, is

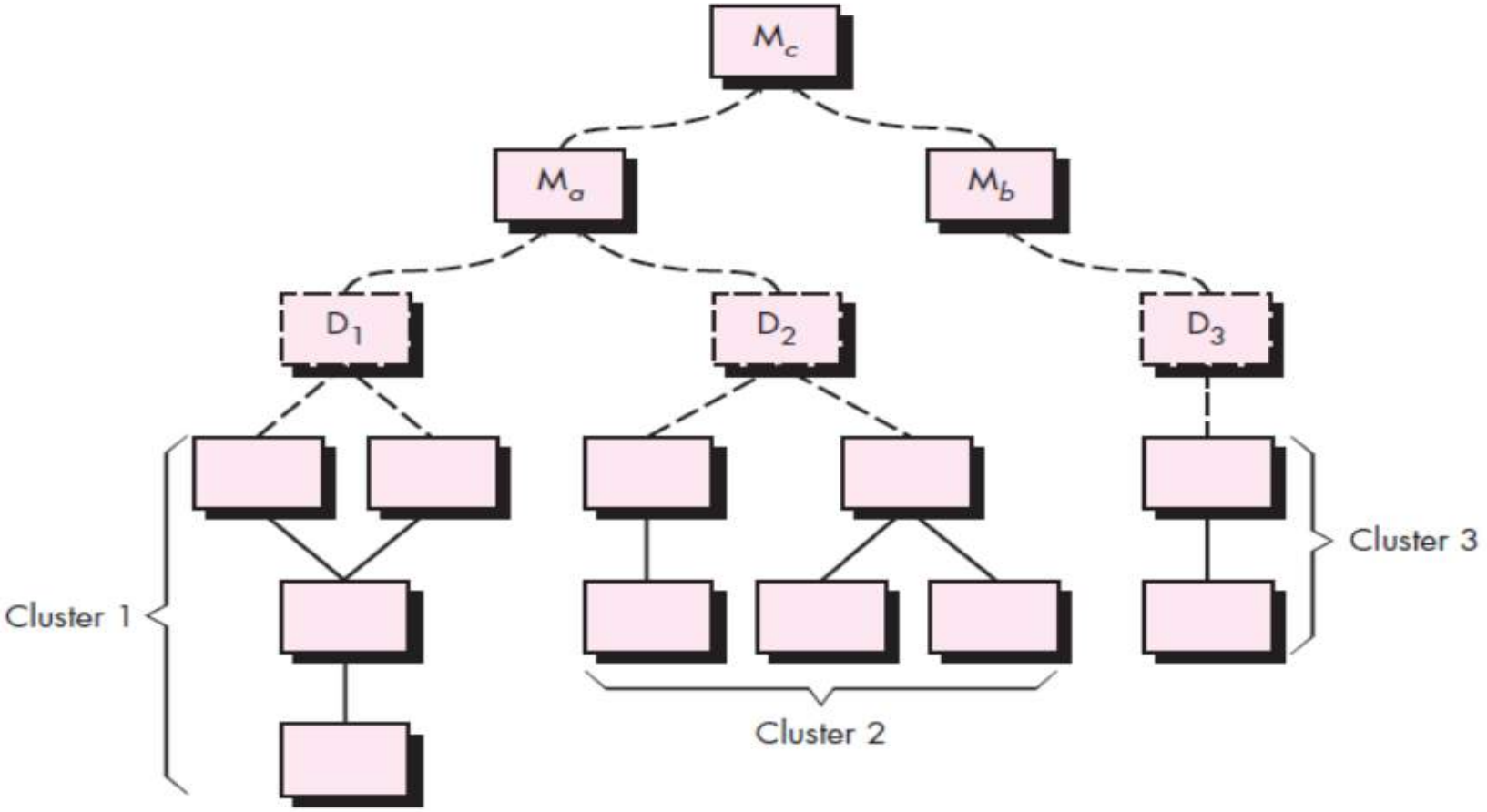
"putting them together"—interfacing.

- Incremental Approach is desirable
  - Top-Down Integration
  - Bottom-Up Integration

Top-down Integration



Bottom-up integration



Regression testing

- *Re-execution of some subset of tests that have already been* conducted to ensure that changes have not propagated unintended side effects.
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes do not introduce unintended behavior or additional errors.

Validation Testing

- Testing focuses on user-visible actions and user-recognizable output from the system.
- Validation **succeeds when software functions in a manner that can be reasonably expected by the customer**
- If Software Requirements Specification has been developed, it forms the basis for a validation-testing approach

Acceptance Test

- Alpha Testing
  - Conducted at the developer's site by a representative group of end users in a controlled environment.
  - The software is used in a natural setting with the developer and records errors and usage problems.
- Beta Testing
  - Conducted at one or more end-user sites
  - The developer generally is not present.
  - It is a "live" application of the software in an environment that cannot be controlled by the developer.
  - The customer records all problems.

System Testing

- Recovery Testing
  - Systems must recover from faults and resume processing with little or no downtime.
  - It is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security Testing
  - Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress Testing
  - Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
  - It test the run-time performance of software within the context of an integrated system.

Debugging

- Debugging is not testing but often occurs as a consequence of testing
- When a test case uncovers an error, debugging is the process that results in the removal of the error
- Attempts to match symptom with cause, thereby leading to error correction
- The activity must track down the cause of an error
- Debugging is difficult

Code Inspection

- An inspection is an activity in which one or more people systematically examine source code or documentation, looking for defects.
- Both testing and inspection rely on different aspects of human intelligence
- Inspecting allows you to get rid of many defects quickly.

THANK YOU

Introduction to Metrics

Introduction

- **Measure:**

  It provides a quantitative indication of the extent, dimension, size and the capacity of a product.

- **Measurement:**

  It is defined as the act of evaluating a measure.

- **Metric:**

  It is a quantitative measure of the degree to which a system or its component possesses a given attribute.

A Quote on Measurement

- "When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science."

LORD WILLIAM KELVIN (1824 – 1907)

What are Software Metrics?

A **software metric** is a measure of **software** characteristics which are measurable or countable.

Software metrics are valuable for many reasons, including measuring **software** performance, planning work items, measuring productivity, and many other uses.

What are Metrics?

- Software metrics are quantitative measures
- They are a management tool
- They offer insight into the effectiveness of the software process and the projects.
- Basic quality and productivity data are collected
- These data are analyzed, compared against past averages, and assessed
- The goal is to determine whether quality and productivity improvements have occurred
- The data can also be used to pinpoint problem areas
- Remedies can then be developed and the software process can be improved

Uses of Measurement

- Can be applied to the software process with the intent of improving it on a continuous basis
- Can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control
- Can be used to help assess the quality of software work products and to assist in tactical decision making as a project proceeds

What to be measured?

- **Characteristics of the software product:**
  - Software size and complexity
  - Software reliability and quality
  - Functionalities
  - Performance
- **Characteristics of the software project:**
  - Number of software developer
  - Staffing pattern over the life cycle of software
  - Cost and schedule
  - Productivity
- **Characteristics of software process:**
  - Methods, tools, and techniques used for software development
  - Efficiency of detection of fault

Reasons to Measure

- To characterize in order to
  - Gain an understanding of processes, products, resources, and environments
  - Establish baselines for comparisons with future assessments
- To evaluate in order to
  - Determine status with respect to plans
- To improve in order to
  - Identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance

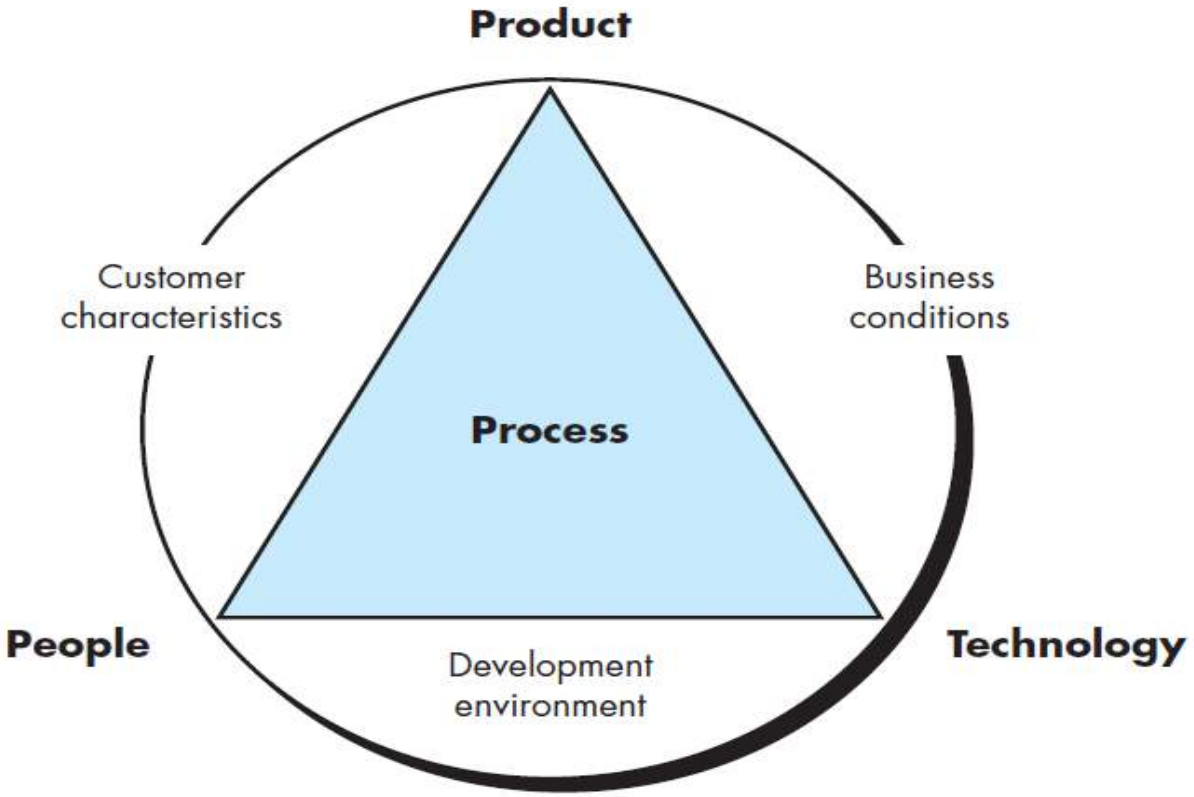
Role of Metrics in the Process Domain

- Process metrics are collected across all projects and over long periods of time
- They are used for making strategic decisions
- The intent is to provide a set of process indicators that lead to long-term software process improvement
- The only way to know how/where to improve any process is to
  - Measure specific attributes of the process
  - Develop a set of meaningful metrics based on these attributes
  - Use the metrics to provide indicators that will lead to a strategy for improvement

Metrics in the Process Domain.....

- We measure the effectiveness of a process by deriving a set of metrics based on outcomes of the process such as
  - Errors uncovered before release of the software
  - Defects delivered to and reported by the end users
  - Work products delivered
  - Human effort expended
  - Calendar time expended
  - Conformance to the schedule
  - Time and effort to complete each generic activity

Determinants for software quality an organizational effectiveness.



Etiquette of Process Metrics

- Use common sense and organizational sensitivity when interpreting metrics data
- Provide regular feedback to the individuals and teams who collect measures and metrics
- Don't use metrics to evaluate individuals
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them
- Never use metrics to threaten individuals or teams
- Metrics data that indicate a problem should not be considered "negative"
  - Such data are merely an indicator for process improvement
- Don't obsess on a single metric to the exclusion of other important metrics

Role of Metrics in the Project Domain

- Project metrics enable a software project manager to
  - Assess the status of an ongoing project
  - Track potential risks
  - Uncover problem areas before their status becomes critical
  - Adjust work flow or tasks
  - Evaluate the project team's ability to control quality of software work products
- Many of the same metrics are used in both the process and project domain
- Project metrics are used for making tactical decisions
  - They are used to adapt project workflow and technical activities

Use of Project Metrics

- The first application of project metrics occurs during estimation
  - Metrics from past projects are used as a basis for estimating time and effort
- As a project proceeds, the amount of time and effort expended are compared to original estimates
- As technical work commences, other project metrics become important
  - Production rates are measured (represented in terms of models created, review hours, function points, and delivered source lines of code)
  - Error uncovered during each generic framework activity (i.e, communication, planning, modeling, construction, deployment) are measured

Use of Project Metrics……

- Project metrics are used to
  - Minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks
  - Assess product quality on an ongoing basis and, when necessary, to modify the technical approach to improve quality
- In summary
  - As quality improves, defects are minimized
  - As defects go down, the amount of rework required during the project is also reduced
  - As rework goes down, the overall project cost is reduced

Categories of Software Measurement

- Two categories of software measurement
  - Direct measures of the
    - Software process (cost, effort, etc.)
    - Software product (lines of code produced, execution speed, defects reported over time, etc.)
  - Indirect measures of the
    - Software product (functionality, quality, complexity, efficiency, reliability, maintainability, etc.)

Size-oriented Metrics

- Derived by normalizing quality and/or productivity measures by considering the size of the software produced
- Thousand lines of code (KLOC) are often chosen as the normalization value
- Based on the LOC/KLOC count of software, many other metrics can be computed**:**
  - Errors/KLOC.
  - $/ KLOC.
  - Defects/KLOC.
  - Pages of documentation/KLOC.
- You can also compute other things like,
  - Errors/PM.
  - Productivity = KLOC/PM (effort is measured in person-months).
  - $/ Page of documentation.

Size-oriented Metrics

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

Size-oriented Metrics.....

- Size-oriented metrics are not universally accepted as the best way to measure the software process
- Opponents argue that KLOC measurements
  - Are dependent on the programming language
  - Penalize well-designed but short programs
  - Require a level of detail that may be difficult to achieve

Function-Based Metrics

- In 1977, A. J. Albrecht of IBM developed a method of software metrics based on the functionality of the software delivered by an application as a normalization value.
- He called it the **Function Points (FPs).**
- They are derived using an empirical relationship based on direct measures of software's information domain and assessments of software complexity.
- FPs try to quantify the functionality of the system, i.e., what the system performs.
- This is taken as the method of measurement as FPs cannot be measured directly.
- **FP is not a single characteristic but is a combination of several software features/characteristics.**

Function-Based Metrics

- The effort required to develop the project depends on what the software does.
- FP is programming language independent.
- FP method is used for data processing systems, business systems like information systems.

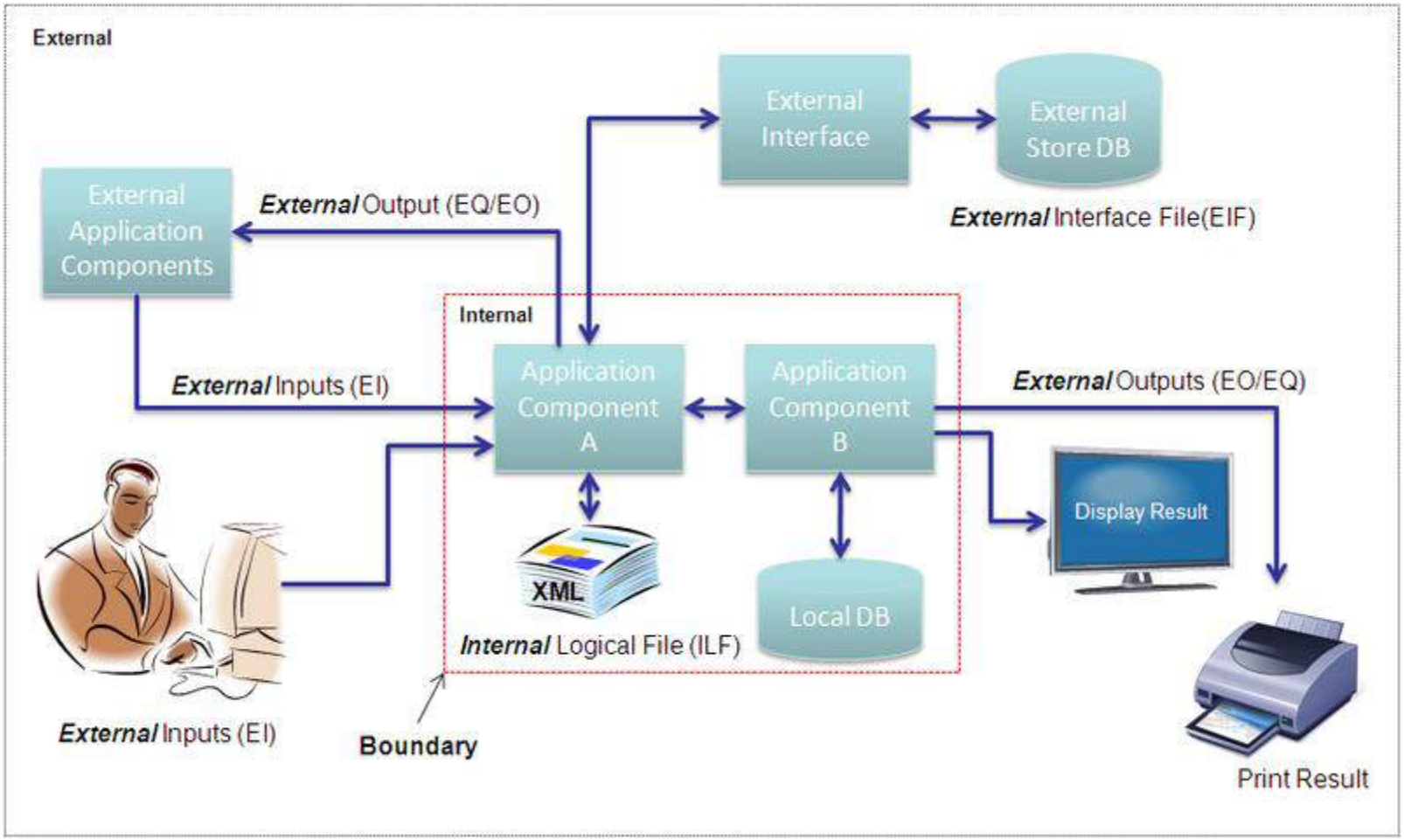
Function Point Computation

- FPs of an application is found out by counting the number and types of functions used in the applications.
- Various functions used in an application can be put under five types as shown in Table:

Measurement Parameter	*Examples*
1. Number of external inputs (EI)	Input screen and tables.
2. Number of external outputs (EO)	Output screens and reports.
3. Number of external inquiries (EQ)	Prompts and interrupts.
4. Number of internal files (ILF)	Databases and directories.
5. Number of external interfaces (EIF)	Shared databases and shared routines.

- The 5 parameters mentioned are also known as information domain characteristics.
- All these parameters are then individually assessed for complexity.

FPA components



Function Point Computation.....

- The calculation begins with the counting of the five function types of a project or application:
- These 5 function types are then ranked according to their complexity: Low, Average or High, using a set of prescriptive standards.
- These counts are then multiplied with the corresponding weights(complexity) values and the values are added up to determine the **UFP** (Unadjusted Function Point) or **count total** of the subsystem.

Function Point Computation.....

- Organizations that use FP methods, develop criteria for determining whether a particular entry is Low, Average or High.
- Nonetheless, the determination of complexity is somewhat subjective.

Table 2.3 Computing FPs

Measurement Parameter	*Count*		*Weighing factor*			
			Simple	*Average*	*Complex*	
1. Number of external inputs (EI)	—	*	3	4	6 =	—
2. Number of external outputs (EO)	—	*	4	5	7 =	—
3. Number of external inquiries (EQ)	—	*	3	4	6 =	—
4. Number of internal files (ILF)	—	*	7	10	15 =	—
5. Number of external interfaces (EIF)	—	*	5	7	10 =	—
Count-total →						—

Function Point Computation.....

complexity adjustment value/ factor

- The last step involves assessing the environment and processing complexity of the project or application as a whole.
- In this step, the impact of 14 general system characteristics is rated on a scale from 0 to 5 in terms of their likely effect on the project or application.

complexity adjustment value/ factor

- complexity adjustment value/ factor are calculated based on responses to the following questions

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require online data entry?
7. Does the online data entry require the input transaction to be built over multiple screens or operations?
8. Are the ILFs updated online?

Computing Value Adjustment Factor....

9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

complexity adjustment value/ factor

- Each of these questions is answered using an ordinal scale that ranges from 0 (not important or applicable) to 5 (absolutely essential).
    - 0 – No influence
    - 1 – Incidental
    - 2 – Moderate
    - 3 – Average
    - 4 – Significant
    - 5 - Essential
- The constant values in FP Equation and the weighting factors that are applied to information domain counts are determined empirically.

Function Point Computation.....

- The Function Point (FP) is thus calculated with the following formula

$$\mathbf{FP = Count\text{-}total * [0.65 + 0.01 * \sum(Fi)]}$$

$$\mathbf{= Count\text{-}total * CAF}$$

where Count-total is obtained from the table 2.3.

$$CAF = [0.65 + 0.01 * \sum(Fi)]$$

- $\sum(Fi)$ is the sum of all 14 questionnaires and show the **complexity adjustment value/ factor-CAF** (where i ranges from 1 to 14).

Function Point Computation.....

- $\sum$(Fi) ranges from 0 to 70,

  i.e., 0 <= $\sum$(Fi) <=70 and CAF ranges from 0.65 to 1.35

- because

  (a) When $\sum$(Fi) = 0 then CAF = 0.65

  (b) When $\sum$(Fi) = 70 then CAF = 0.65 + (0.01 * 70) = 0.65 + 0.7 = 1.35

Function Point Computation.....

- Based on the FP measure of software many other metrics can be computed:

  (a) Errors/FP

  (b) $/FP.

  (c) Defects/FP

  (d) Pages of documentation/FP

  (e) Errors/PM.

  (f) Productivity = FP/PM (effort is measured in person-months).

  (g) $/Page of Documentation.

Example

Given the following values, compute function point and productivity when all complexity adjustment factor (CAF) and weighting factors are **average**.

User Input = 50

User Output = 40

User Inquiries = 35

User Files = 6

External Interface = 4

Effort = 36.9 p-m

Example.....

- **Solution:**
- **Step-1:**

  As complexity adjustment factor is average (given in question), hence, scale = 3.

$$\sum(\text{Fi}) = 14 * 3 = 42$$

- **Step-2:**

$$\text{CAF} = [0.65 + 0.01 * \sum(\text{Fi})]$$

$$\text{CAF} = 0.65 + (0.01 * 42) = 1.07$$

Example.....

- **Step-3:**

  As weighting factors are also average (given in question) hence we will multiply each individual function point to corresponding values in function point table 2.3

Count Total = (50*4) + (40*5) + (35*4) + (6*10) + (4*7)
= 628

- **Step-4:**

FP = Count-total * [0.65 + 0.01 * ∑(Fi)]
= Count-total * CAF
= 628 * 1.07 = 671.96

Example.....

Productivity = FP/ Effort

= 671.96/36.9

= 18.21

Function Point Controversy

- Like the KLOC measure, function point use also has proponents and opponents
- Proponents claim that
  - FP is programming language independent
  - FP is based on data that are more likely to be known in the early stages of a project, making it more attractive as an estimation approach
- Opponents claim that
  - FP requires some "sleight of hand" because the computation is based on subjective data.
  - FPA has been criticized as not being universally applicable to all types of software.
    - For example, FPA doesn't capture all functional characteristics of real-time software

Object-oriented Metrics

- If you are planning to develop a software through an object oriented approach, then you can go for object oriented metrics.

- Conventional metrics are not provide granularity for schedule and effort estimation for incremental or evolutionary projects.

Object-oriented Metrics.....

- Lorenz and Kidd [Lor94] suggest the following set of metrics for OO projects:
- **Number of scenario scripts** (i.e., use cases)
  - detailed sequence of steps that describes the interaction between the user and the application.
  - Each script is organized into triplets of the form

    { **initiator,** ***action, participant*** }
  - This number is directly related to the size of an application and to the number of test cases required to test the system

Object-oriented Metrics

- **Number of key classes** (the highly independent components)
  - Key classes are defined early in object-oriented analysis and are central to the problem domain
  - This number indicates the amount of effort required to develop the software
  - It also indicates the potential amount of reuse to be applied during development
- **Number of support classes**
  - Support classes are required to implement the system but are not immediately related to the problem domain (e.g., user interface, database, computation)

Object-oriented Metrics...

- **Average number of support classes per key class**
  - Key classes are identified early in a project (e.g., at requirements analysis)
  - Estimation of the number of support classes can be made from the number of key classes
  - GUI applications have between two and three times more support classes as key classes
  - Non-GUI applications have between one and two times more support classes as key classes
- **Number of subsystems**
  - A subsystem is an aggregation of classes that support a function that is visible to the end user of a system

Use Case-Oriented Metrics

- Describes customer-level or business domain requirements that imply software features and functions.
- Use case is defined early in the software process, allowing it to be used for estimation before significant modeling and construction activities are initiated.
- Independent of programming language.
- The number of use cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Use Case-Oriented Metrics...

- Use cases can be created at vastly different levels of abstraction, there is no standard "size" for a use case.
- *use-case points (UCPs) as a mechanism for estimating* project effort and other characteristics.
- The UCP is a function of the number of actors and transactions implied by the use-case models.

Metrics for Software Quality

- You can use measurement to assess the quality of the requirements and design models, the source code, and the test cases that have been created as the software is engineered.
- A project manager must also evaluate quality as the project progresses.
- Private metrics collected by individual software engineers are combined to provide project-level results.
- Although there are many measures of software quality, correctness, maintainability, integrity, and usability provide useful indicators for the project team.

Metrics for Software Quality.....

- Correctness
  - This is the number of defects per KLOC, where a defect is a verified lack of conformance to requirements
  - Defects are those problems reported by a program user after the program is released for general use
- Maintainability
  - This describes the ease with which a program can be corrected if an error is found, adapted if the environment changes, or enhanced if the customer has changed requirements
  - Mean time to change (MTTC) : the time to analyze, design, implement, test, and distribute a change to all users
    - Maintainable programs on average have a lower MTTC

Metrics for Software Quality.....

- **integrity**
  - measures system's ability to withstand attacks on its security
- To measure integrity, two additional attributes must be defined:
    - threat
    - security.
- *Threat is the* probability that an attack of a specific type will occur within a given time.
- *Security* is the probability that the attack of a specific type will be repelled.

Metrics for Software Quality.....

- The integrity of a system can then be defined as:

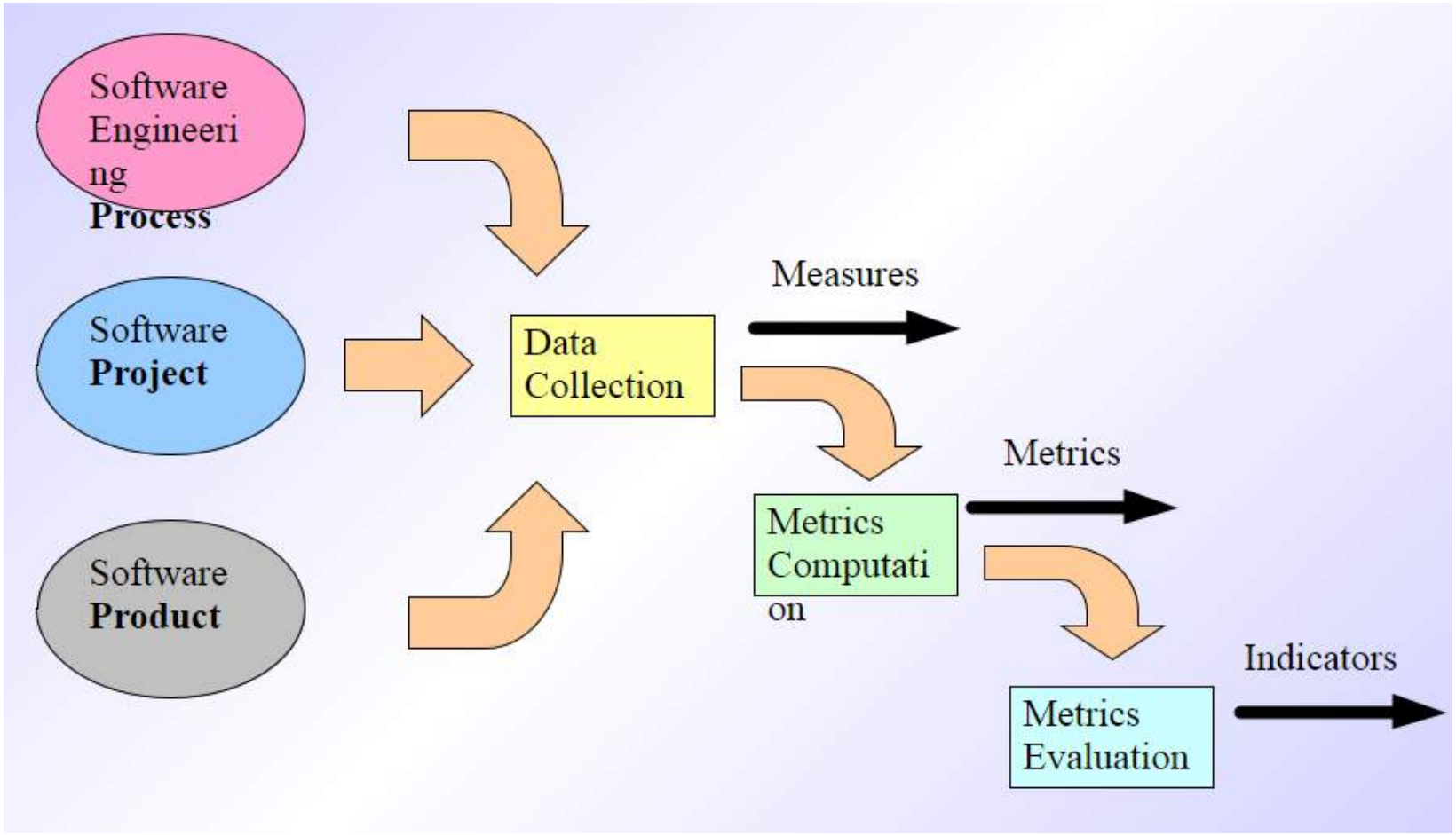
  $$\text{Integrity} = \sum [1 - (\text{threat} * (1 - \text{security}))]$$

- For example, if threat (the probability that an attack will occur) is 0.25 and security (the likelihood of repelling an attack) is 0.95, the integrity of the system is 0.99 (very high).
- **Usability.**
  - Usability is an attempt to quantify ease of use and can be measured in terms of the characteristics

Arguments for Software Metrics

- Most software developers do not measure, and most have little desire to begin
- Establishing a successful company-wide software metrics program can be a multi-year effort
- But if we do not measure, there is no real way of determining whether we are improving
- Measurement is used to establish a process baseline from which improvements can be assessed
- Software metrics help people to develop better project estimates, produce higher-quality systems, and get products out the door on time

Software Metrics Baseline Process



Establishing a Metrics Baseline

- By establishing a metrics baseline, benefits can be obtained at the software process, product, and project levels
- The same metrics can serve many masters
- The baseline consists of data collected from past projects
- Baseline data must have the following attributes
  - Data must be reasonably accurate (guesses should be avoided)
  - Data should be collected for as many projects as possible
  - Measures must be consistent (e.g., a line of code must be interpreted consistently across all projects)
  - Past applications should be similar to the work that is to be estimated
- After data is collected and metrics are computed, the metrics should be evaluated and applied during estimation, technical work, project control, and process improvement