Object Oriented Modelling And Design

Vasudevan T V

Module I

- Overview of Object Oriented Systems
- History of Object Orientation
- Qualities of an Object Oriented Model

Module II

- History of UML
- UML notation for classes, attributes and operations
- Class diagrams
- Object Interaction Diagrams
    - Collaboration Diagrams
    - Sequence Diagrams
- State Diagrams
- Activity Diagrams
- Use Case Diagrams

Module III

- Architecture Diagrams
  - Package Diagrams
  - Deployment Diagrams
- Interface Diagrams
  - Window Layout Diagrams
  - Window Navigation Diagrams

Module IV

- Encapsulation Structure
- Connascence
- Domains of Object Classes

- Encumbrance
- Class Cohesion
- State Space and behaviour of classes and subclasses

- class invariant
- preconditions and postconditions
- principle of type conformance
- principle of closed behaviour

Module V

- Problems related with Inheritance and Polymorphism
- Mix-in Classes
- Class Cohesion
- Components

Reference Books

1.Page-Jones .M, Fundamentals of object-oriented design in UML, Addison Wesely

2.Booch. G, Rumbaugh J, and Jacobson. I, The Unified Modelling Language User Guide, Addison Wesely.

3.Bahrami.A, Object Oriented System Development, McGrawHill.

4.Booch. G, Rumbaugh J, and Jacobson. I, The Unified Modelling Language Reference Manual, Addison Wesely.

5.Larman.C, Applying UML & Patterns: An Introduction to Object Oriented Analysis & Design, Addison Wesley

6.Pooley R & Stevens P, Using UML: Software Engineering with Objects & Components, Addison Wesley.

Module I
Overview of Object Oriented Systems

- Objects
- Attributes
- Classes
- Encapsulation
- Inheritance
- Polymorphism
- Messages

- Objects

  Basic run-time entities in an object-oriented system

  eg: Customer, Account

- Attributes

  They are the properties or characteristics possessed by entities.

  eg:Customer - customer-name, customer-id, customer-city

  Account – branch-name, account-no, balance

- Classes

  A user-defined data type which is a collection of similar objects.

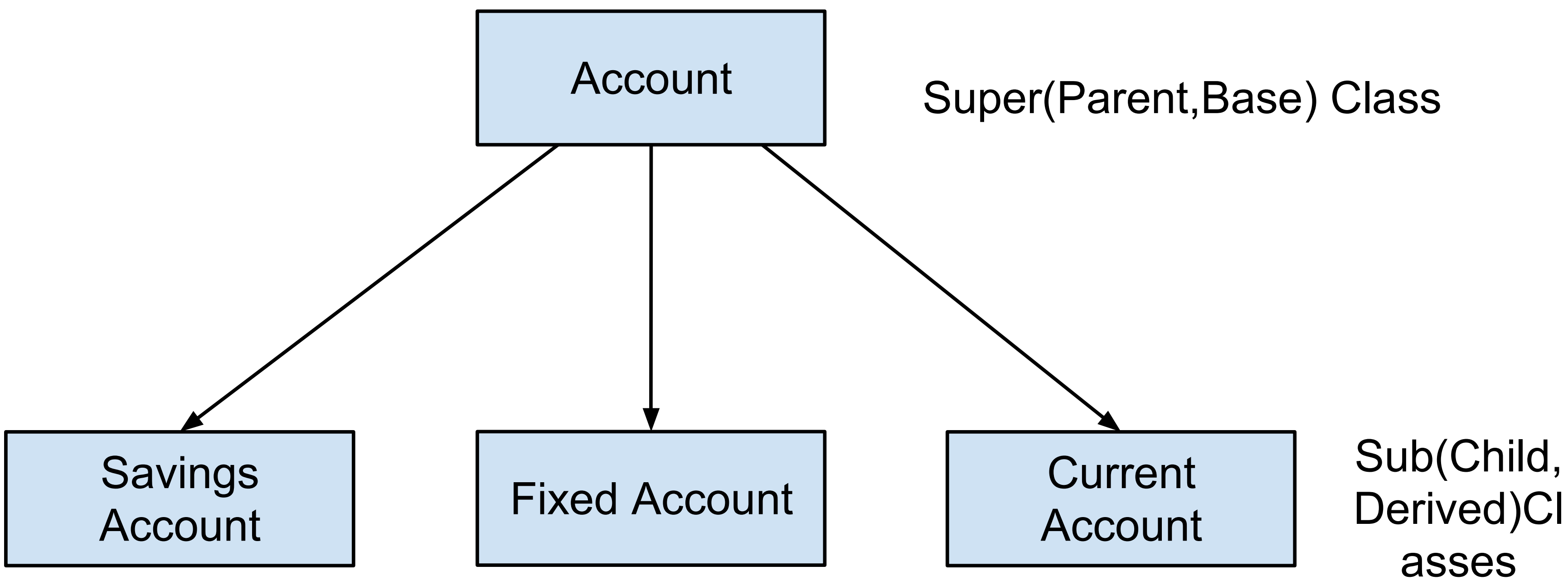
  eg: mango, apple and grape are objects of class fruit

- Encapsulation

The wrapping up of data and functions into a single unit (called class) is known as encapsulation.

- Inheritance

The process by which objects of one class acquire the properties of objects of another class.

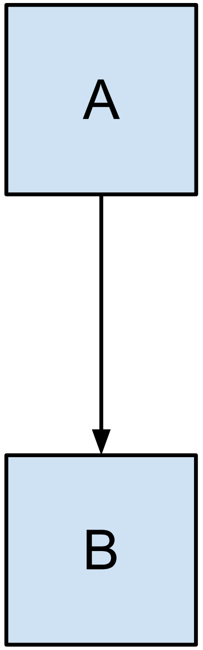


- Inheritance-contd

  It provides the idea of reusability

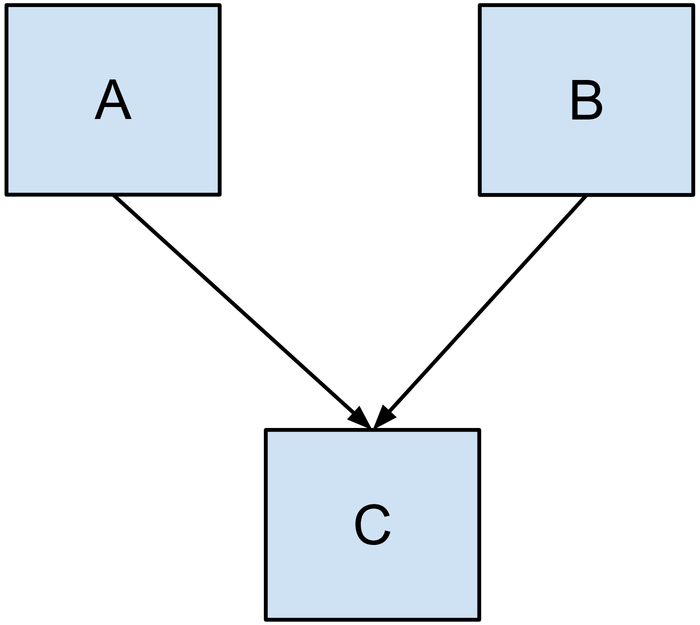
Types of Inheritance

1. Single Inheritance



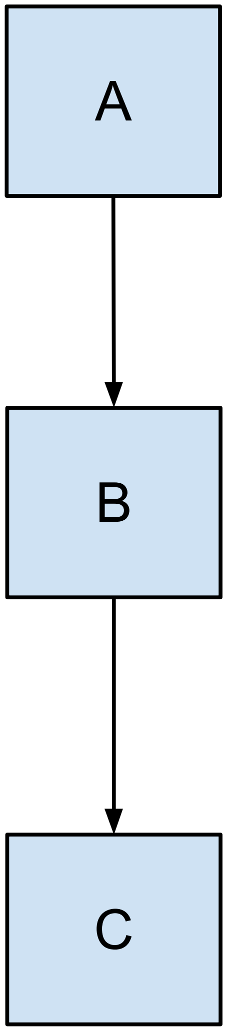
Types of Inheritance-contd

2. Multiple Inheritance



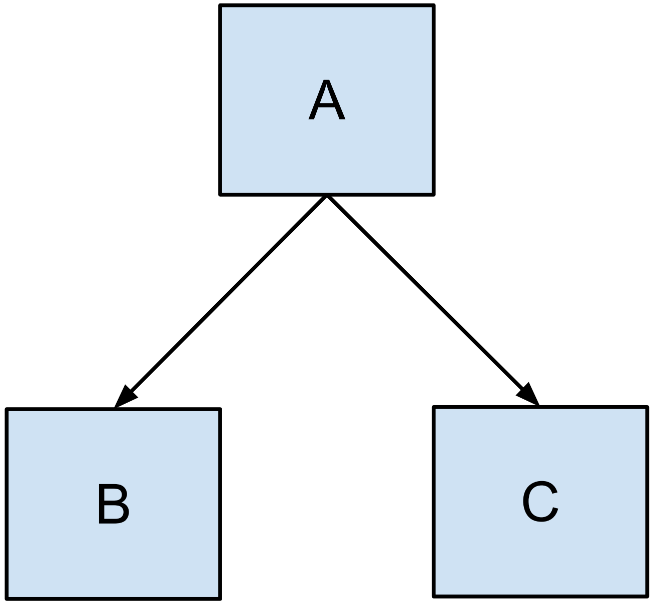
Types of Inheritance-contd

3. Multilevel Inheritance



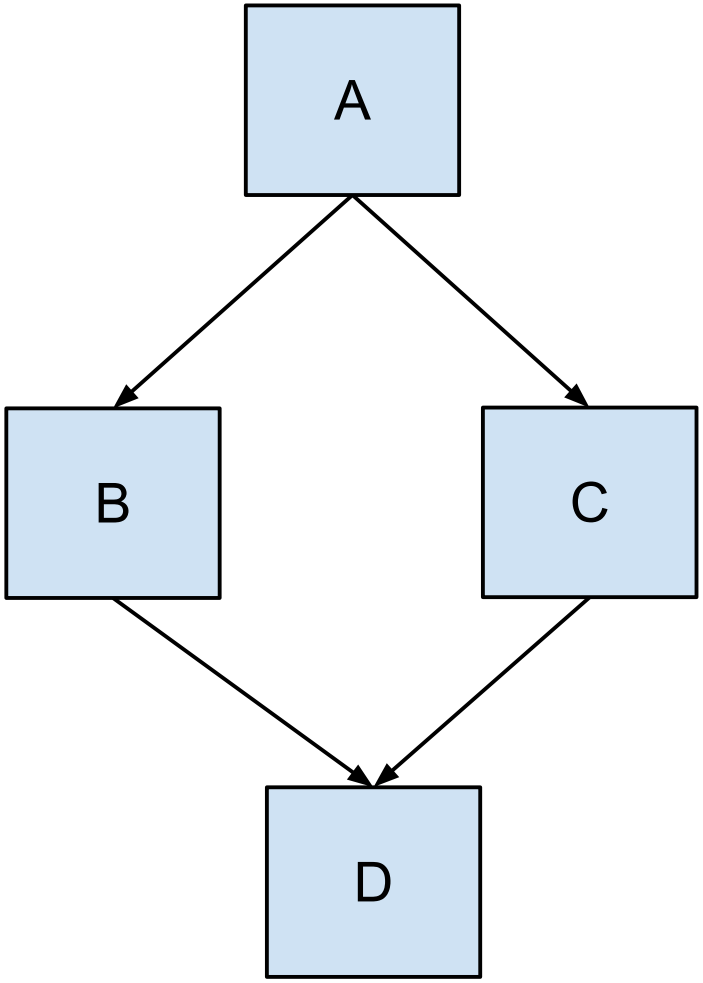
Types of Inheritance-contd

4. Hierarchical Inheritance



Types of Inheritance-contd

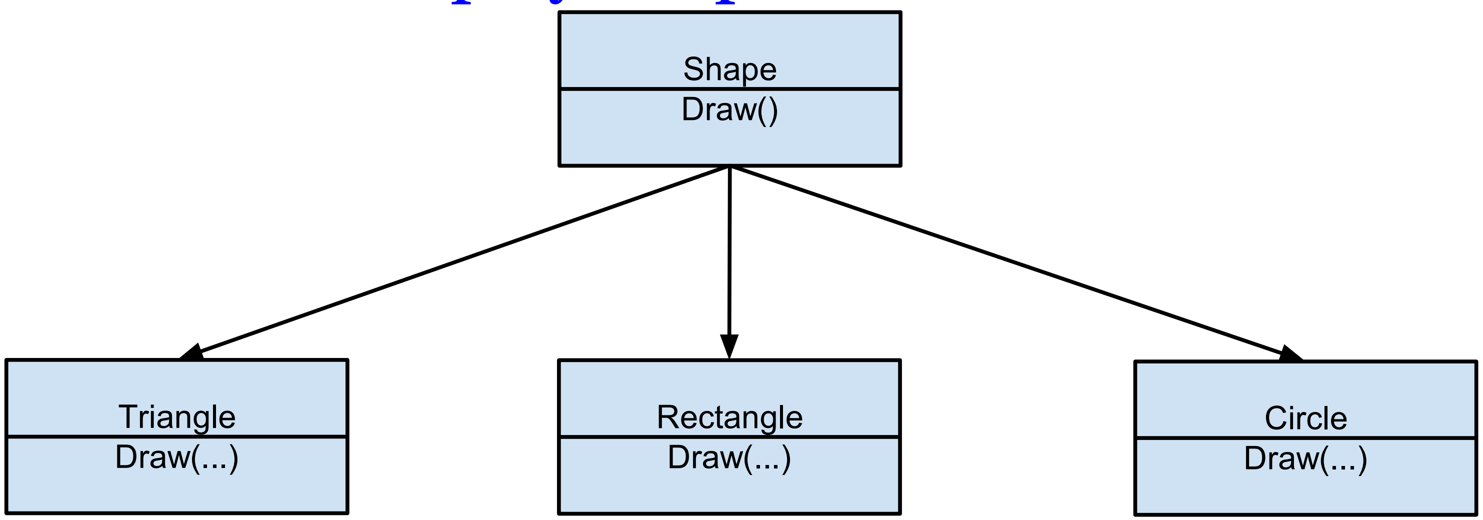
5. Hybrid Inheritance



- Polymorphism

  It means the ability to take more than one form

  - Compile time polymorphism
    1. function overloading
    2. operator overloading
  - Run time polymorphism



- Dynamic Binding
- Overriding

- Messages

  A request by the sender object to execute an operation of the target object

  <u>A message from obj1 to obj2</u>

handle name

obj2. operation (arguments)

Types of Messages

- Informative Message(update, forward or push message)

  eg: employee. got Married (marriage Date : Date)

- Interrogative Message(read, backward or pull message)

  eg: employee. get Name (out name : String)

- Imperative Message(force or action message)

  eg: rectangle. scale (factor : positive Real)

- History Of Object Orientation

People	Contribution
Larry Constantine	did research on criteria for good software design
O J Dahl , K Nygaard	developed the idea of class
Alan Kay, Adele Goldberg and their colleagues	developed the concepts of messages and inheritance
Edsger Dijkstra	developed the idea of encapsulation
Barbara Liskov	contributed to the idea of abstract data type

- History Of Object Orientation - contd

People	Contribution
David Parnas	developed the principles of good modular software construction
Jean Ichbiah and his group	developed the "Green" programming language which later became "Ada".It contained concepts such as Genericity and packages
Bjarne Stroustrup	developed C++
Bertrand Meyer	developed "Eiffel"
Grady Booch, Ivar Jacobson, Jim Rumbaugh	developed UML

- History of Object Orientation- contd

Language	Year in which Developed
Simula	1967
Small Talk	1980
Objective C	1983
C++	1983
Eiffel	1986
CLOS	Late 1980's
Java	1995

- Qualities of an Object Oriented Model
    - Reusability
    - Reliability
    - Robustness
    - Extensibility
    - Distributability
    - Storability

Module II

- History of UML
- UML notation for classes, attributes and operations
- Class diagrams
- Object Interaction Diagrams
  - Collaboration Diagrams
  - Sequence Diagrams
- State Diagrams
- Activity Diagrams
- Use Case Diagrams

- History of UML (Unified Modelling Language)

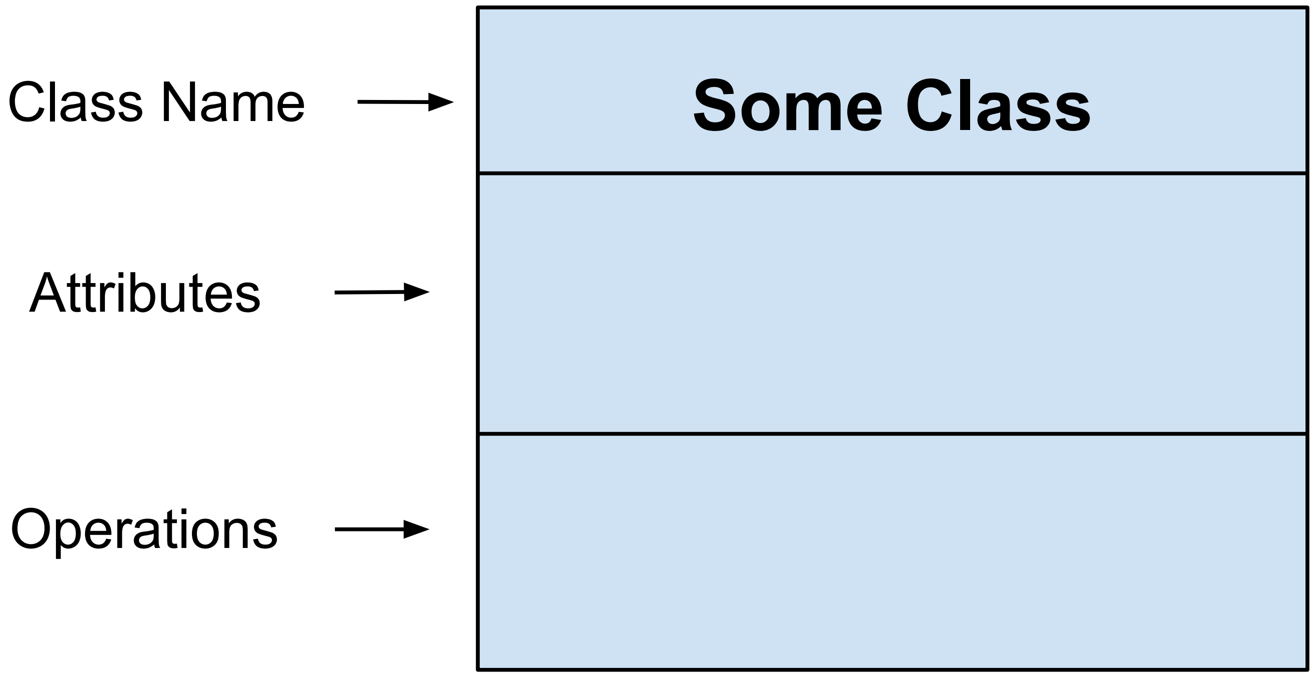
    - A modelling language selected as the standard object oriented modelling language by Object Management Group(OMG)

    - Developed by Grady Booch, Ivar Jacobson and Jim Rumbaugh

    - Contains several diagrams and notations used in object oriented design

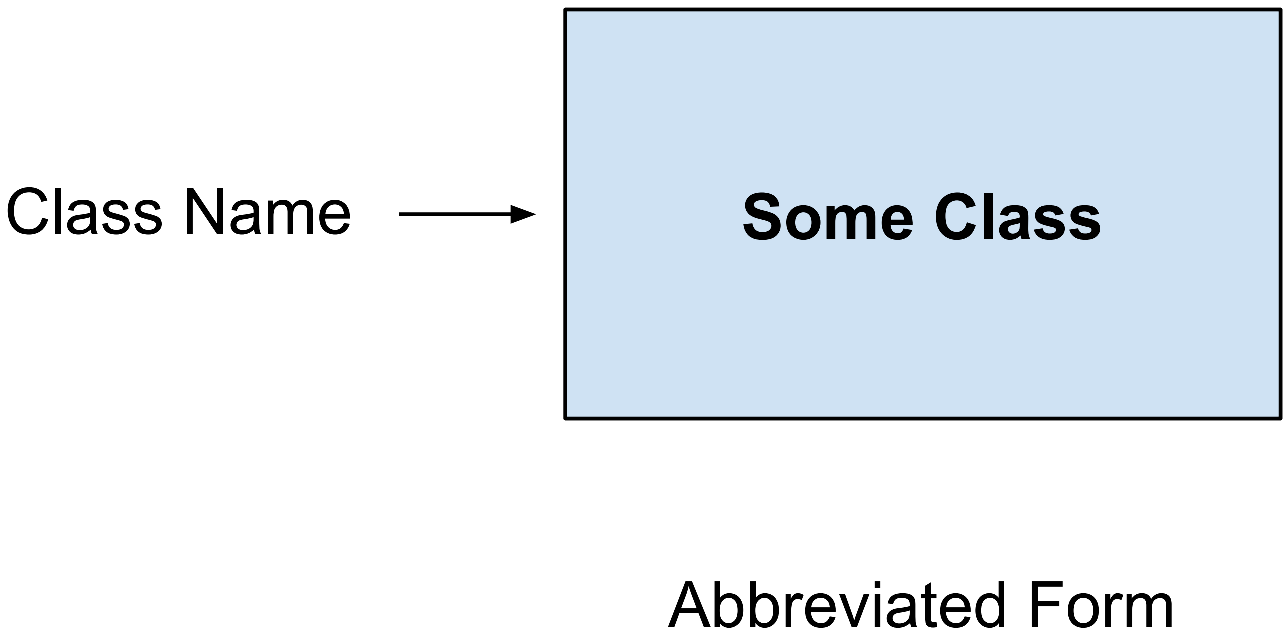
UML Notation for Classes, Attributes and Operations

Class(ADT definition diagram) - Notation

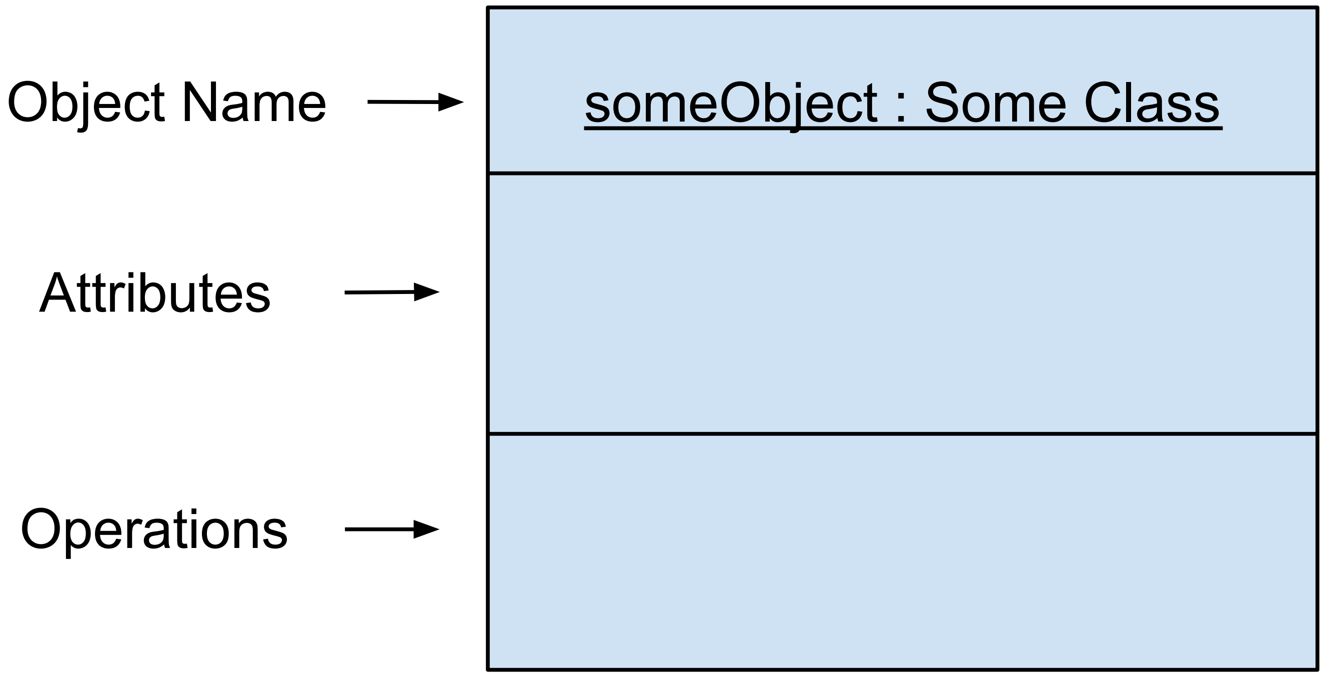


Full Form

Class - Notation - contd

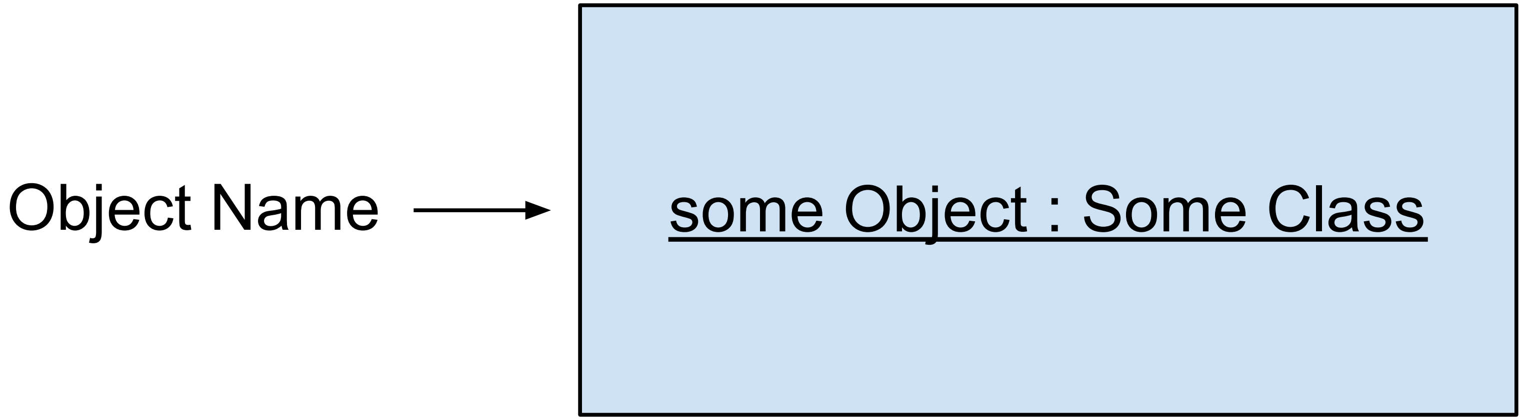


Object - Notation



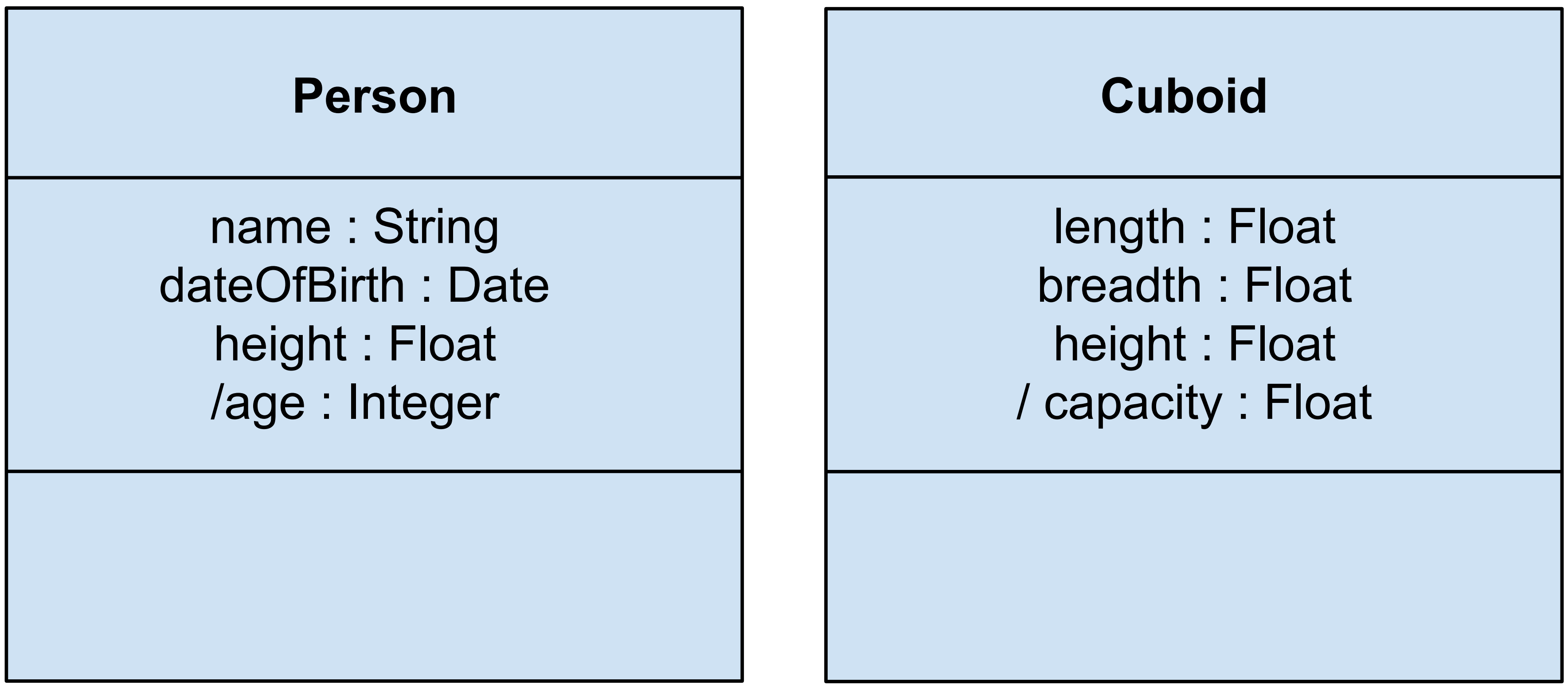
Full Form

Object - Notation - contd



Abbreviated Form

Attributes



read-only attributes are preceded by /

Operations

Person

name : String
dateOfBirth : Date
height : Float
/age : Integer

getName (out name : String)
setName (name : String)
....
getHeight (date : Date, out height : Length)
setHeight(date : Date, height : Length)

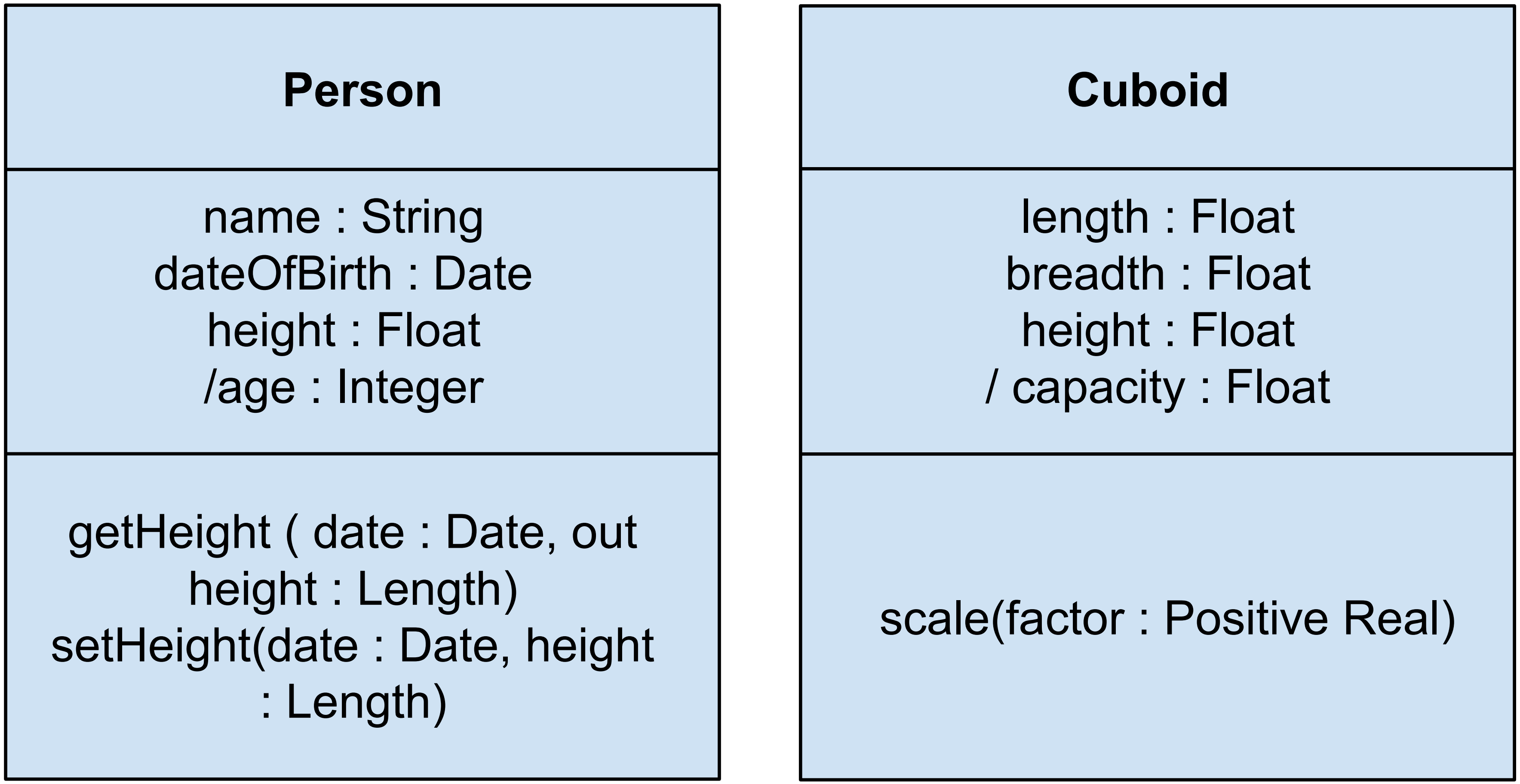
Cuboid

length : Float
breadth : Float
height : Float
/ capacity : Float

getLength (out length : Float)
setLength (length : Float)
....
getCapacity (out capacity : Float)
scale(factor : Positive Real)

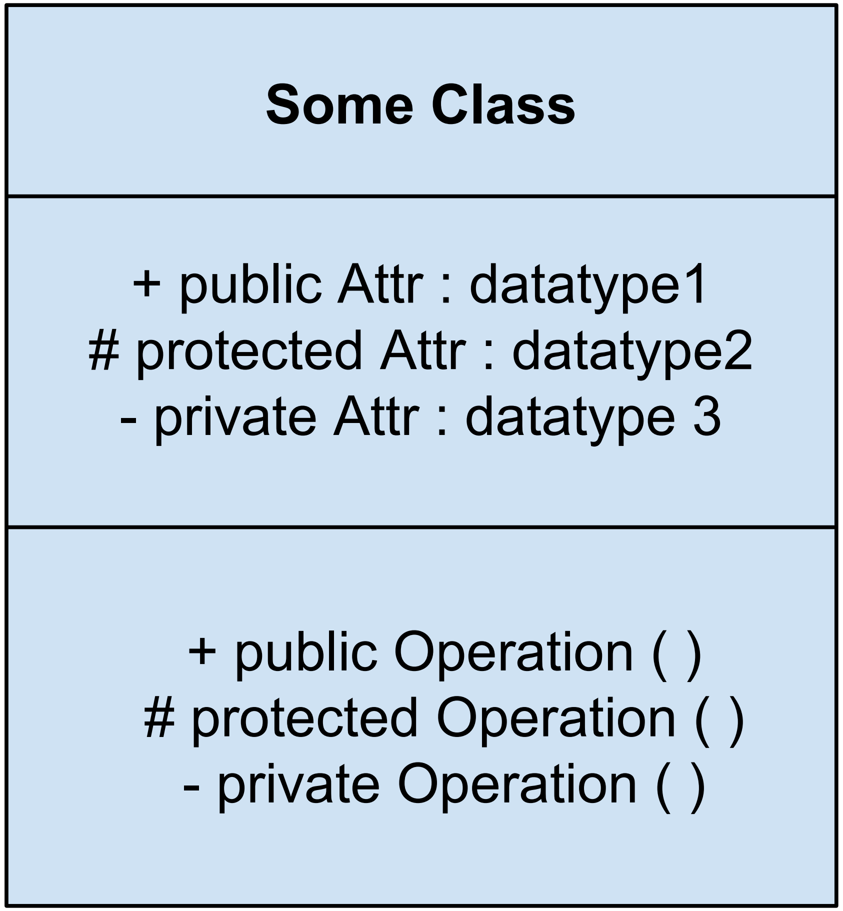
'in' specifies input argument (default)
'out' specifies output argument
'inout' specifies both input and output arguments

Operations- contd

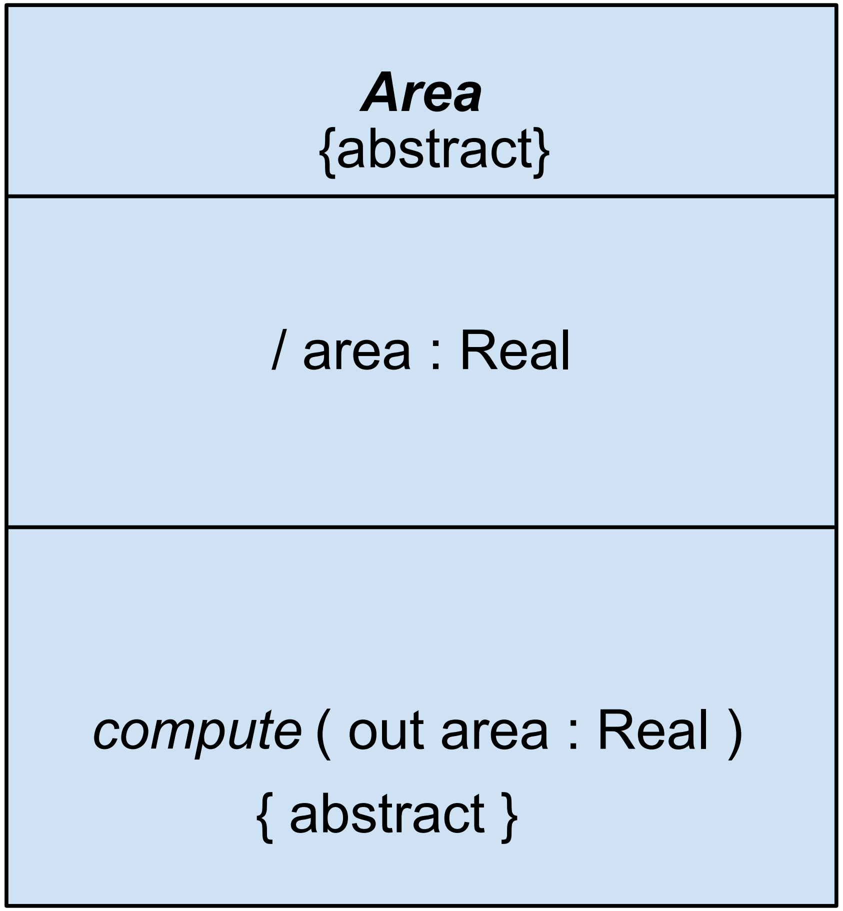


Simplified Notation

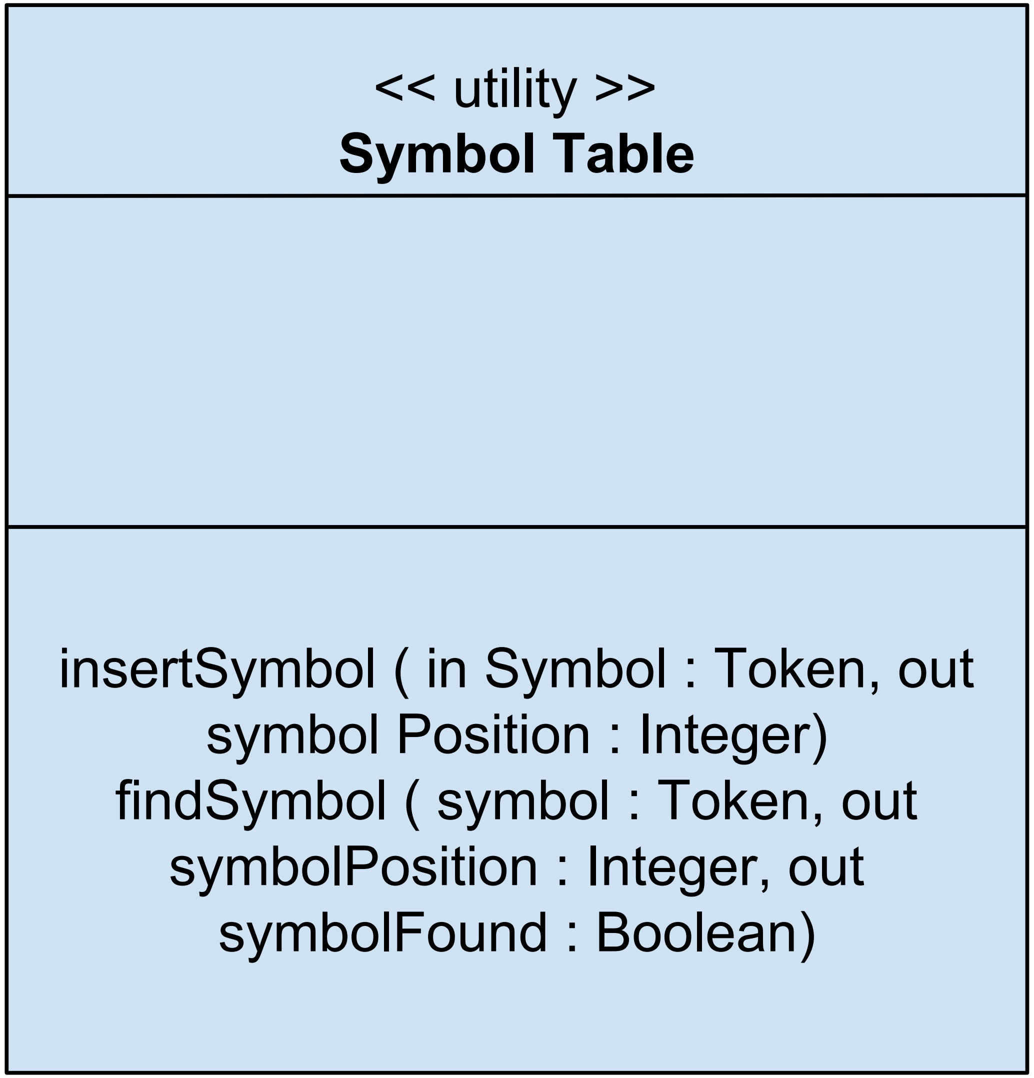
Visibility of Attributes and Operations



Abstract Classes and Abstract operations

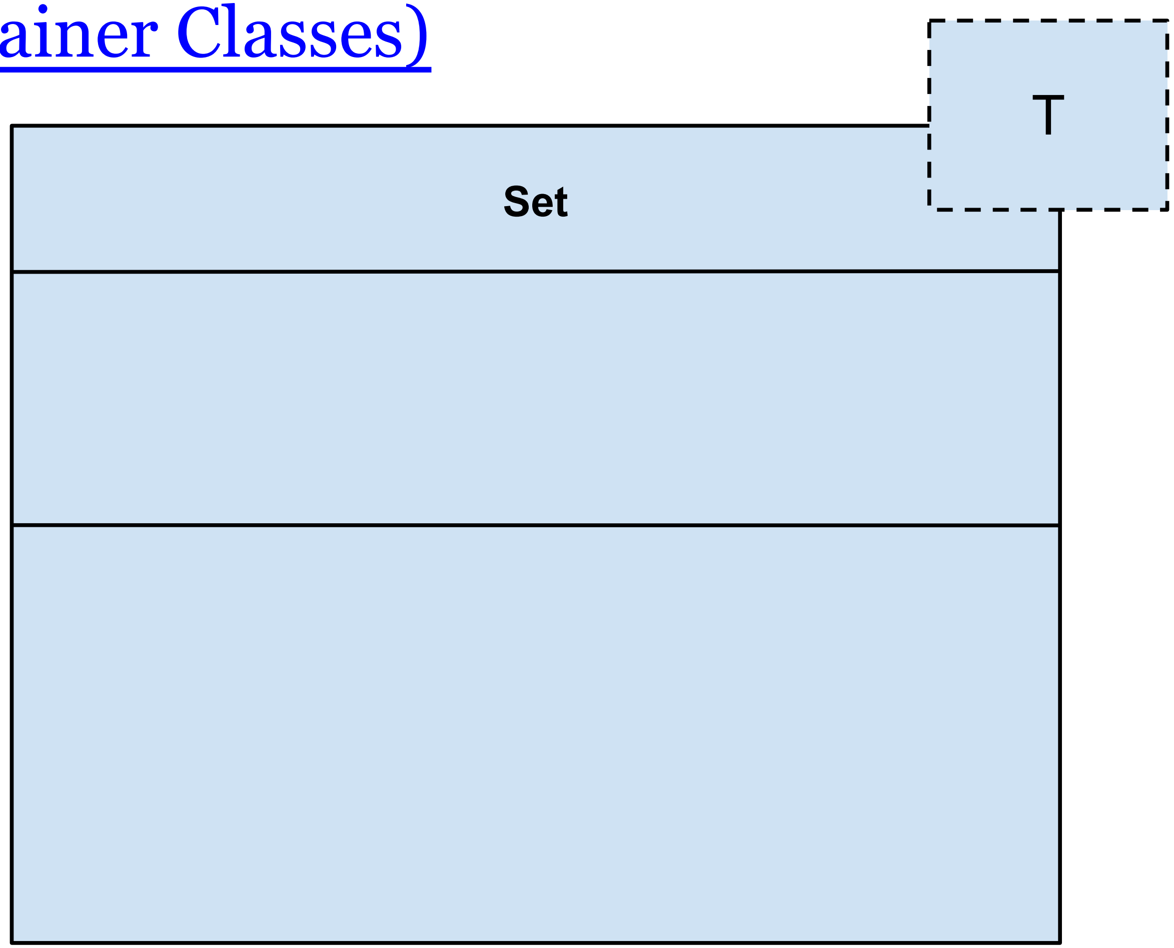


The Utility (Utility Package)



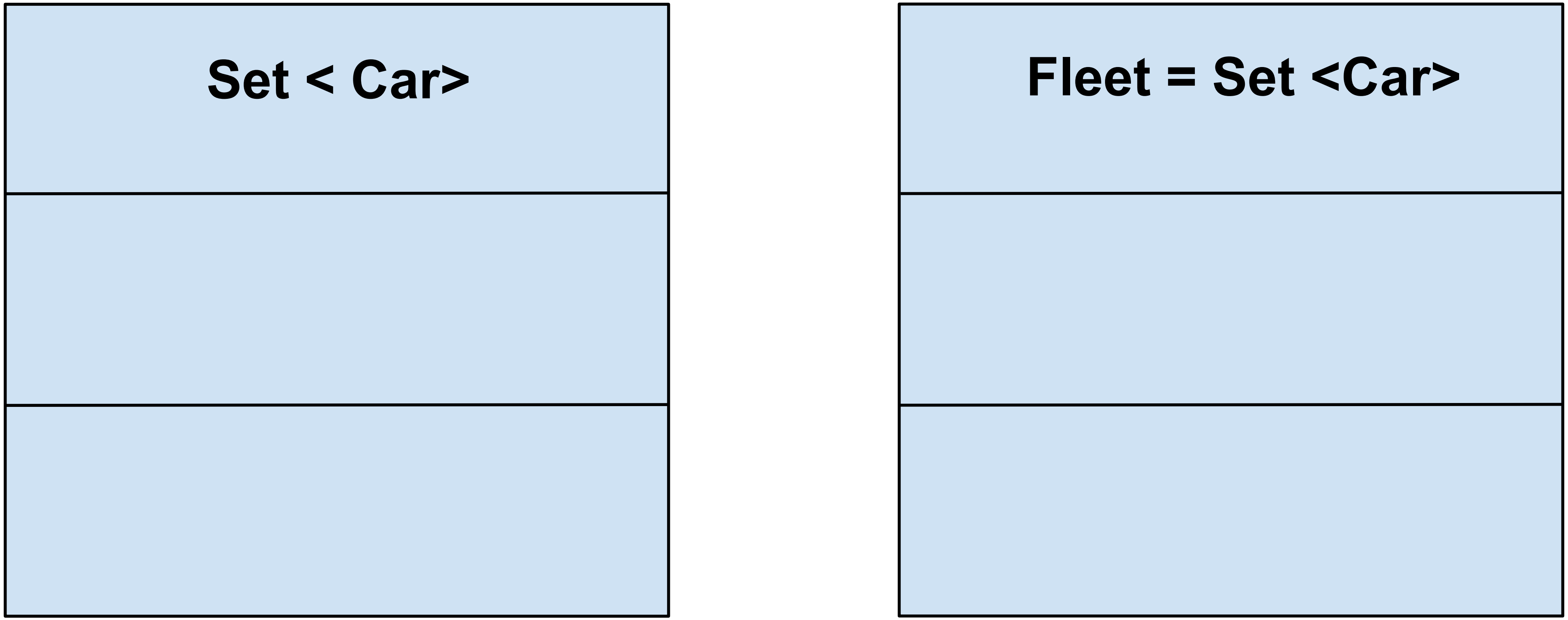
<<   >> are called guillemets

Parameterised Classes(Generic Classes)(Template Classes)(Container Classes)



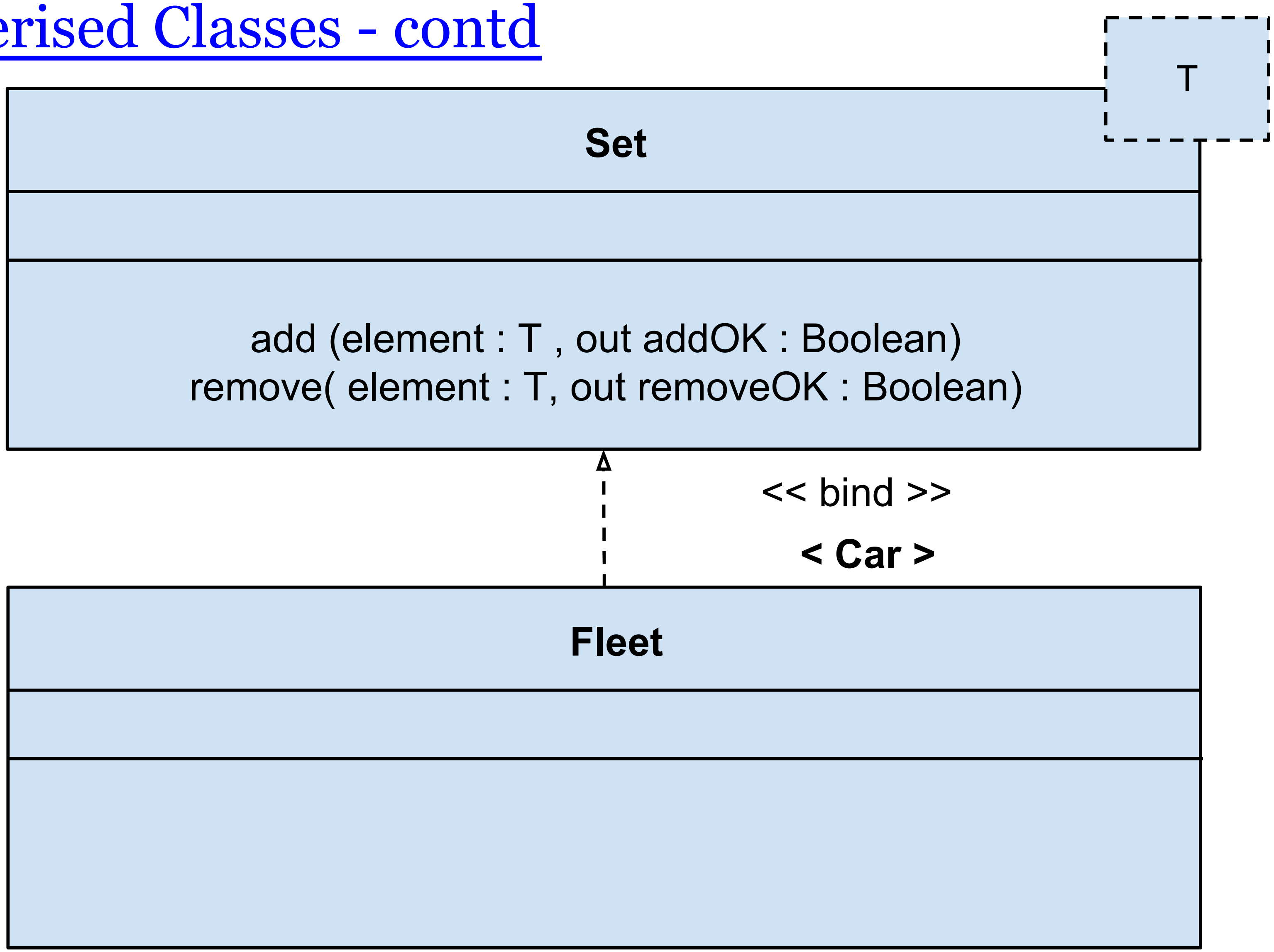
A parameterised class with one formal class argument

Parameterised Classes- contd



A bound class formed from a parameterised class

Parameterised Classes - contd



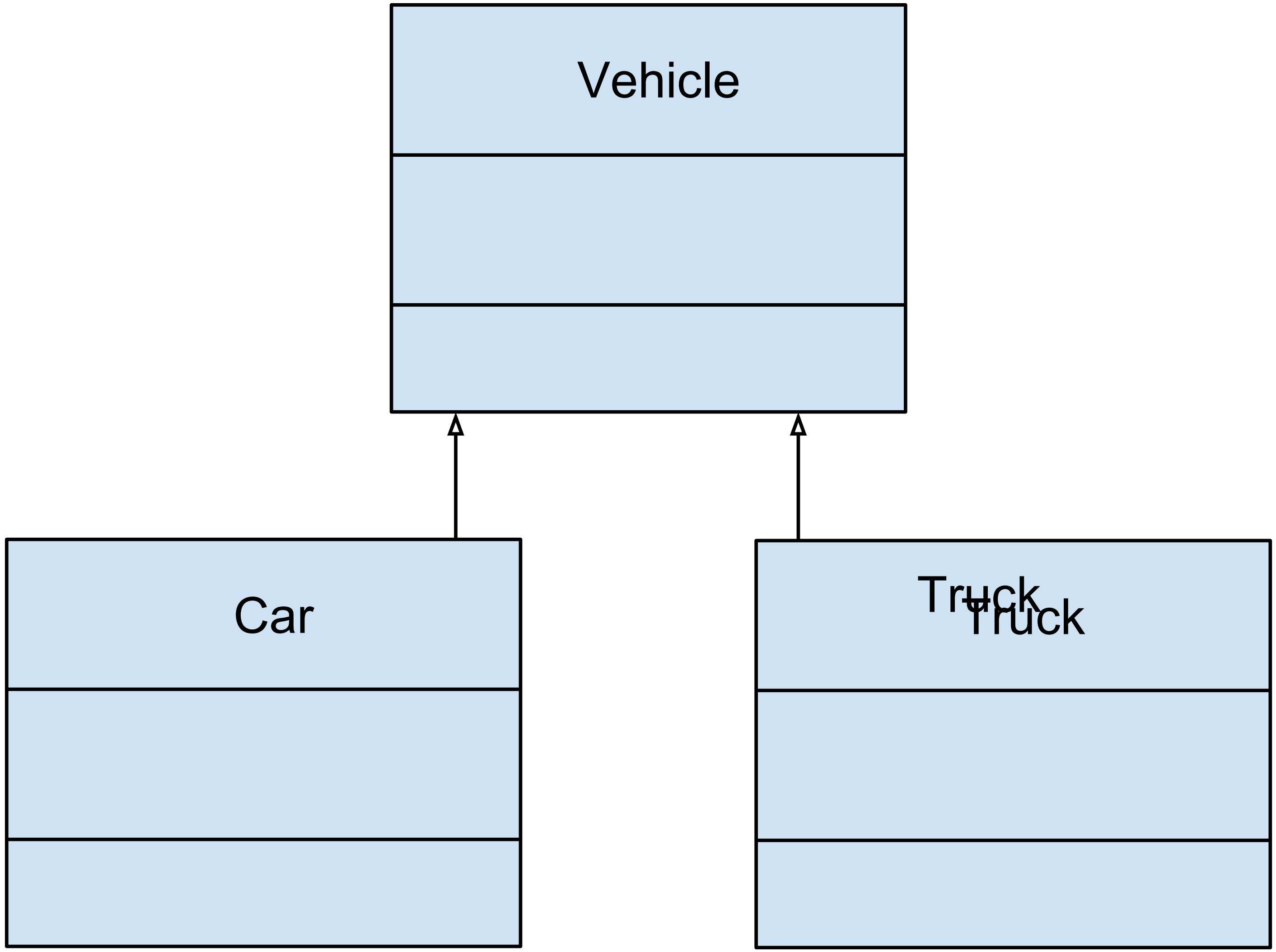
A bound class, formed from a parameterised class - another depiction

Class Diagrams

A class diagram is used for representing various relationships between classes viz.

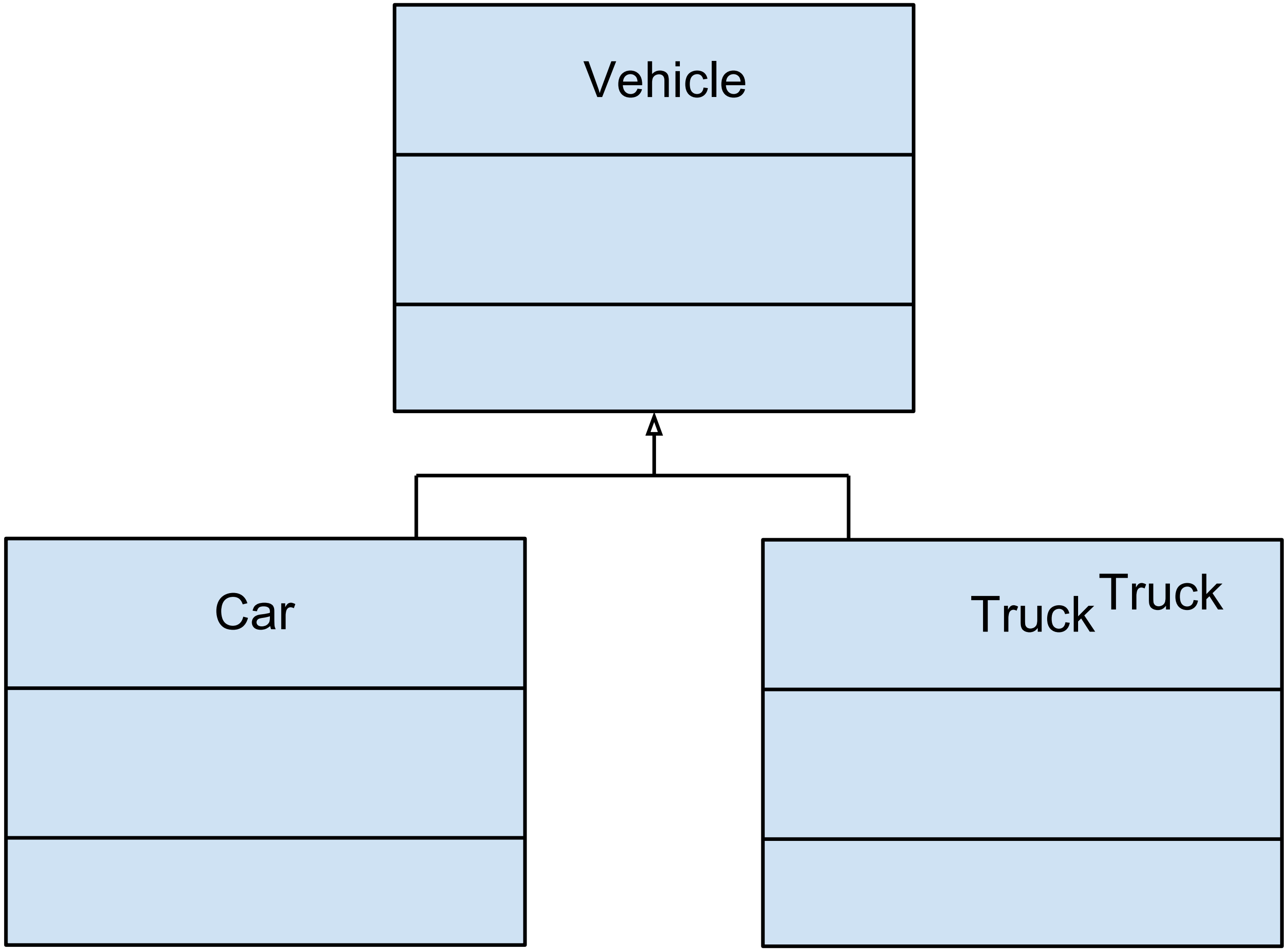
- Generalisation/Specialisation (Inheritance)
- Association
- Whole / Part Association

- Generalisation/Specialisation



representation 1

- Generalisation/Specialisation - contd



Vehicle
    Car
    Truck

representation 2

representation 3

Subclass Partitioning

- disjoint / overlapping partitioning
- complete / incomplete partitioning
- static / dynamic partitioning

Subclass Partitioning

- disjoint / overlapping partitioning

If an instance of the superclass belongs to only one group among the subclasses, then it is called disjoint partitioning

If an instance of the superclass belongs to more than one group among the subclasses, then it is called overlapping partitioning

Subclass Partitioning

- complete / incomplete partitioning

If all the instances of the superclass have a place in the subclass, then it is called complete partitioning

If all the instances of the superclass do not have a place in the subclasses, then it is called incomplete partitioning

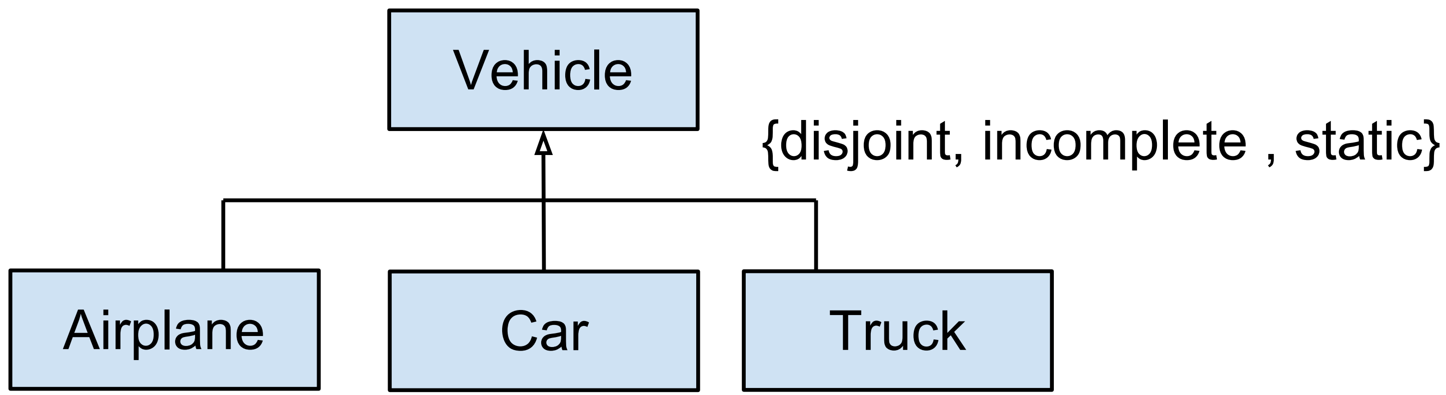
Subclass Partitioning

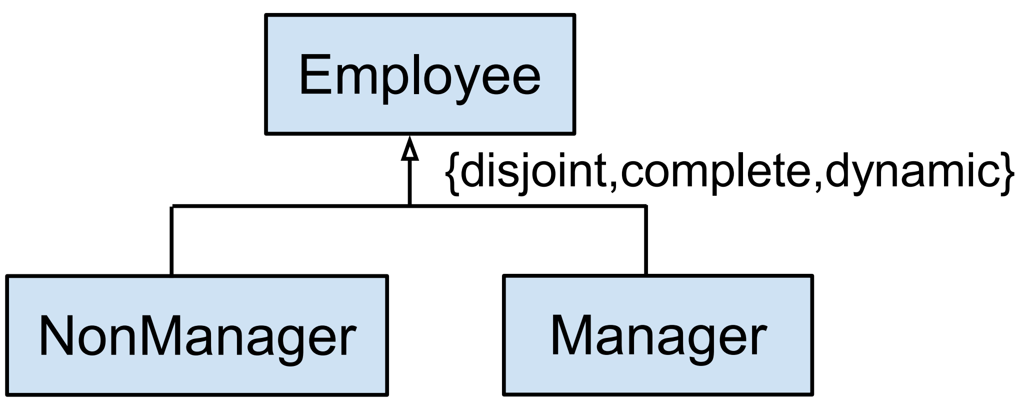
- static / dynamic partitioning

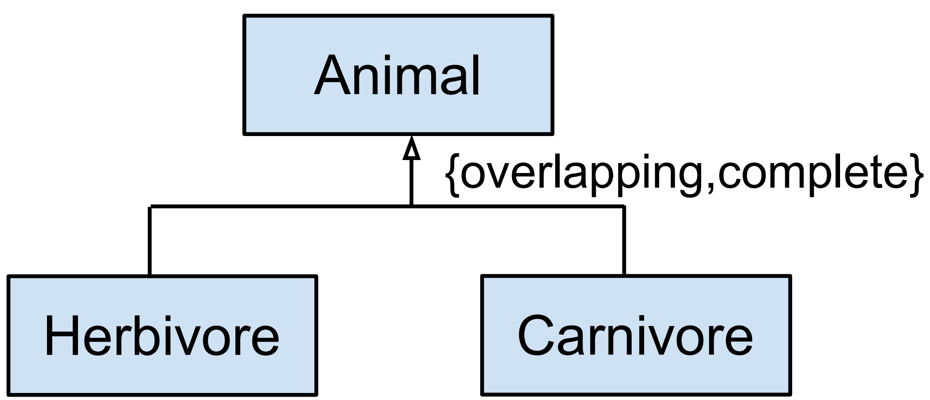
If an instance of the superclass belonging to one subclass at a certain period can belong to another subclass at another period, then it is called dynamic partitioning

If an instance of the superclass always belong to one and only one subclass, then it is called static partitioning

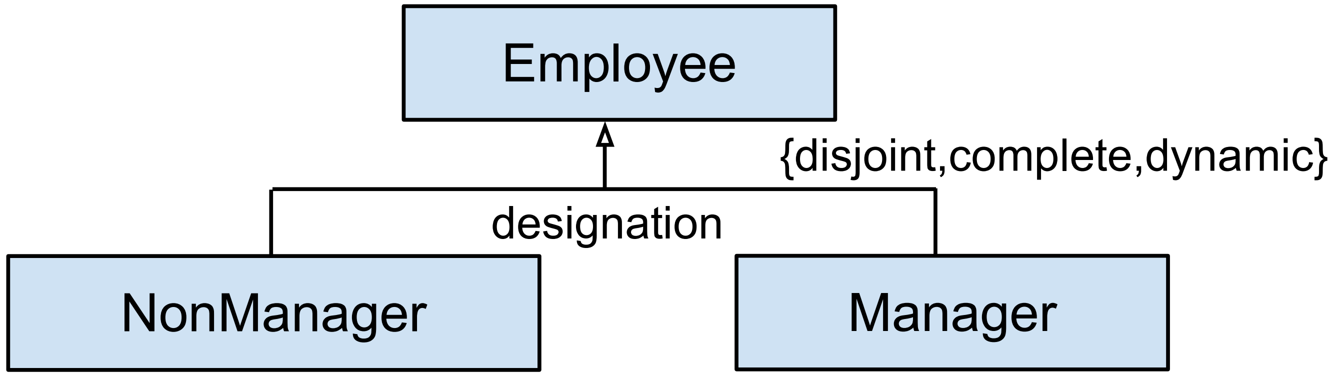
Subclass Partitioning - contd

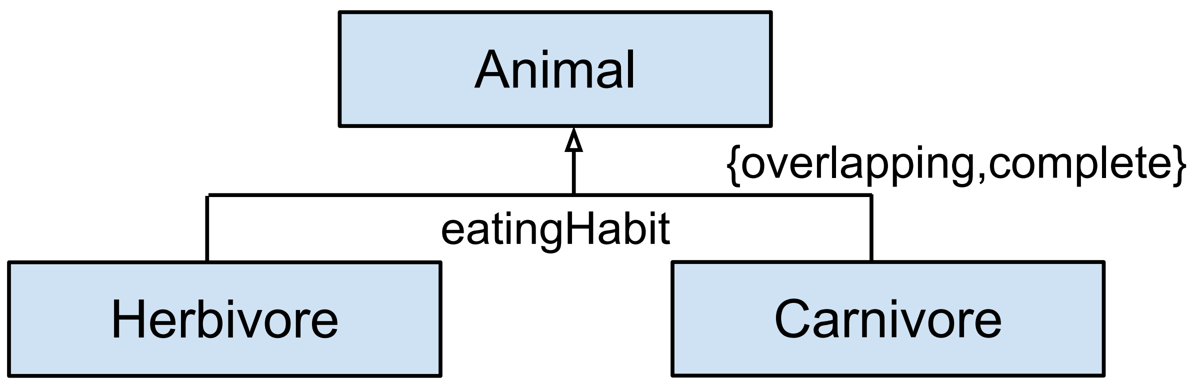






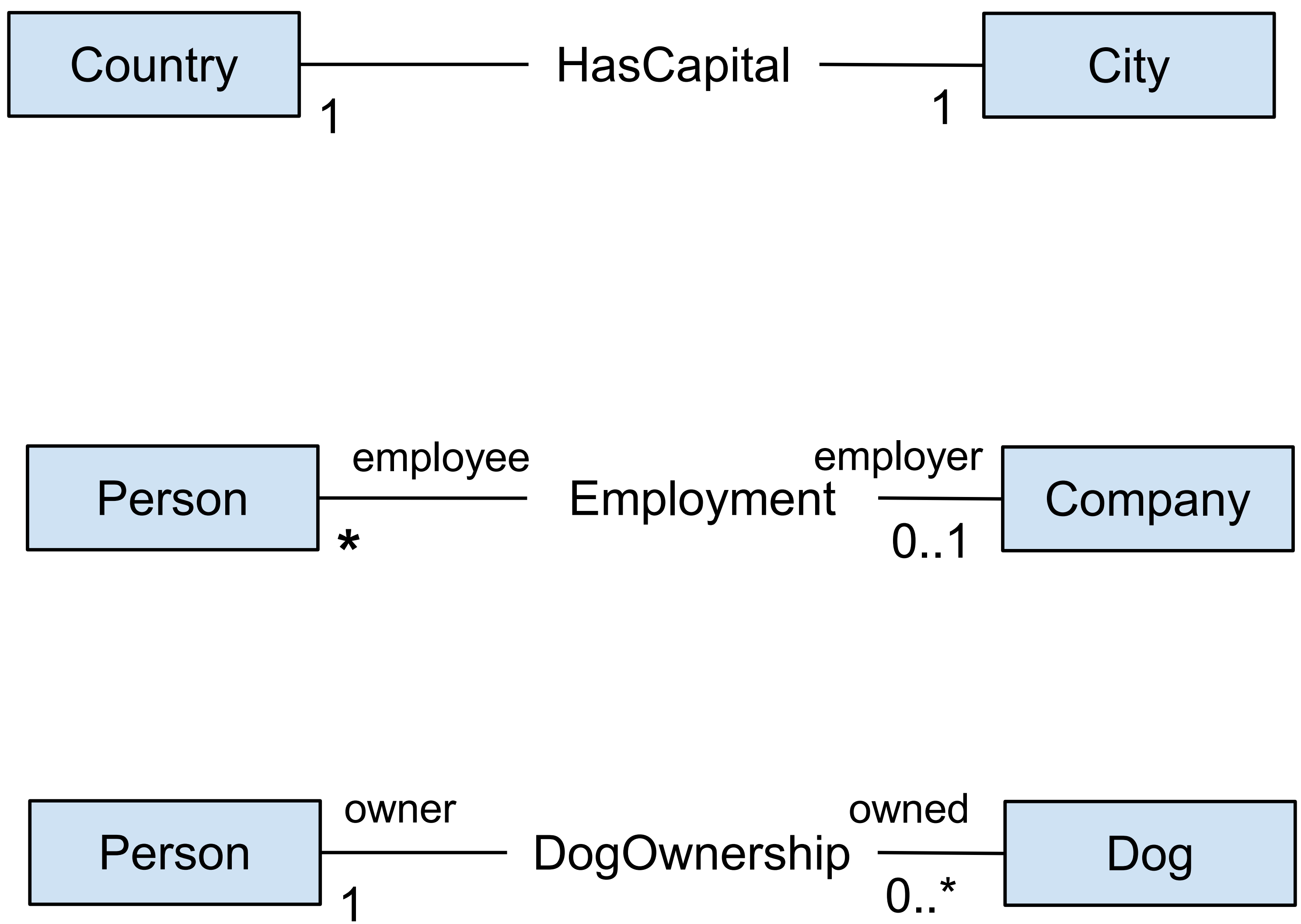
Partitioning Discriminators



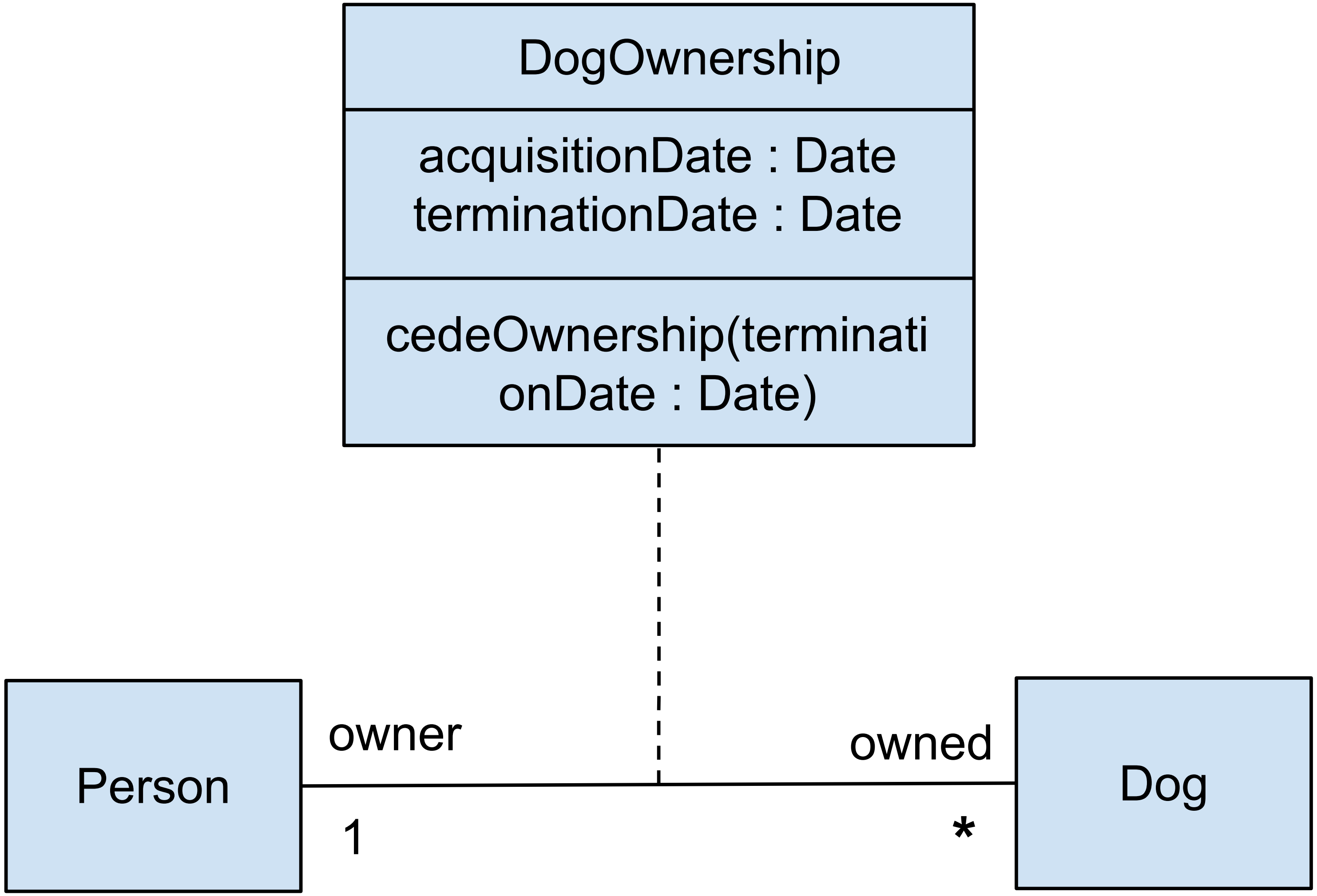


Association

- It is a conceptual relationship between classes

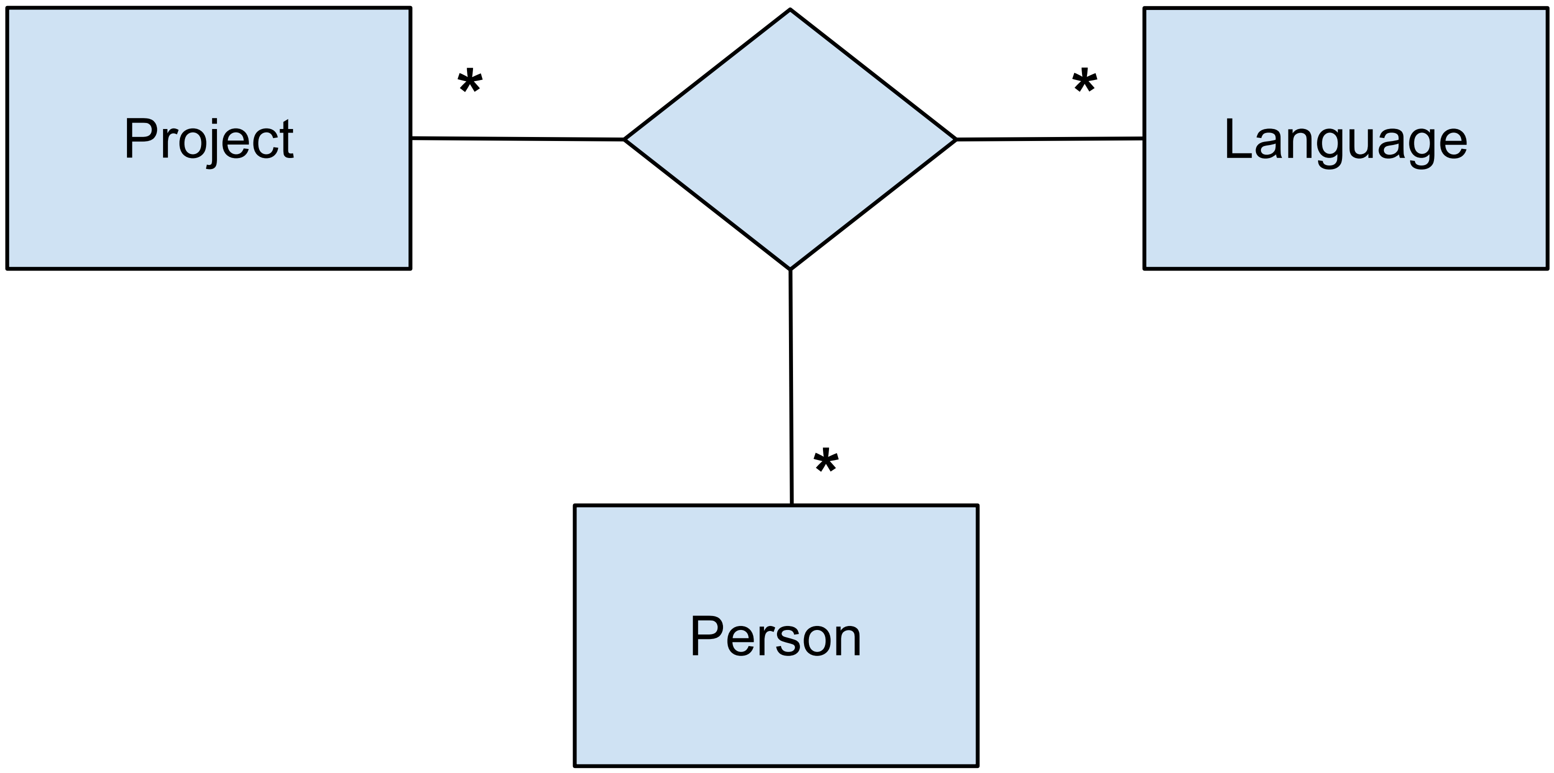


Association - contd



Association as a class

High - Order Associations



In a ternary association 3 classes participate
In a quaternary association 4 classes participate

Navigability of Associations

This navigability is provided by declaring a variable ownedDogs : Set <Dog> in the class Person

This navigability is provided by declaring a variable owner : Person in the class Dog

This navigability is provided by the combination of the above two declarations

Whole/Part Association

- A special kind of association in which objects of one class (part object) are contained within objects of another class(whole object)

- There are two types of whole/part association
  - Aggregation
  - Composition

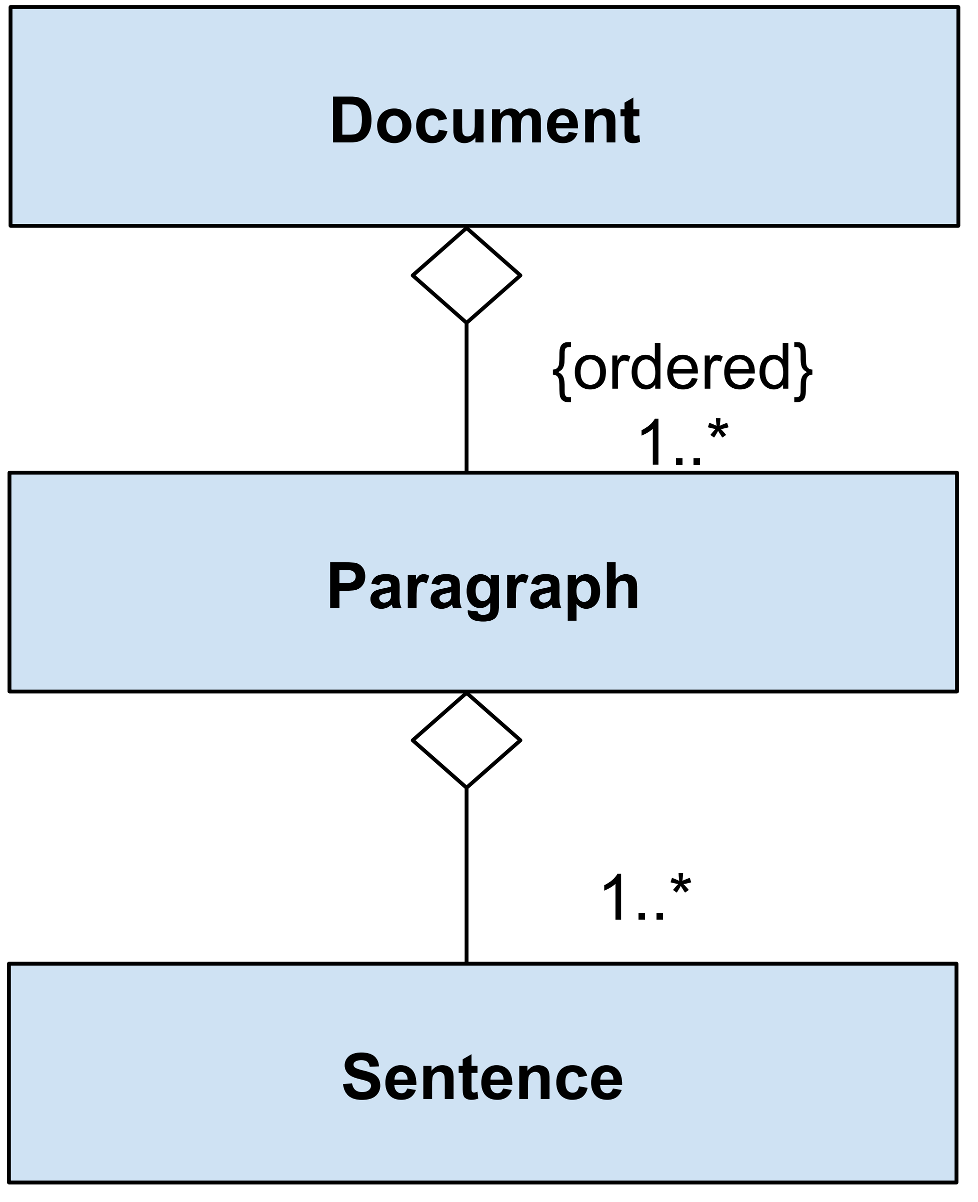
Aggregation

- A whole/part association in which the part object may belong to more than one whole object

- The whole object is called the aggregate object

- The part object is called the constituent object

Aggregation- contd



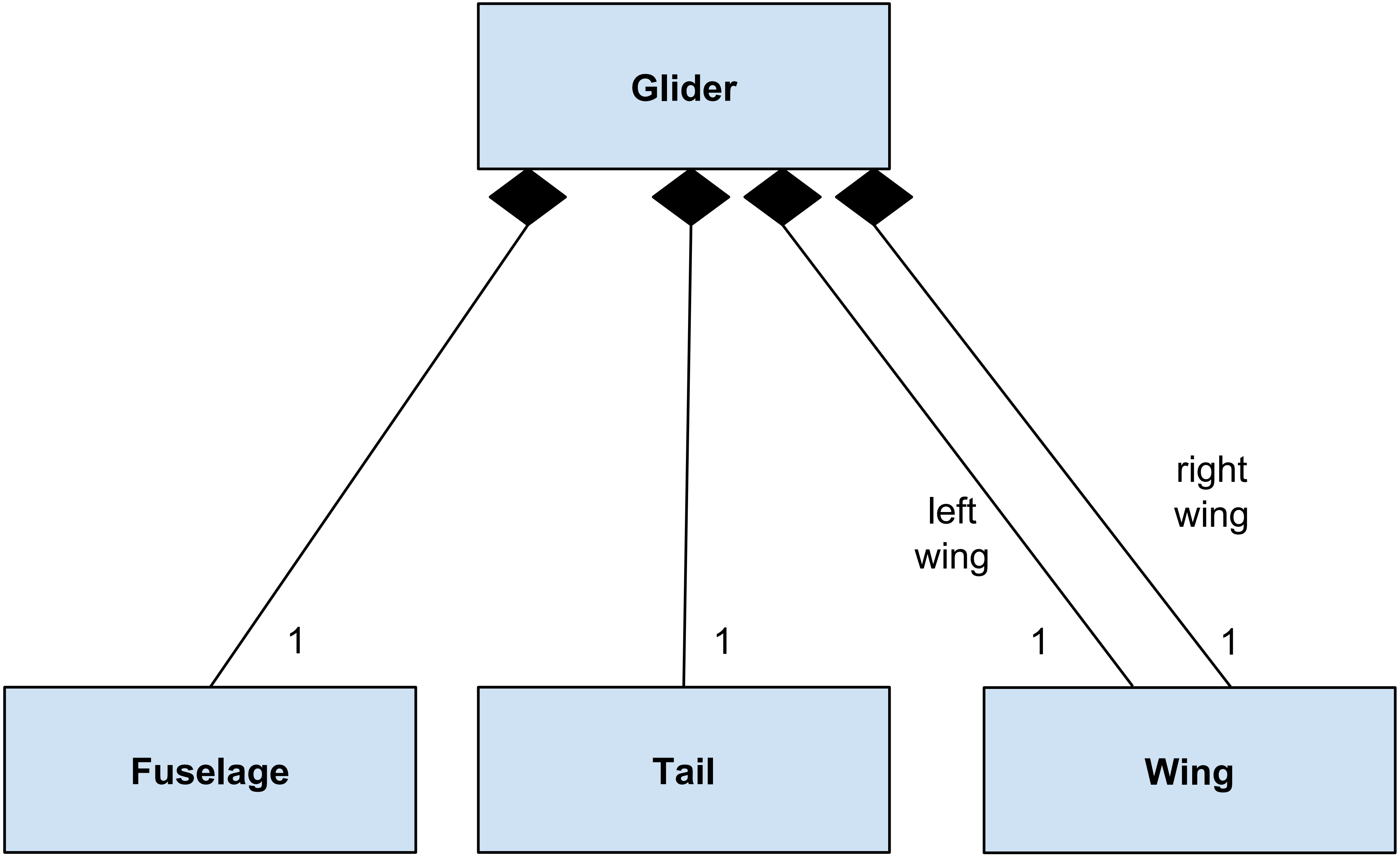
Composition

- A whole/part association in which the part object can belong to only one whole object. Moreover the parts are expected to live and die with the whole

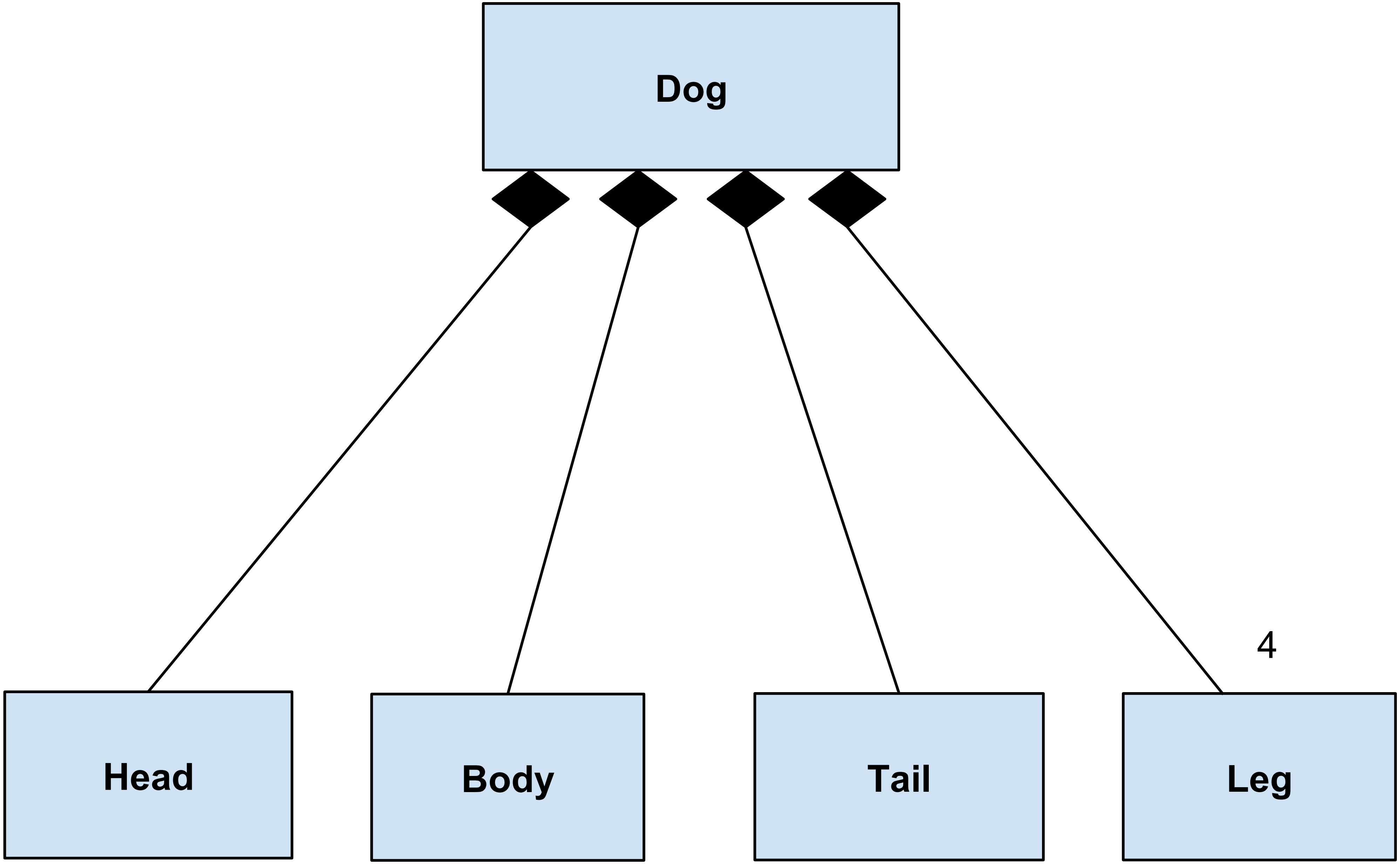
- The whole object is called the composite object

- The part object is called the component object

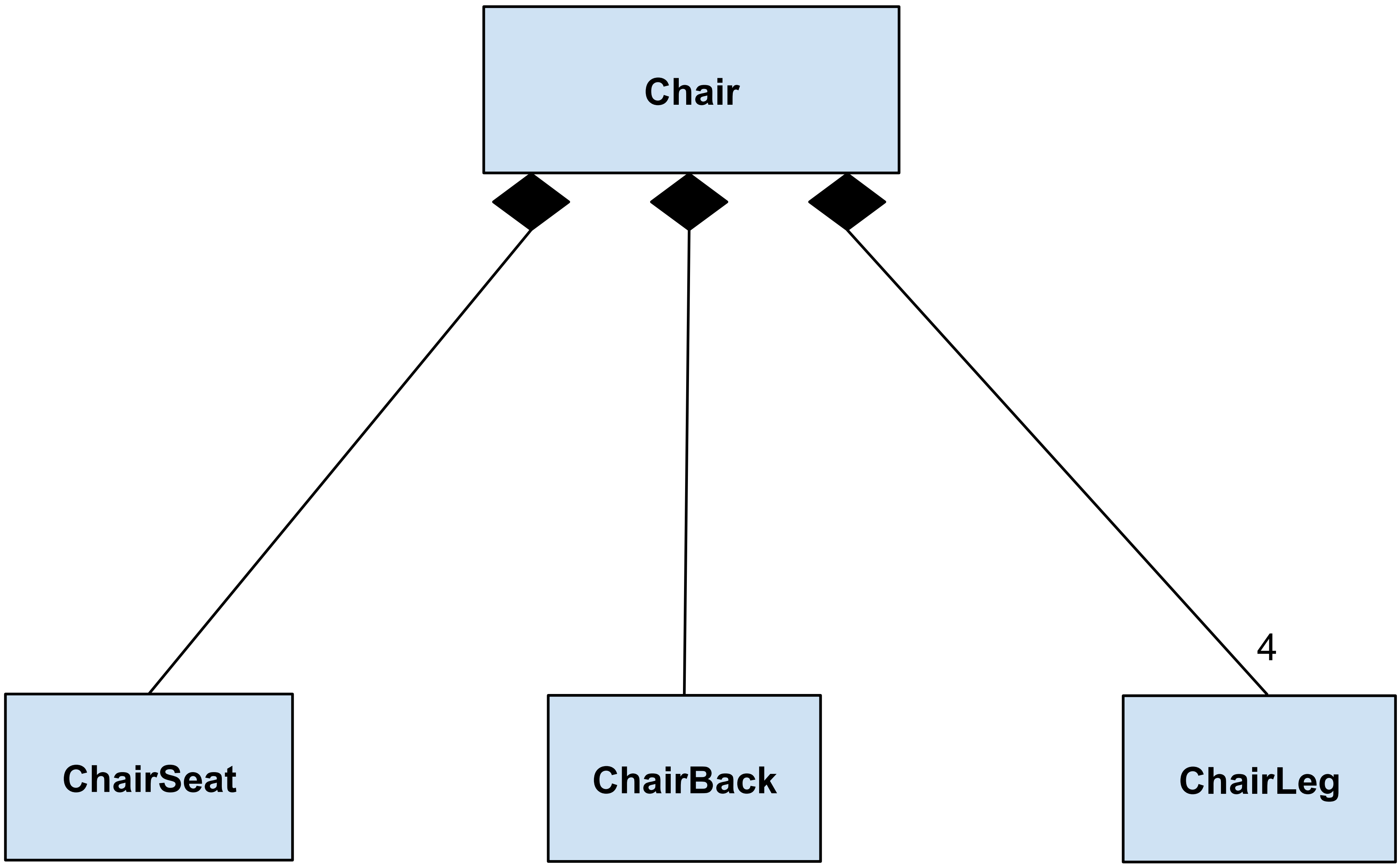
Composition-contd



Composition-contd



Composition-contd



Object Interaction Diagrams (Interaction Diagrams)

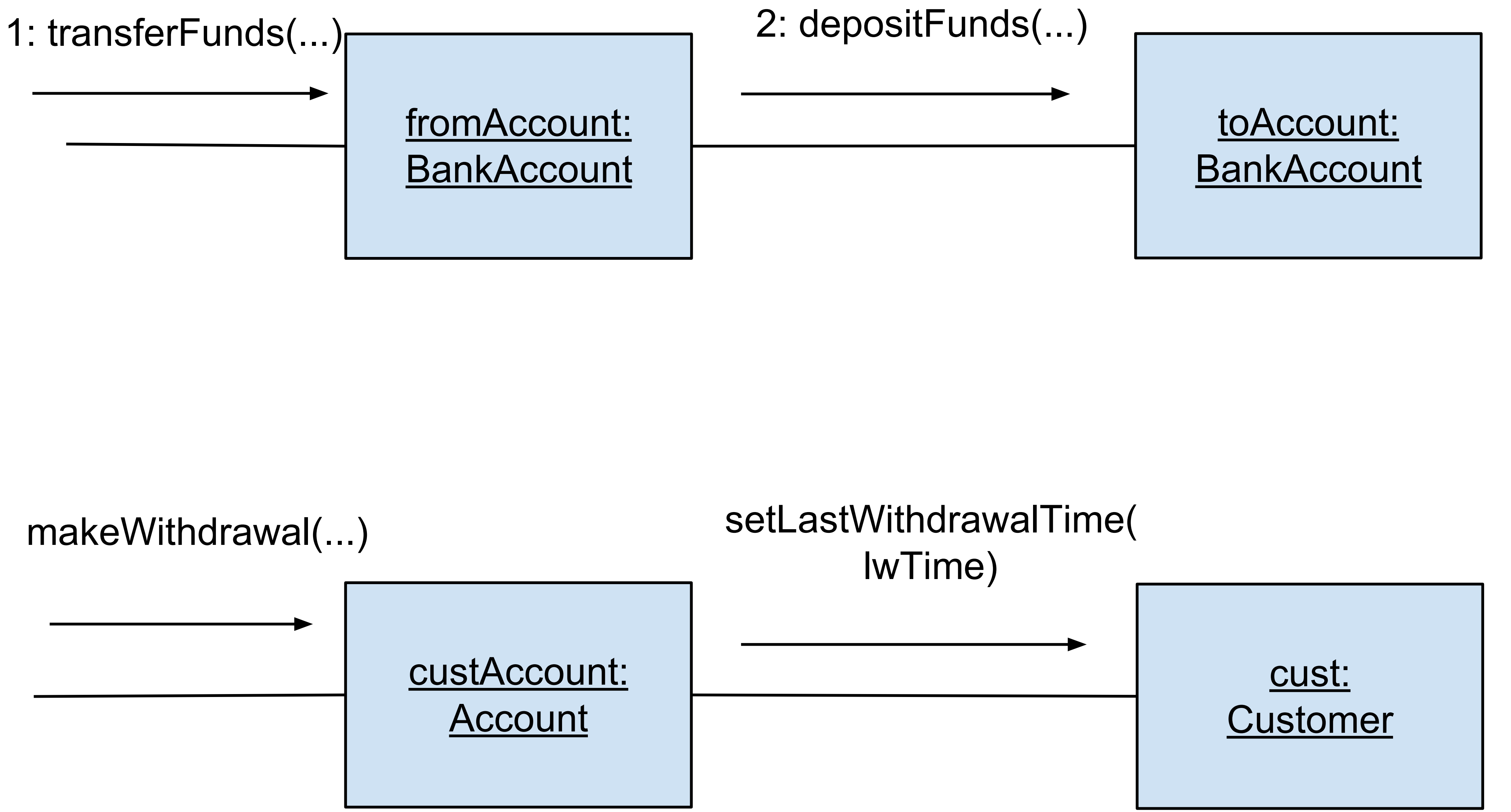
- They show how the objects interact with one another, dynamically, by sending messages.

- There are two types of object interaction diagrams

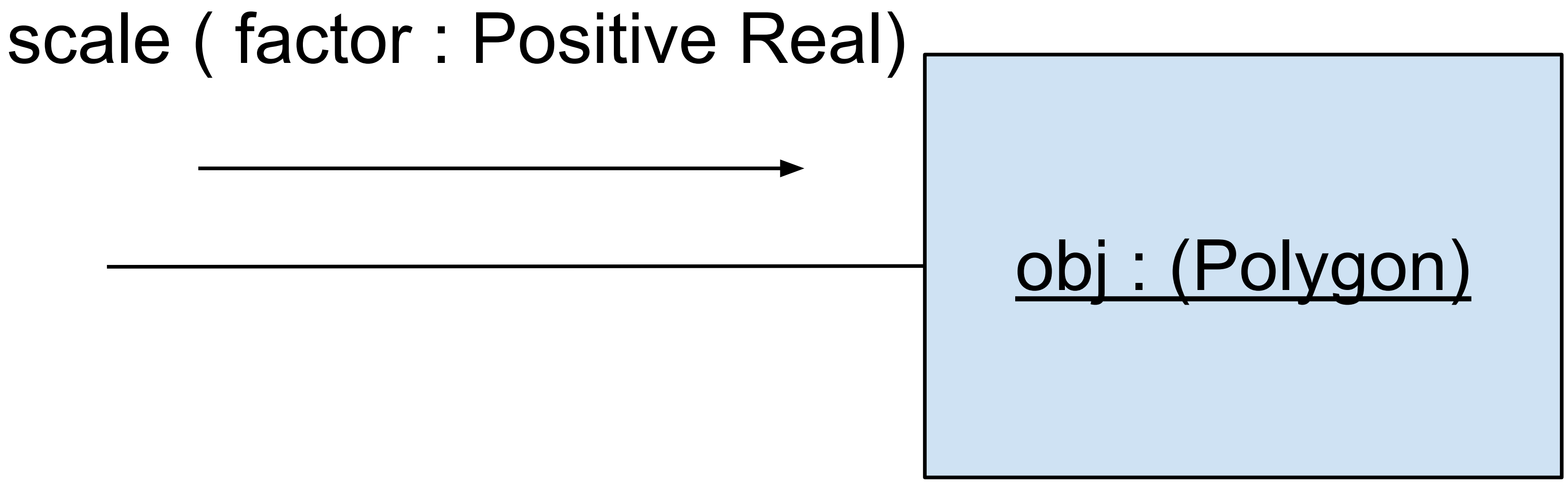
    - Collaboration Diagrams

    - Sequence Diagrams

Collaboration Diagrams



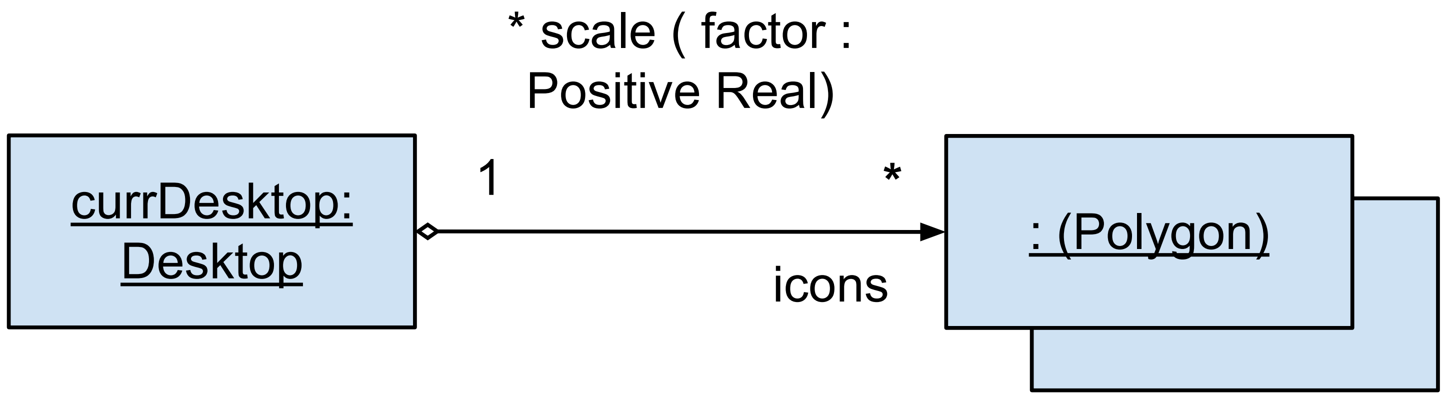
Collaboration Diagrams - contd



Run - Time Polymorphism in a Collaboration Diagram

Iterated Messages

- A message that is sent to each constituent of an aggregate object.

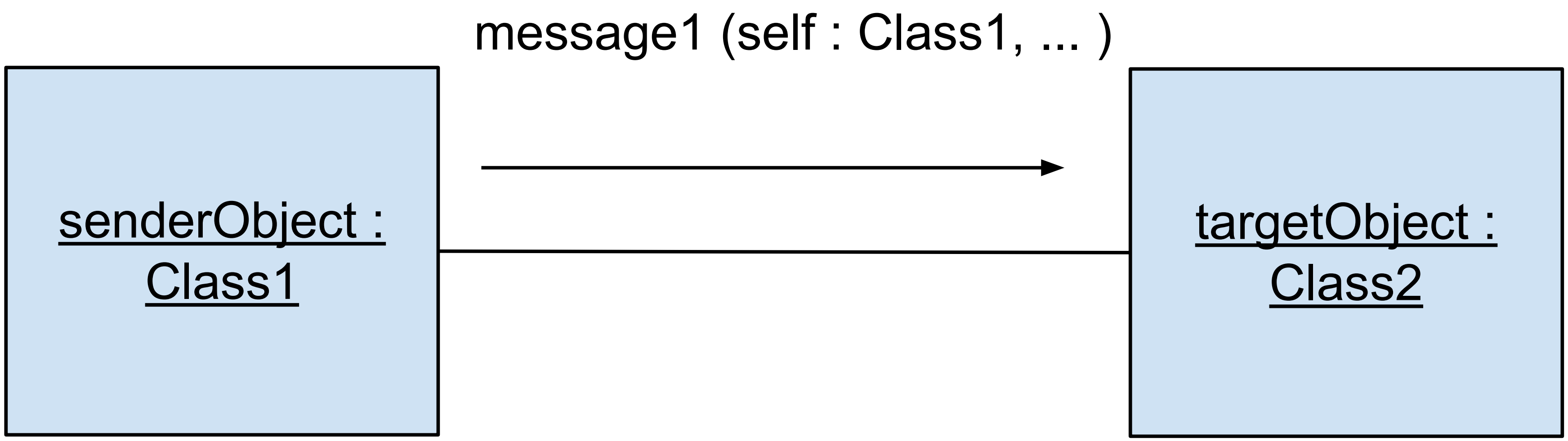


- The message is prefixed by an asterisk(*) , signifying multiplicity
- Each individual target object remains unnamed
- The target object symbol is doubled to signify multiplicity

Use of Self in Messages

It is used to

- Pass itself as an argument thereby telling the target object, which object sent the message

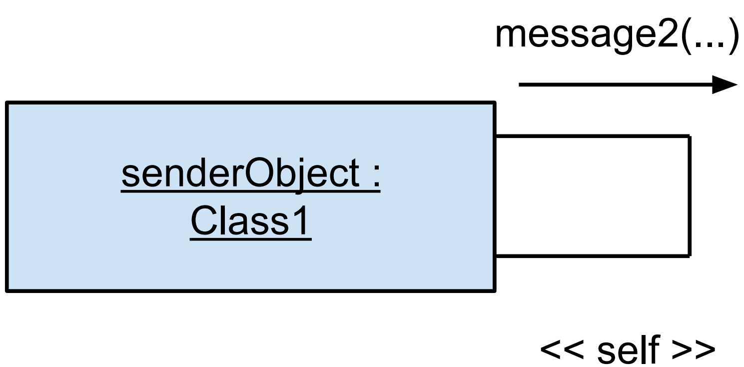


Use of Self in Messages -contd

It is also used to

- send a message to itself

form 1



form 2

Sequence Diagram

- It clearly specifies the sequence in which various messages are sent in an object oriented system

- It is preferred to a collaboration diagram when there are several objects and several message transfers in an object oriented system

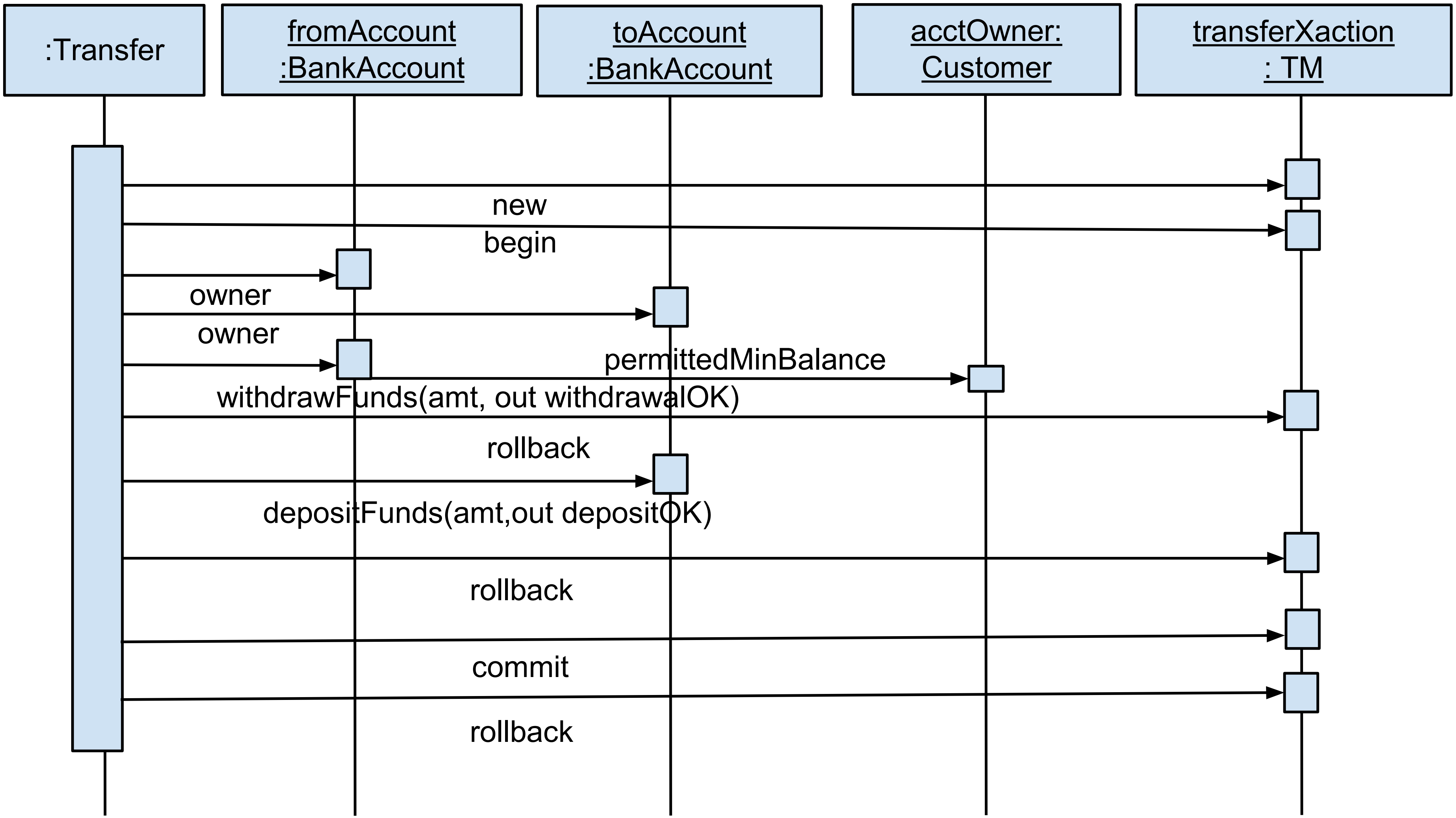
Pseudocde for the operation Transfer.makeTransfer : Boolean

```
create a new transfer transaction
begin transaction
establish fromAccount owner
establish toAccount owner
If the two owners are the same customer,
then
   fromAccount.withdrawFunds(amt, out
   withdrawalOK);
else
    transferXaction.rollback;
    return false;
endif
```

Pseudocde for the operation Transfer.makeTransfer : Boolean-contd

```
If withdrawalOK
then
   toAccount.depositFunds(amt, out depositOK);
else
   transferXaction.rollback;
   return false;
endif
If depositOK
then  transferXaction.commit;
         return true;
else transferXaction.rollback;
      return false;
endif
```

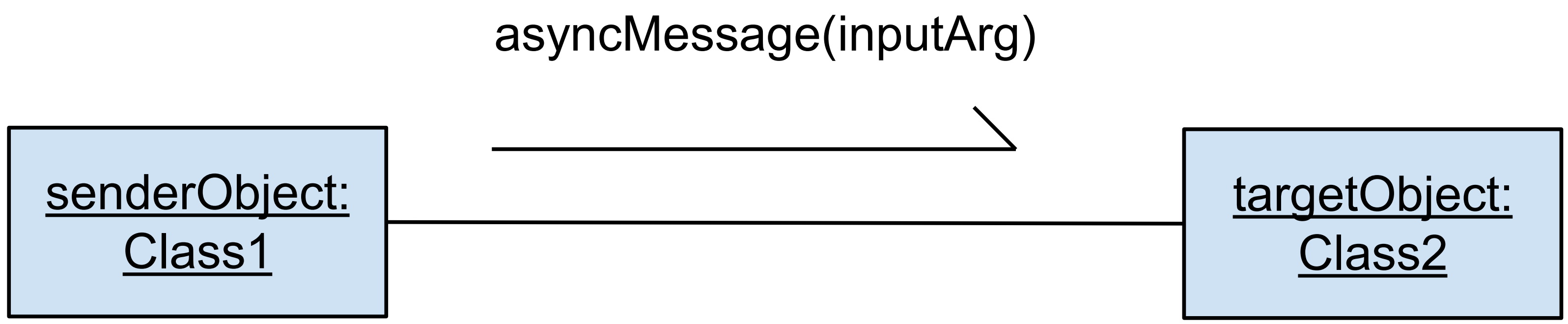
Sequence Diagram for the operation Transfer.makeTransfer : Boolean-contd



Asynchronous Messages and Concurrent Execution

- Synchronous Messages
  - Execution is single threaded
  - This means that only one object is active at a time
  - Only one object in a system can send a message at a given time
  - The sender object must wait for the target object to process the message
  - The target object will process only one message at a time

- Asynchronous Messages
  - Execution is multi threaded
  - This means that several objects execute at the same time
  - They occur in real time systems



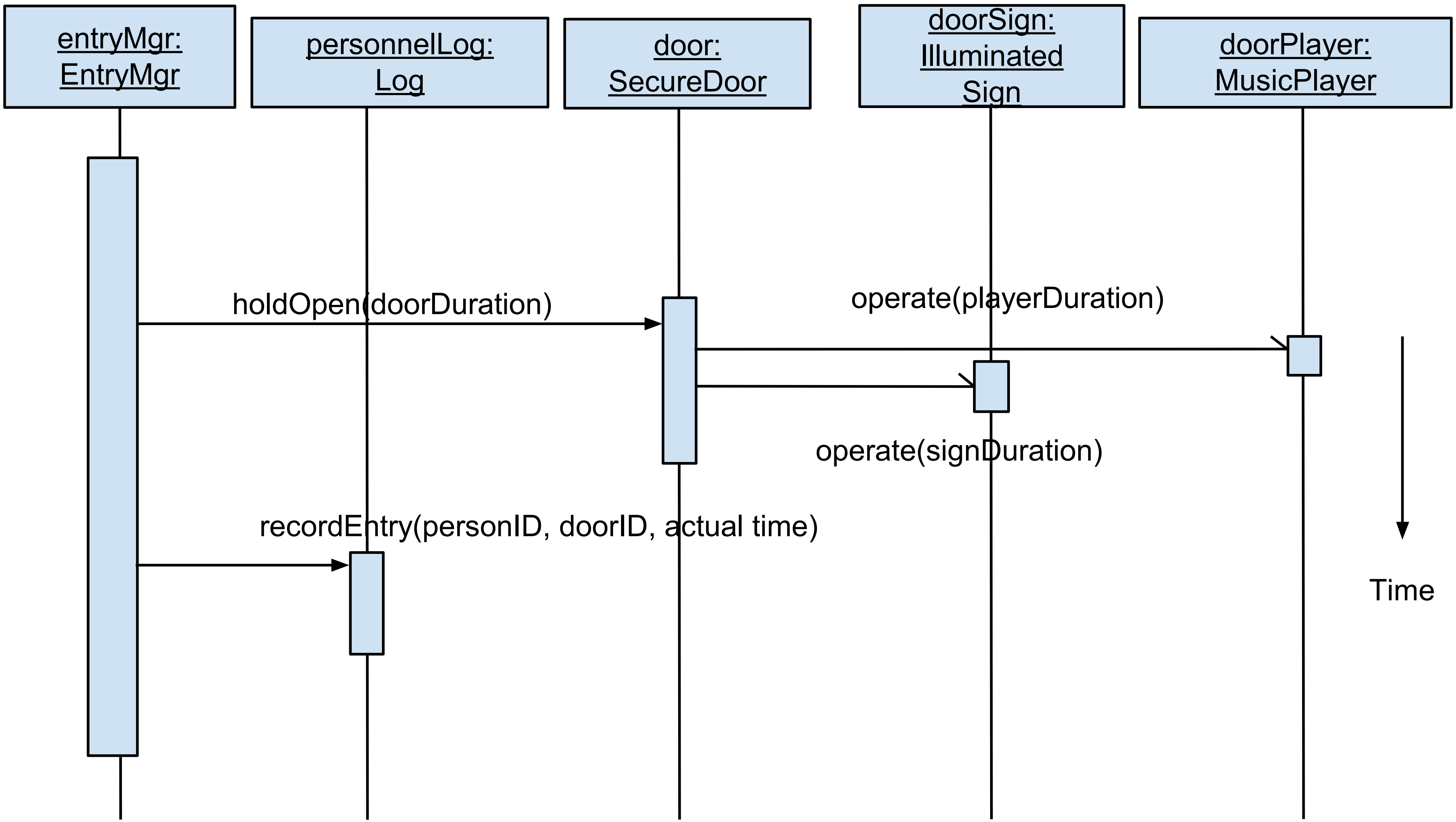
A RealTimeSystem containing asynchronous messages

- This system authorises employees to pass through electronically controlled doors

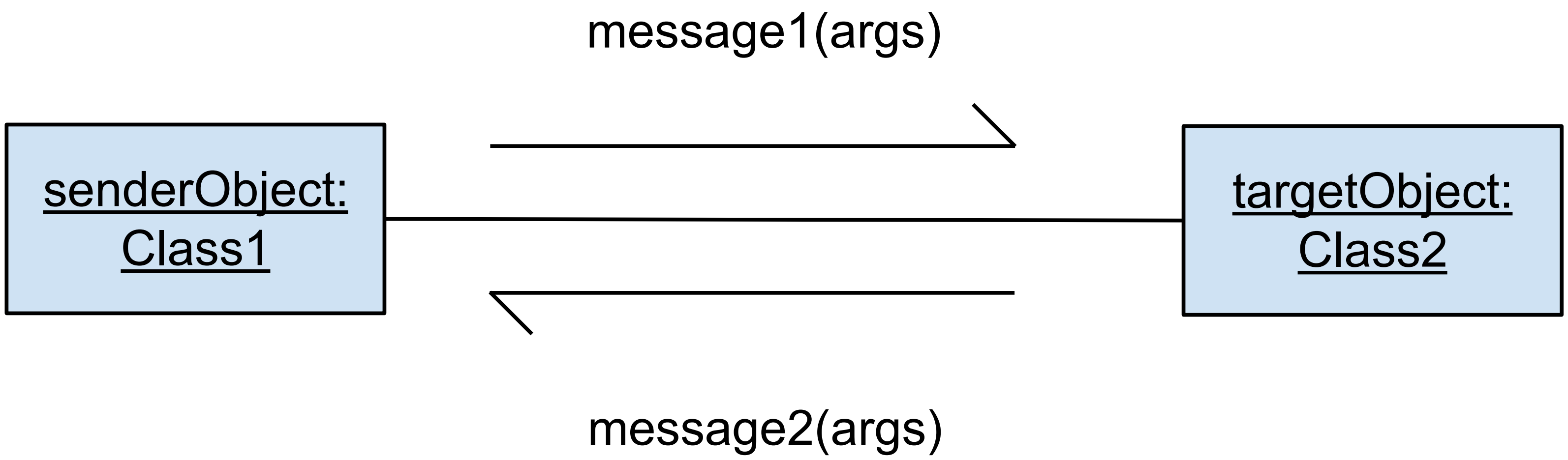
- The employee inserts an id card into a reader

- If the employee is authorised to enter, the system plays music, displays a greeting message and slides the door open

Sequence Diagram for the above Real Time System

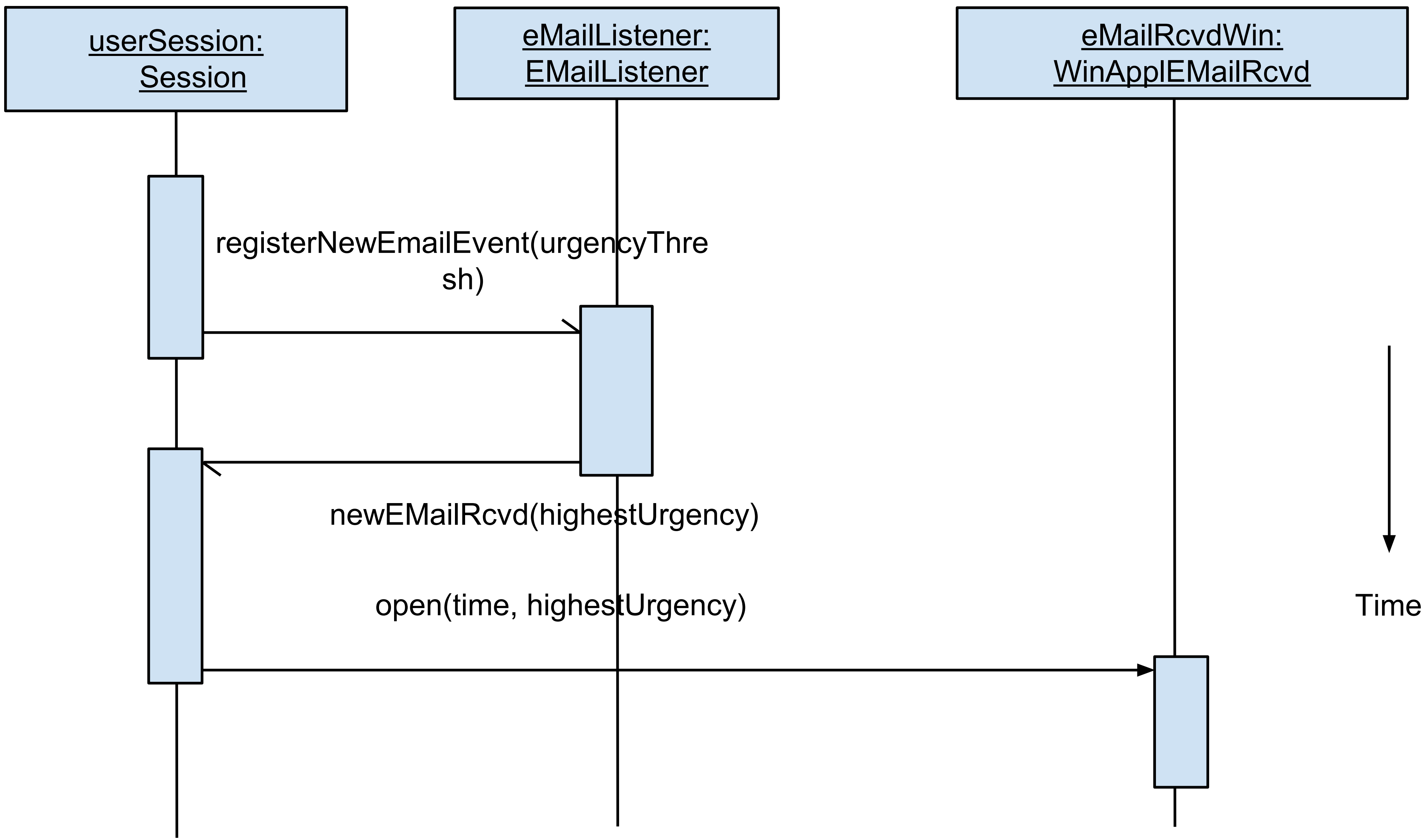


The Callback Mechanism

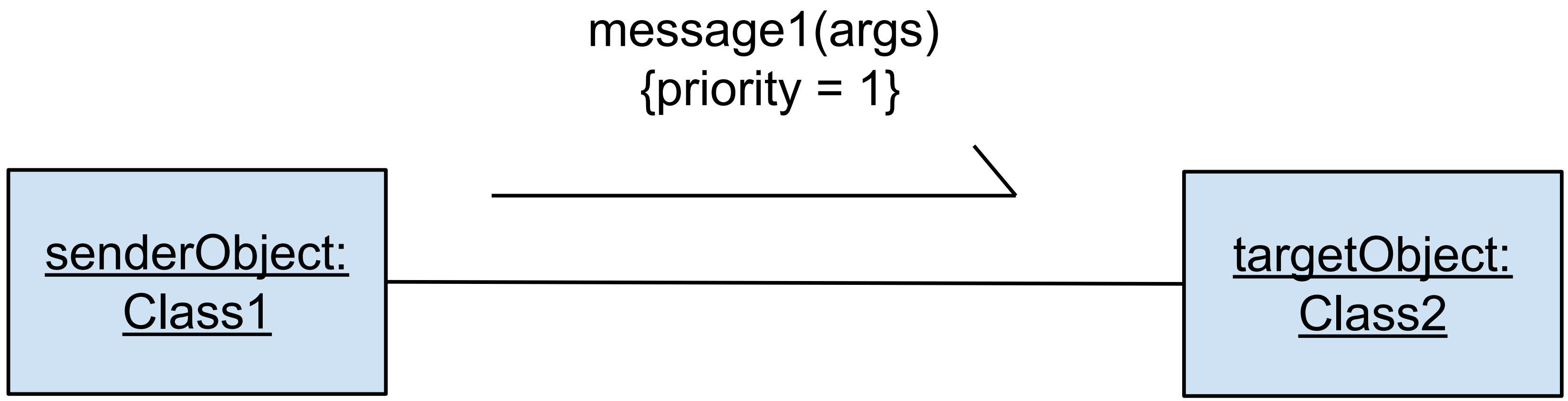


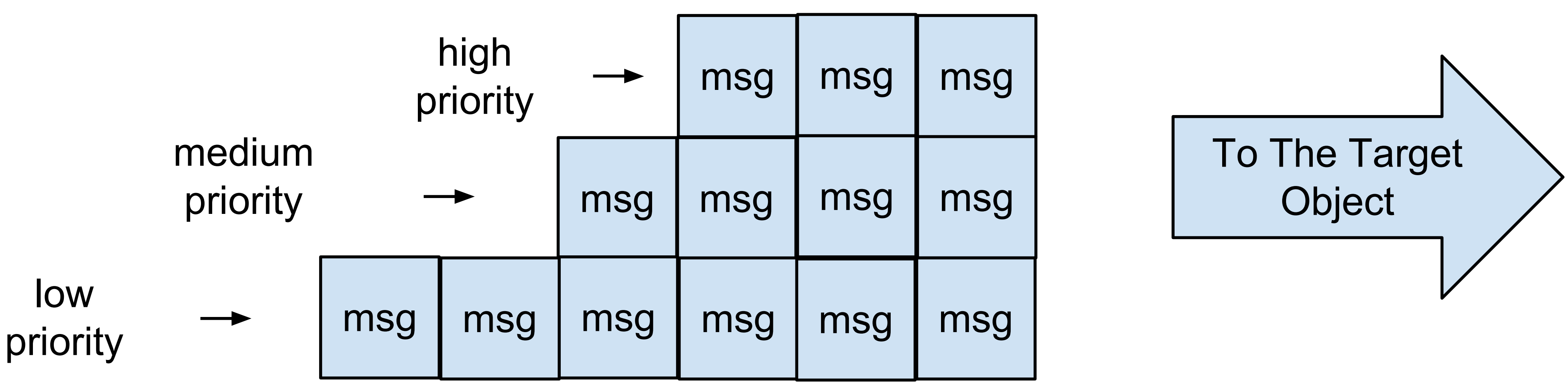
- An example for callback mechanism
  - A program which informs a user about urgent emails

Sequence Diagram for the above Program



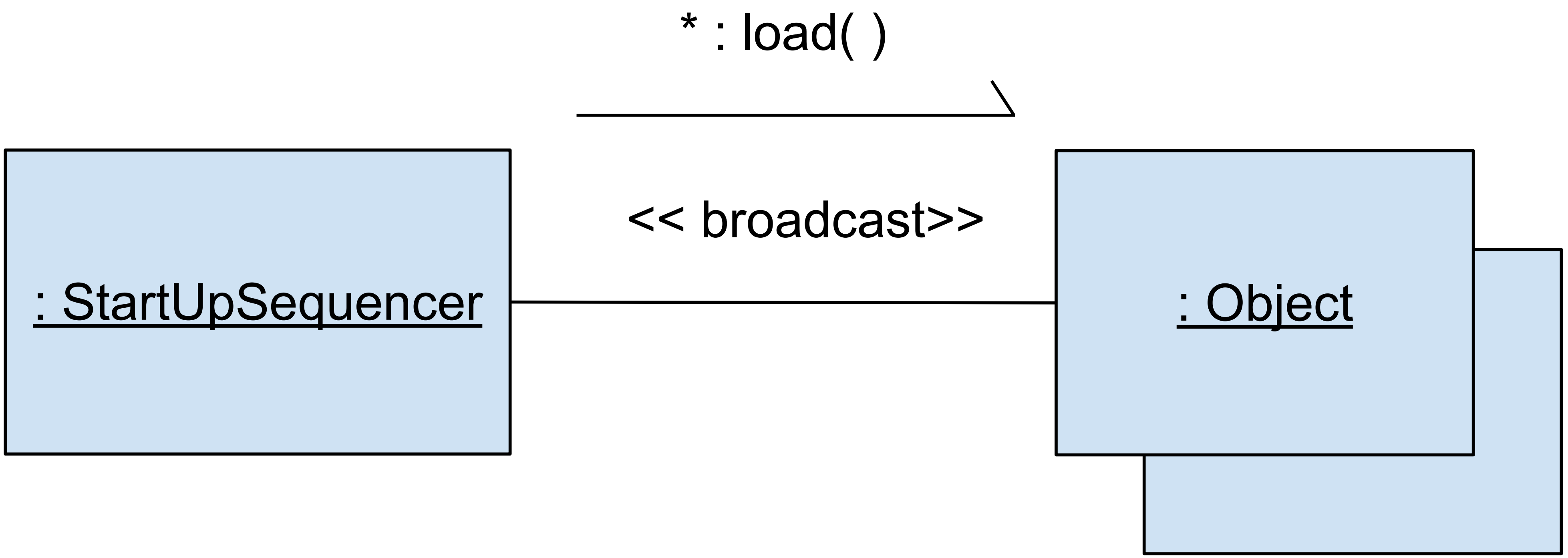
Asynchronous Messages with Priority





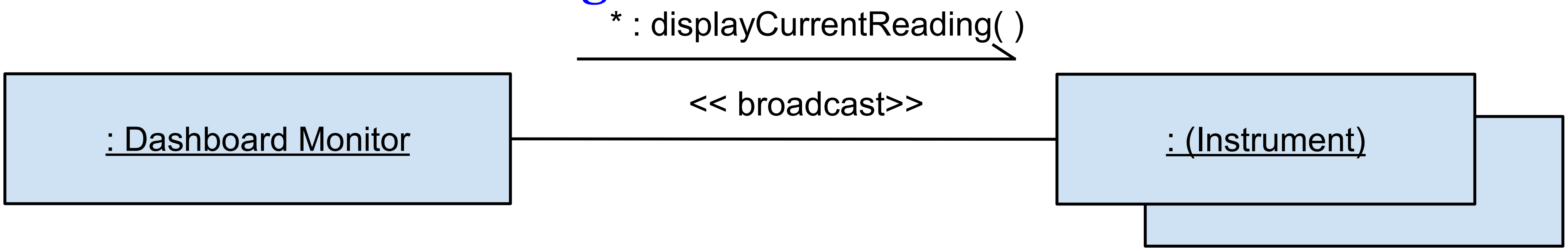
Depicting a Broadcast Message

- Broadcasting is the process of sending a message to every object in the system

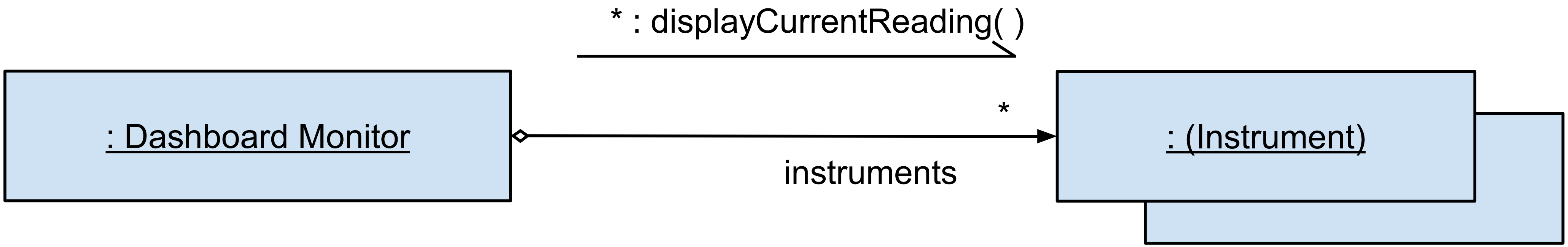


Depicting a Broadcast Message - contd

- Selective Broadcasting to some objects of the system is called Narrowcasting



narrowcast message



iterated message

State Diagrams(StateChart Diagrams)

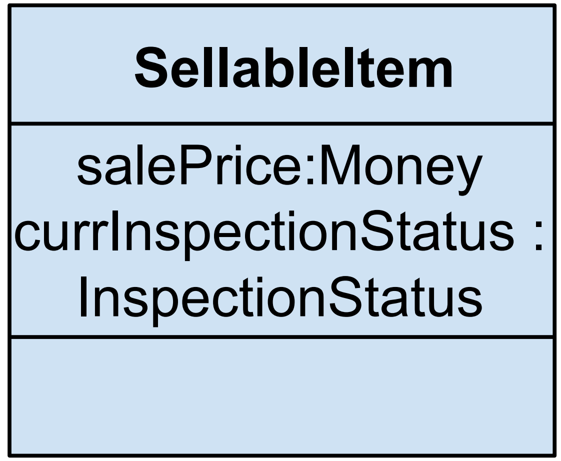
- A State Diagram for a class shows the states that objects of that class can assume and the transitions the objects may take from state to state

- A State Diagram is ideal for modelling an attribute with these two characteristics

    - The attribute possesses few values

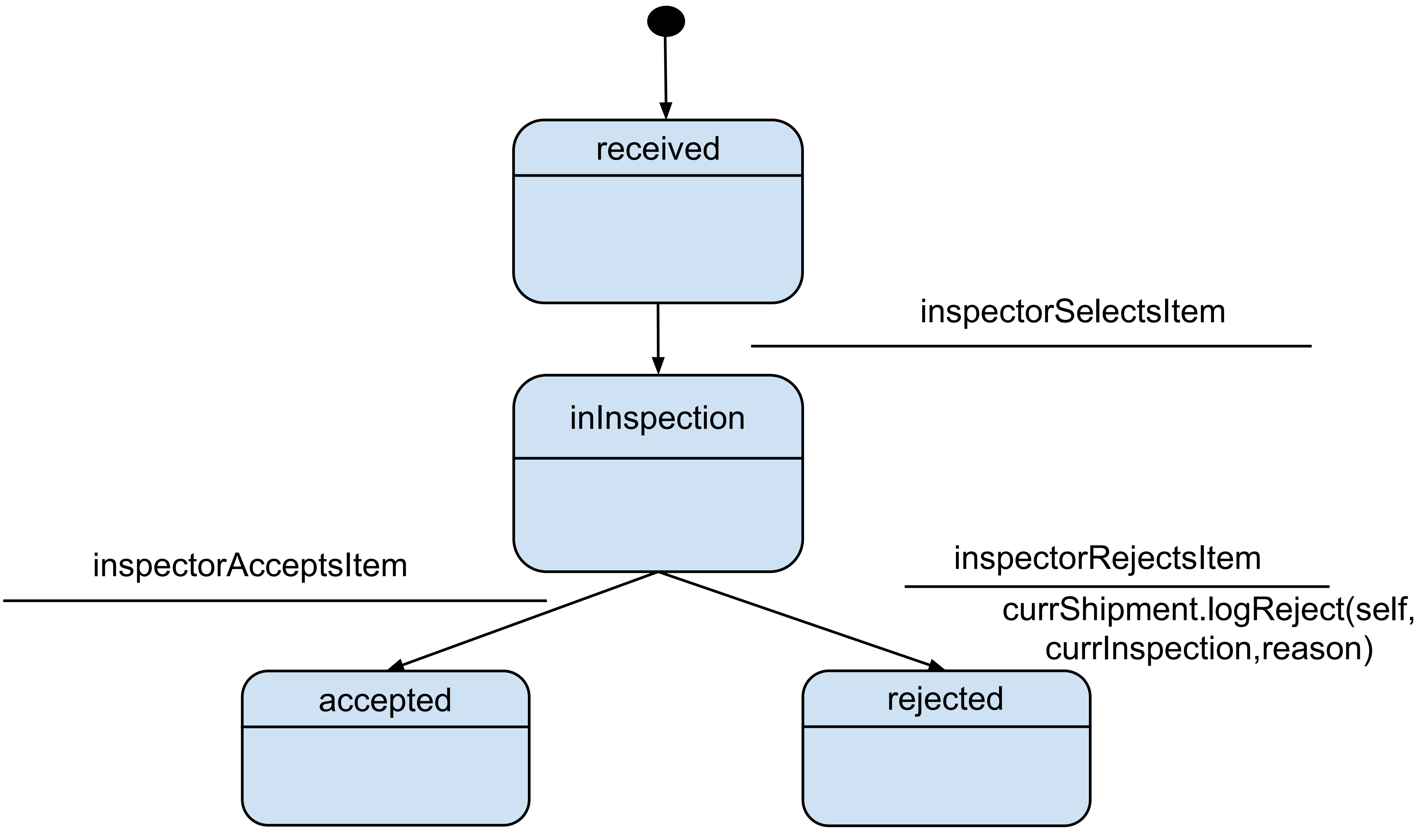
    - The attribute has restrictions on permitted transitions among those values

State Diagrams - contd

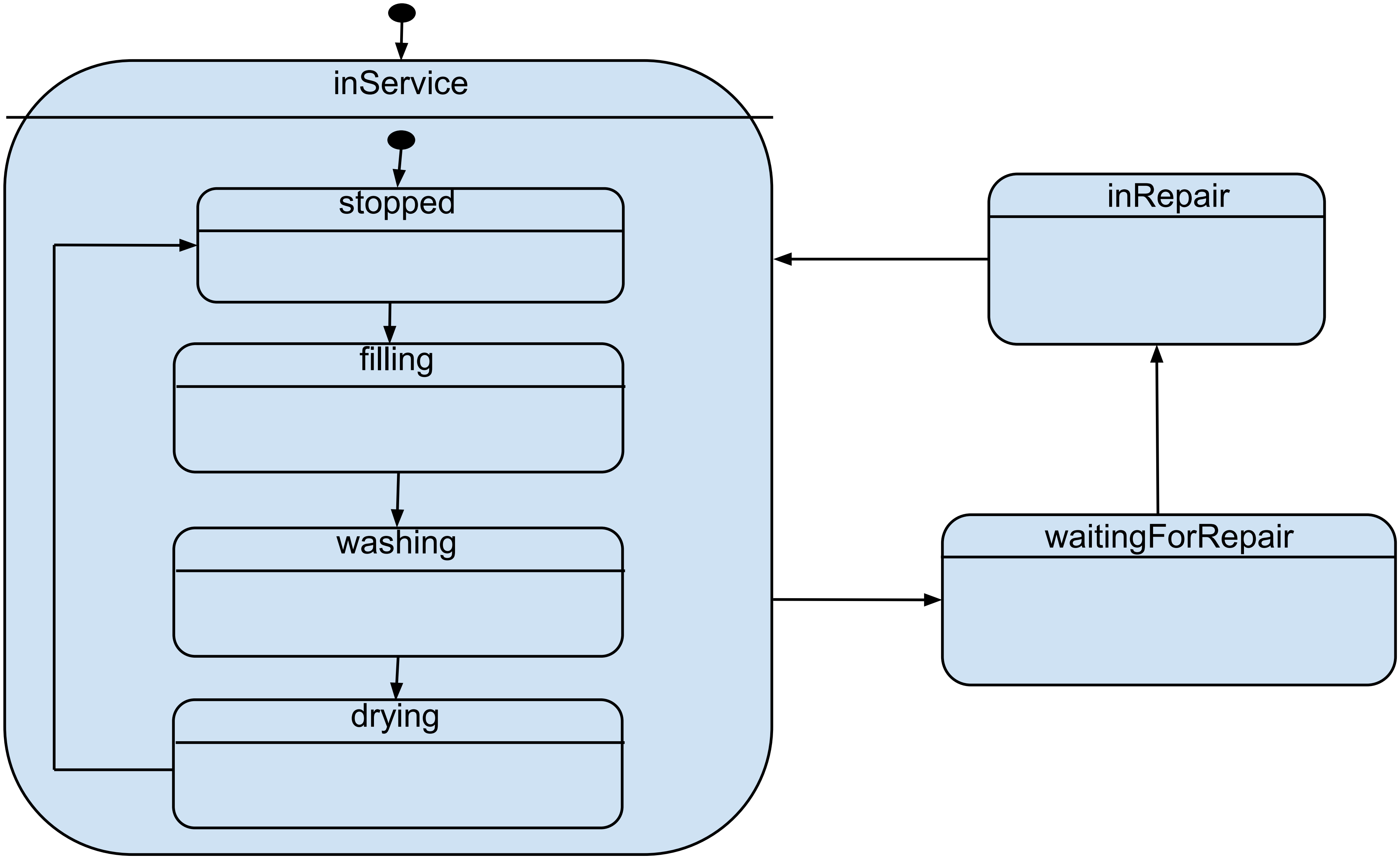


- SalePrice
  - It can have many values
  - Also it changes with much restriction
- currInspectionStatus
  - It can have only a few values such as received, inInspection, accepted and rejected
  - Transitions take place based on certain conditions

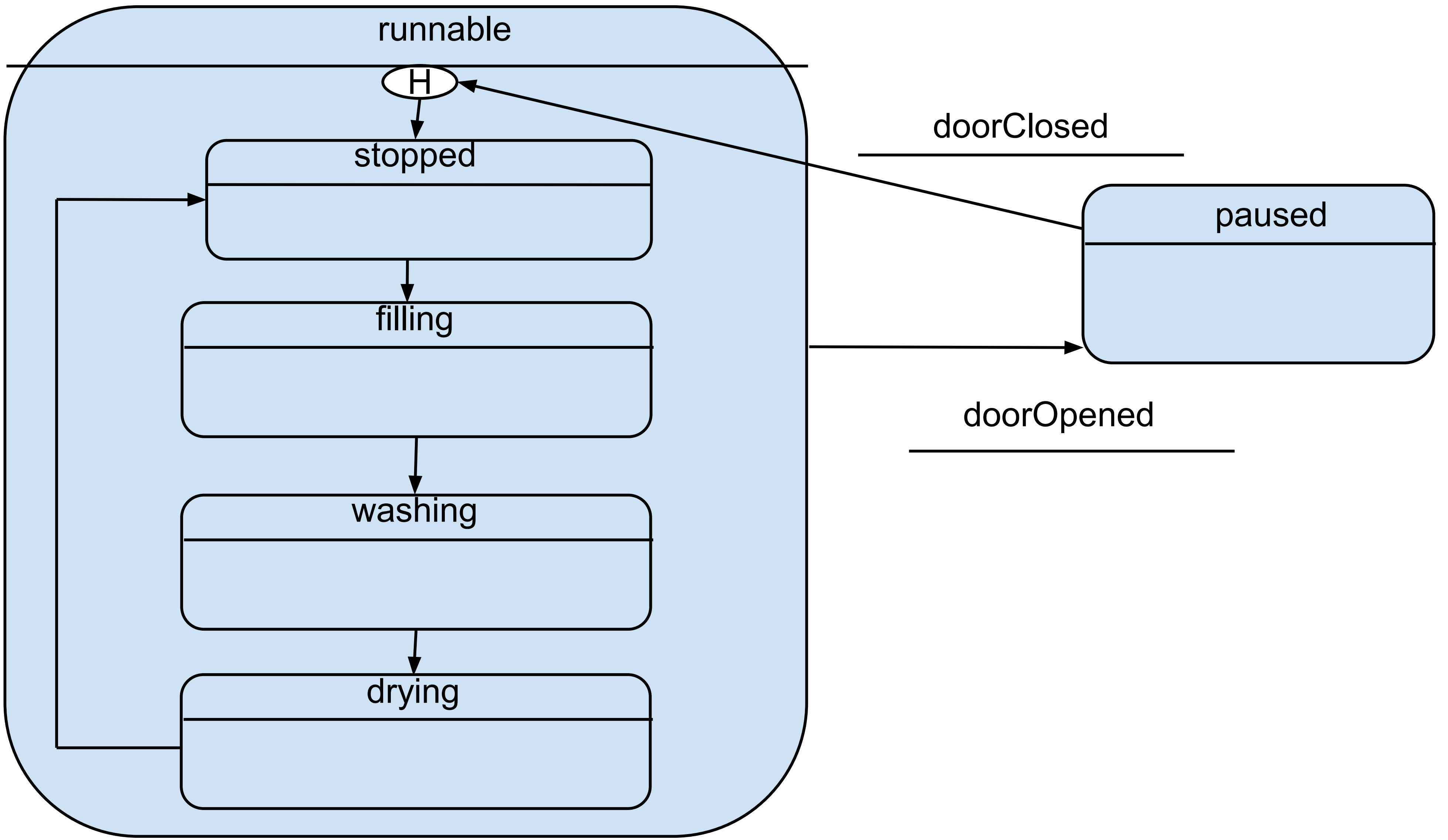
State Diagram - An Example



Nested States

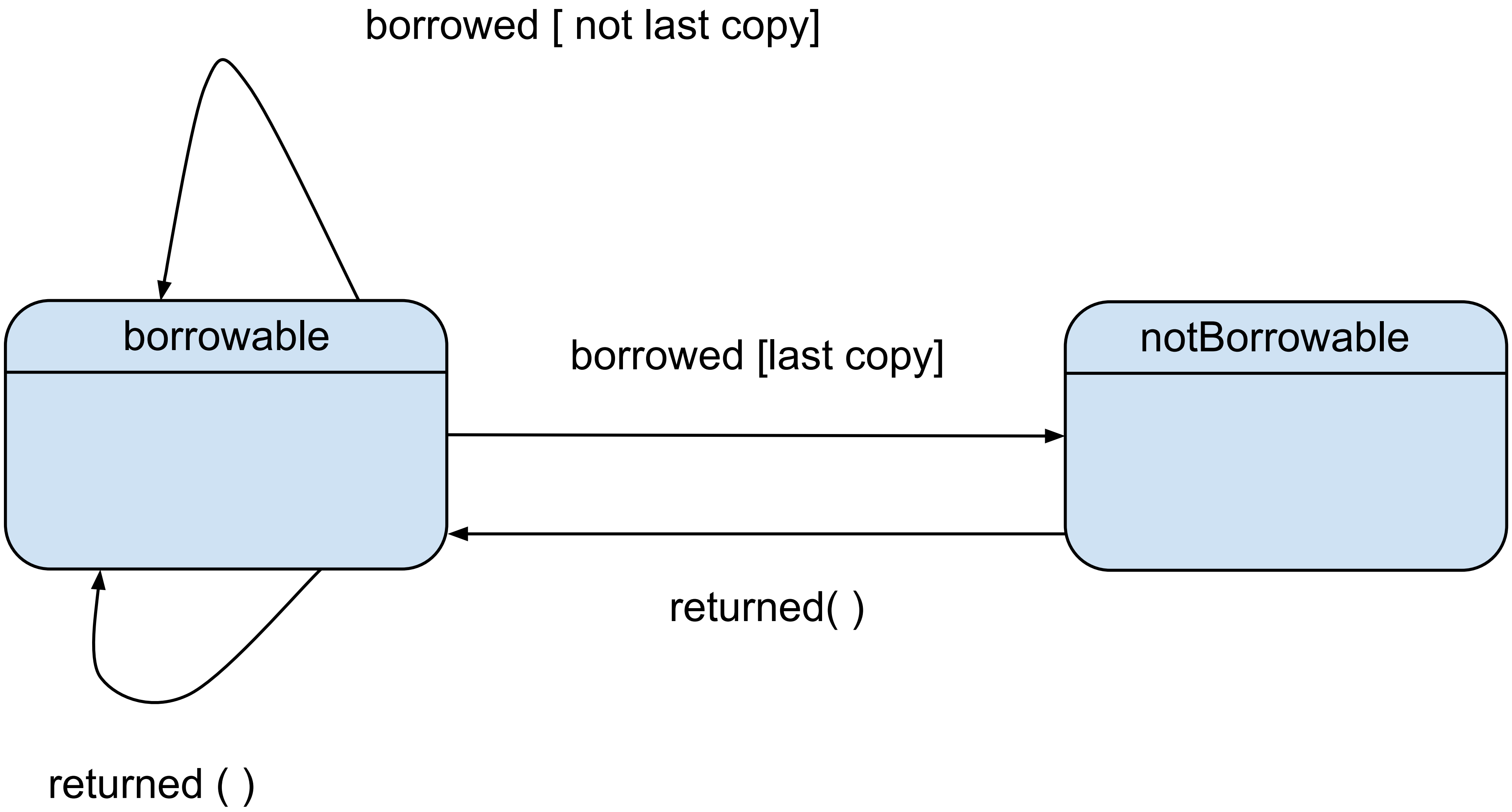


History Symbol in a state diagram

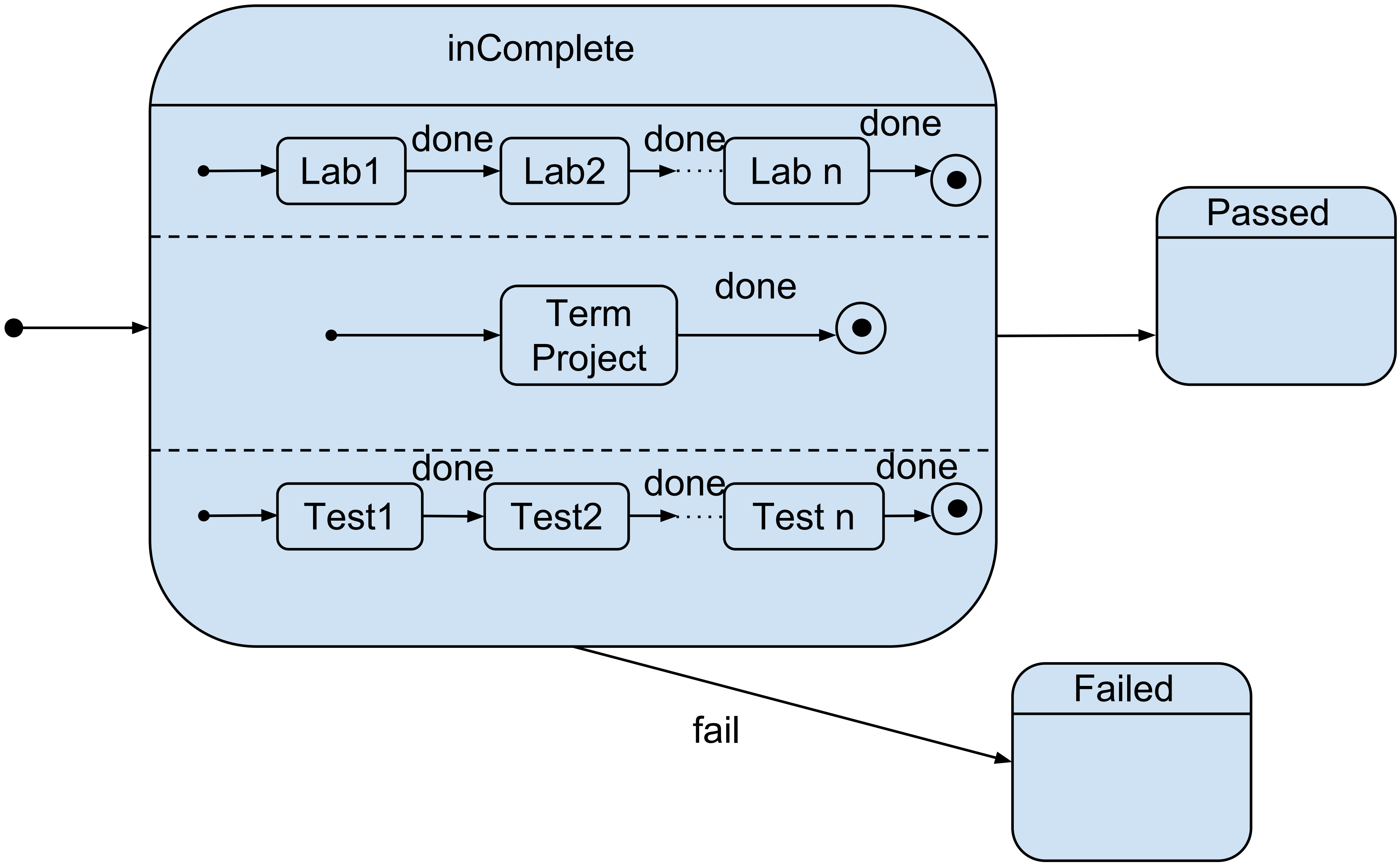


Guard in a state diagram

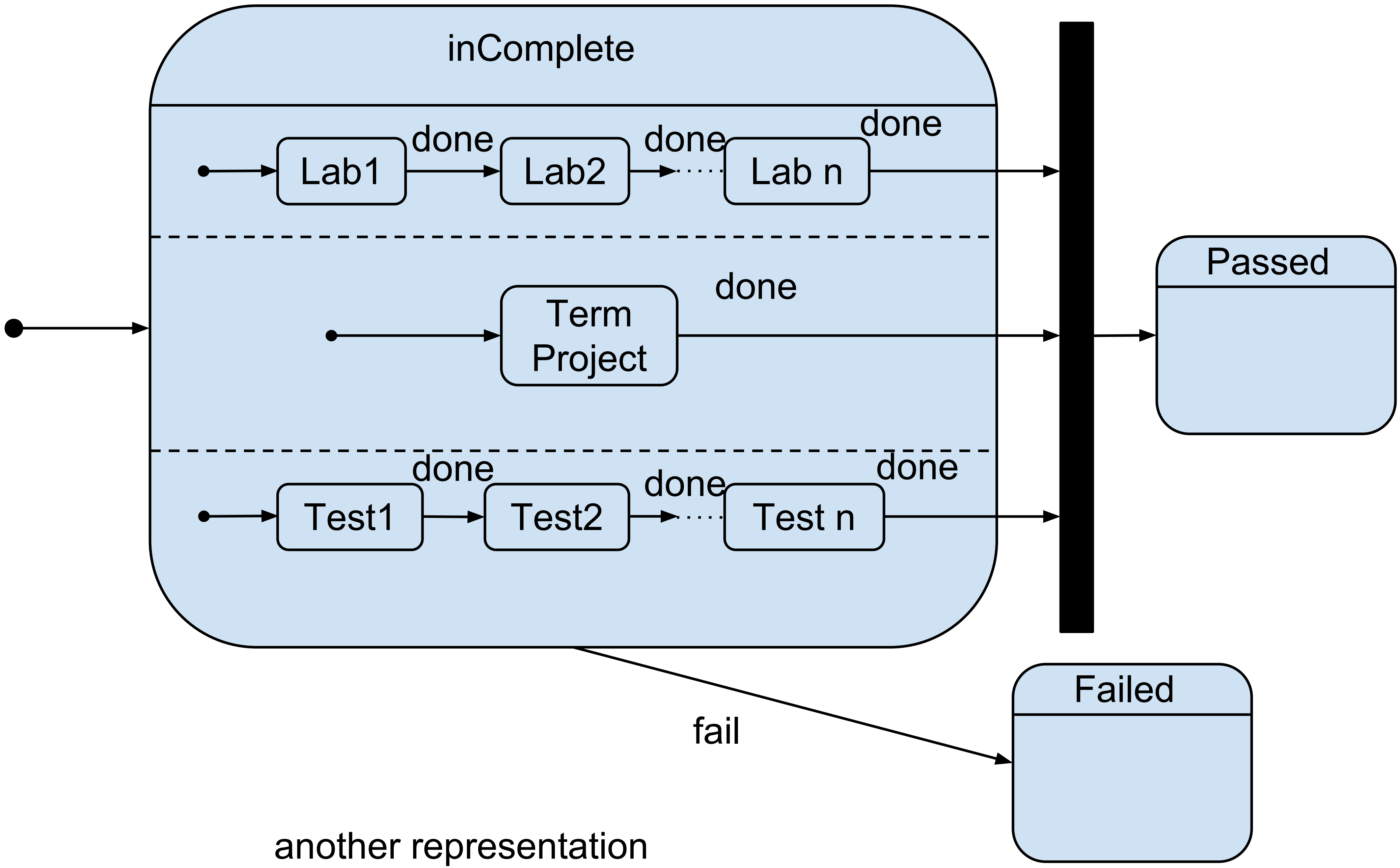
- It represents a condition in a state diagram that has to be satisfied for a transition to take place



Concurrent States and Synchronisation



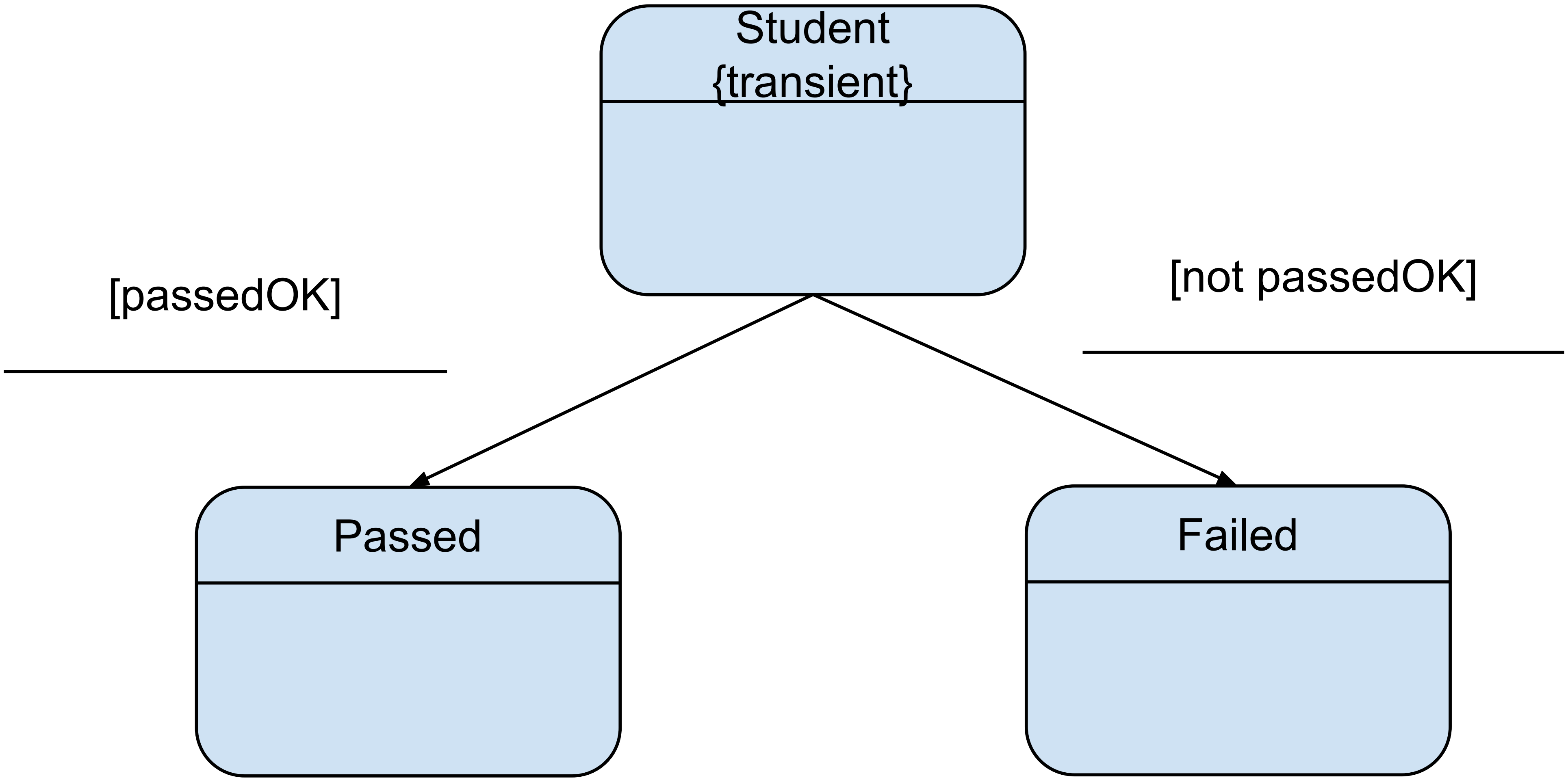
Concurrent States and Synchronisation- contd



another representation

Transient State

- A state in the UML diagram at which branching takes place based on a boolean guard

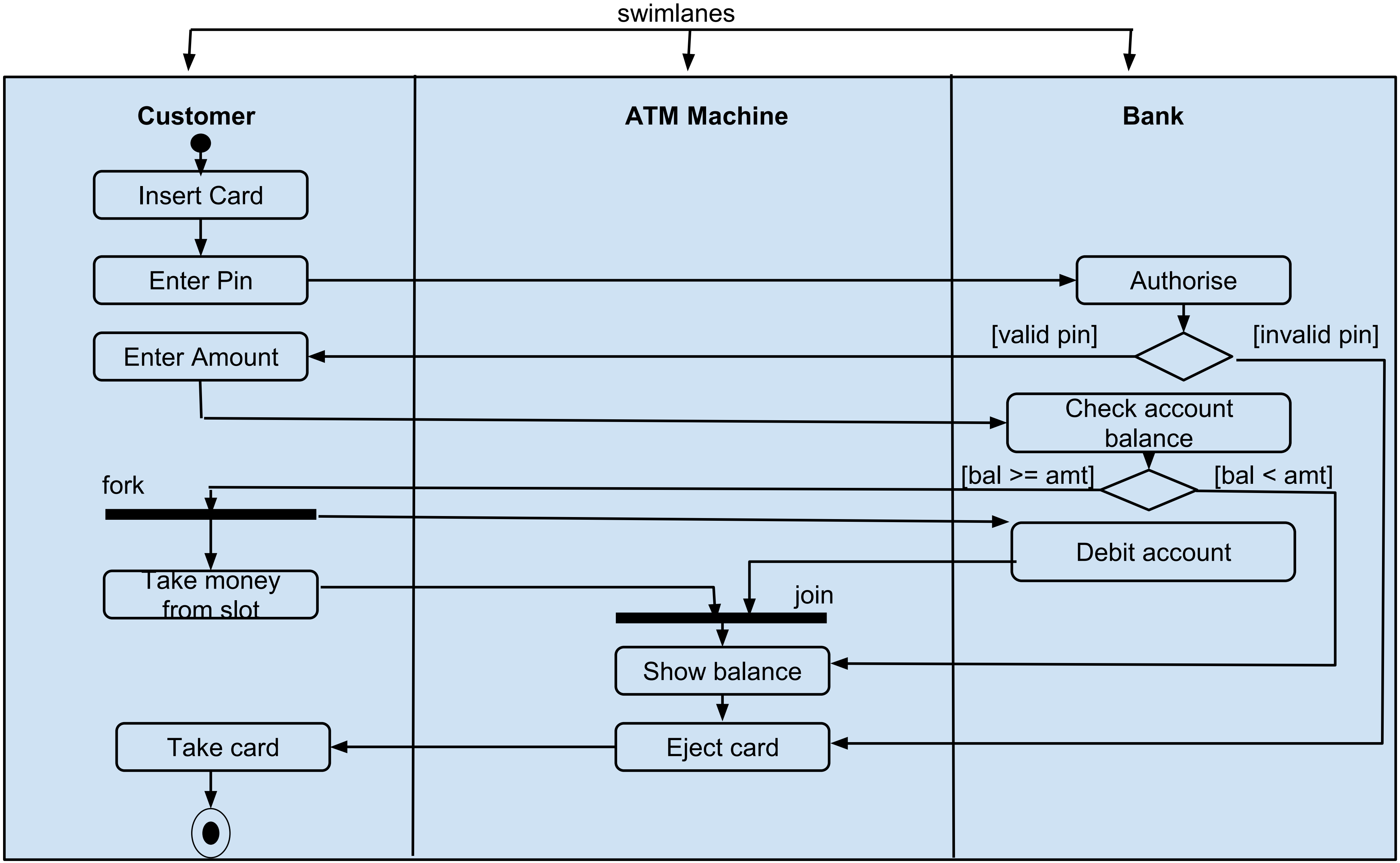


Activity Diagrams

- Illustrates the dynamic nature of a system by modelling the flow of control from activity to activity

- An activity represents an operation on some class that results in a change in the state of the system

Activity Diagrams



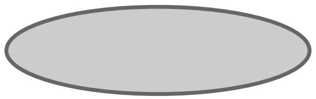
Use Case Diagrams

- A diagram that shows a set of use cases, actors and the relationships between them

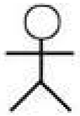
- Captures system functionality as seen by users

- Built in early stages of development

Elements of a Use Case Diagram

- Use Case 

A way of representing system functionality expected by the user

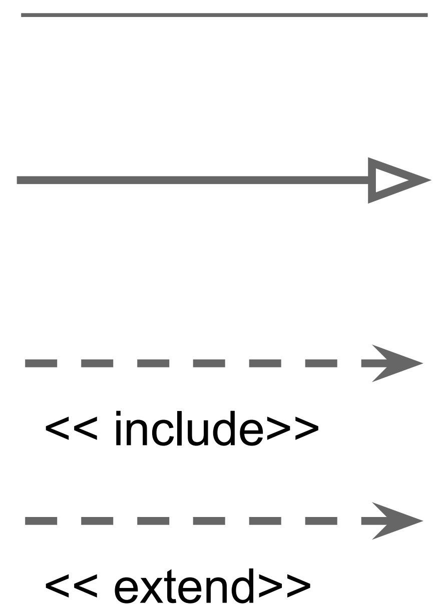
- Actor 

Each type of user is represented as an actor
They can be humans, computer system or an executable process

- Relationships
  - between actor and use case
  - between use case and use case
  - between actor and actor

Elements of a Use Case Diagram

- Types of Relationships
  - association
  - generalisation
  - include
  - extend



Elements of a Use Case Diagram

- association

  The communication path between an actor and a use case that it participates in

- generalisation

  A relationship between a general use case and a more specific use case that inherits and adds features to it

Elements of a Use Case Diagram

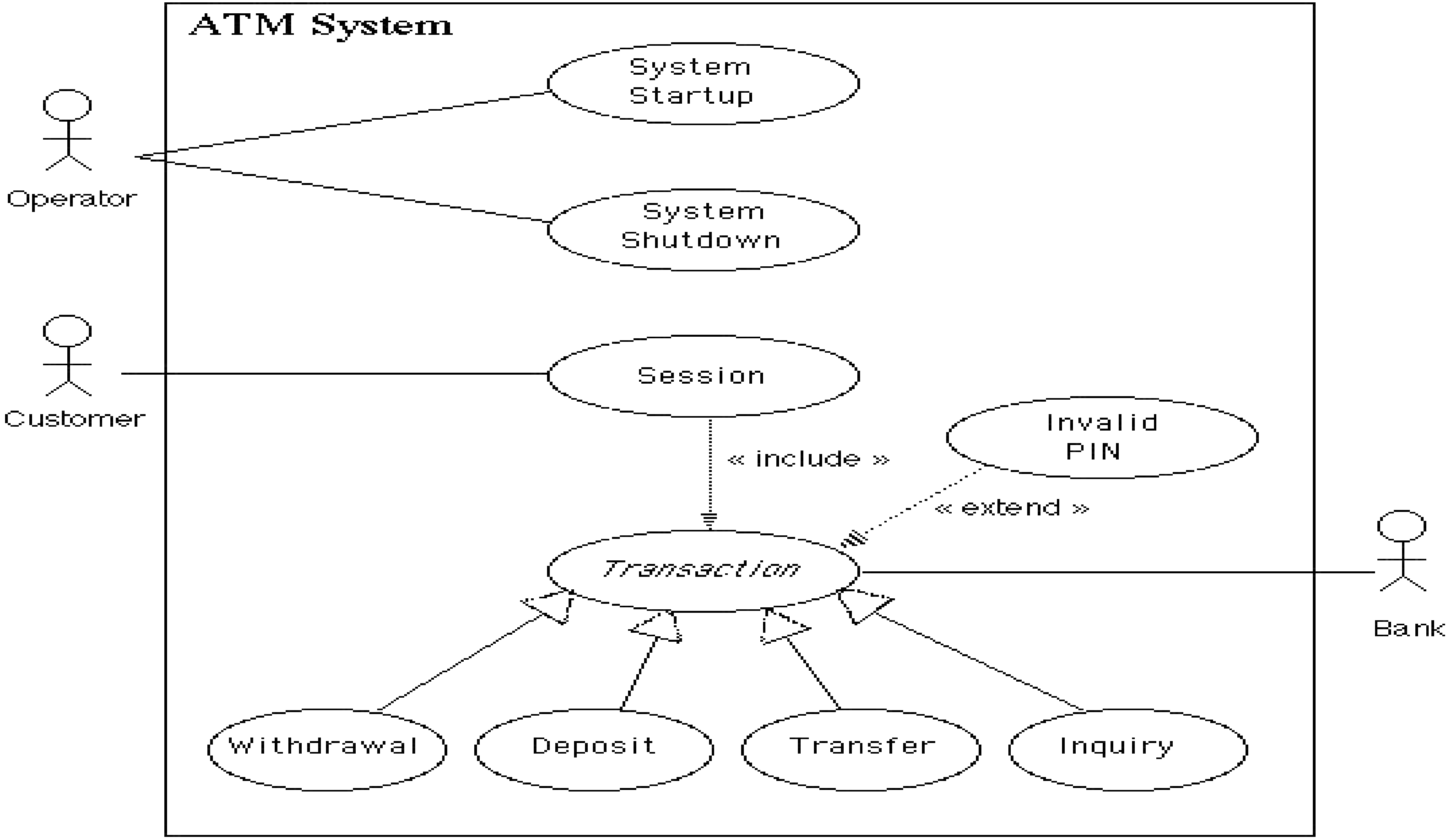
- extend

  The insertion of additional behaviour into a base use case that is executed only when the extending condition is true

- include

  The insertion of additional behaviour into a base use case that is executed unconditionally

Example - Use Case Diagram



Module III

- Architecture Diagrams
    - Package Diagrams
    - Deployment Diagrams
- Interface Diagrams
    - Window Layout Diagrams
    - Window Navigation Diagrams

Architecture Diagrams

- They are used for representing system architecture
  - Package Diagrams
    - They are used for depicting software architecture
  - Deployment Diagrams
    - They are used for depicting hardware architecture, and for describing the interaction between software architecture and hardware architecture

Package Diagram

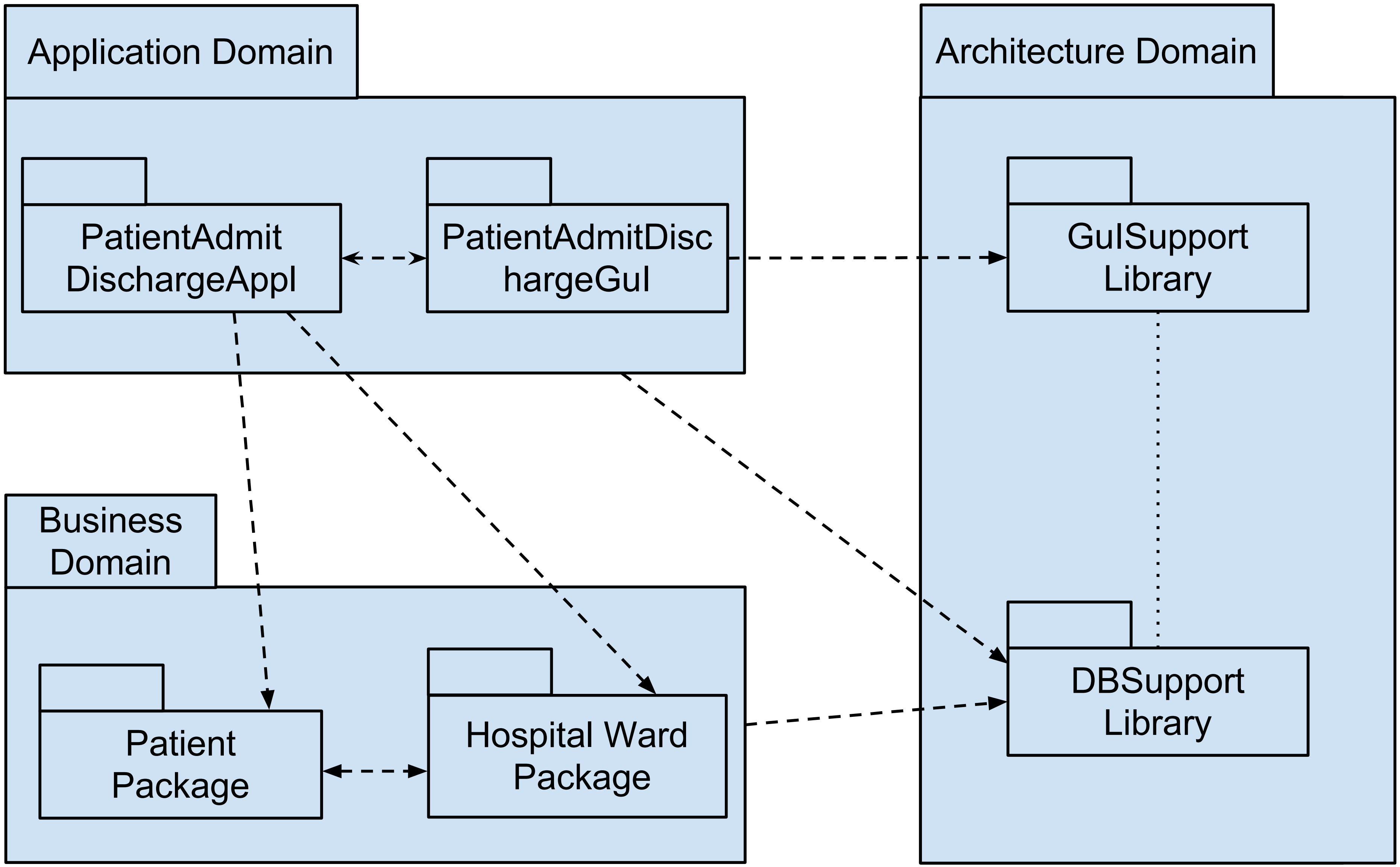
- It represents the various packages associated with the system and the different dependencies existing between them

A Package Diagram coming inside a Hospital Information System

Patient Package - Patient, Patient Medical History

Ward Package - Ward, Bed, Nurse

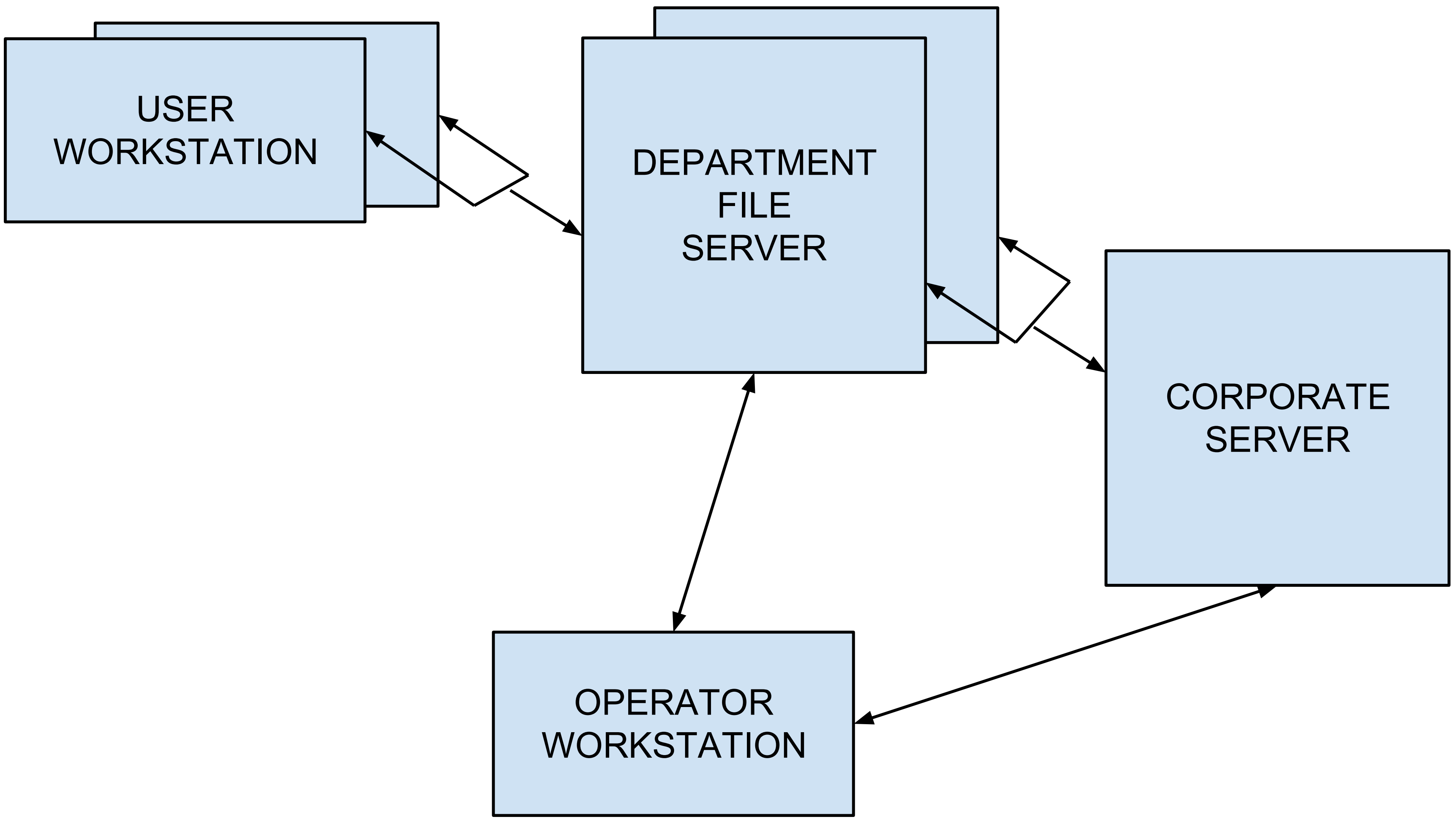
Package Diagram - contd



Detailed Package Diagram for the Hospital Information System

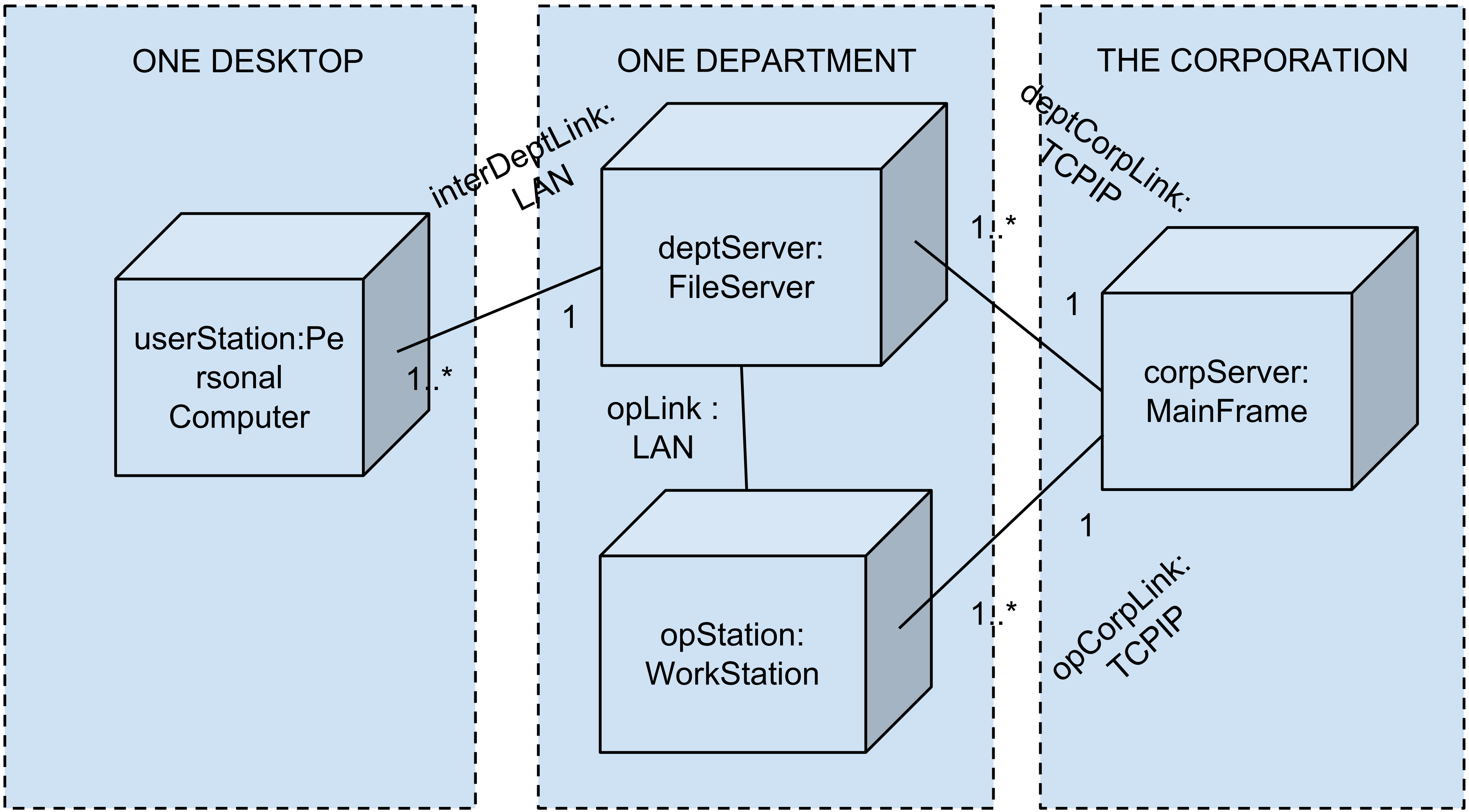
Deployment Diagrams for Hardware Artifacts

- Example - Client-Server Business System



Deployment Diagrams for Hardware Artifacts-contd

- Deployment Diagram-Client-ServerBusiness System System



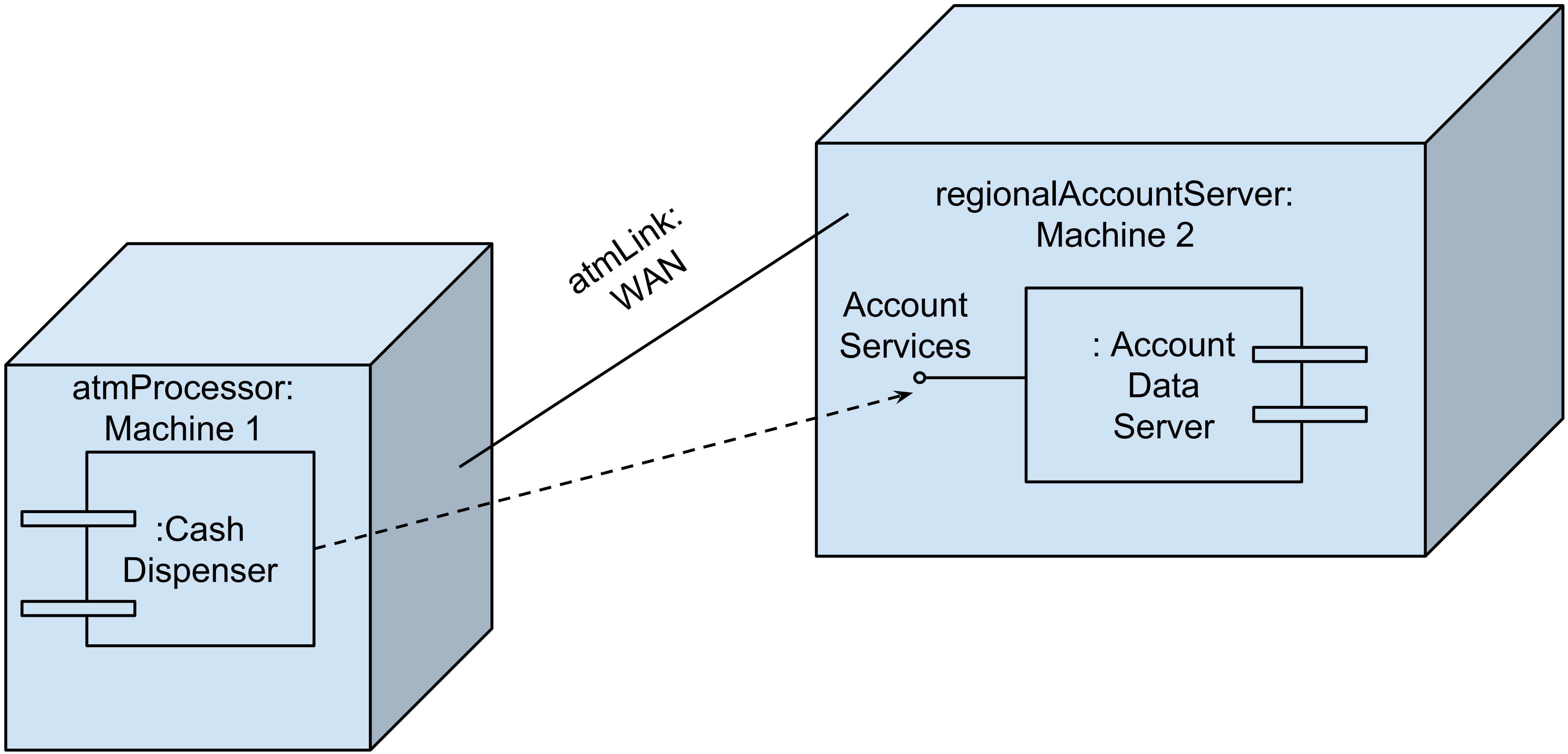
Deployment Diagrams for Software Constructs

- They can also be used for describing the interaction between software architecture and hardware architecture

- Such diagrams show both hardware components and software components

- A software component is any element of software that has both an interface (through which services are provided to the outside world) and a body

Deployment Diagrams for Software Constructs-contd



deployment diagram for part of a bank's ATM

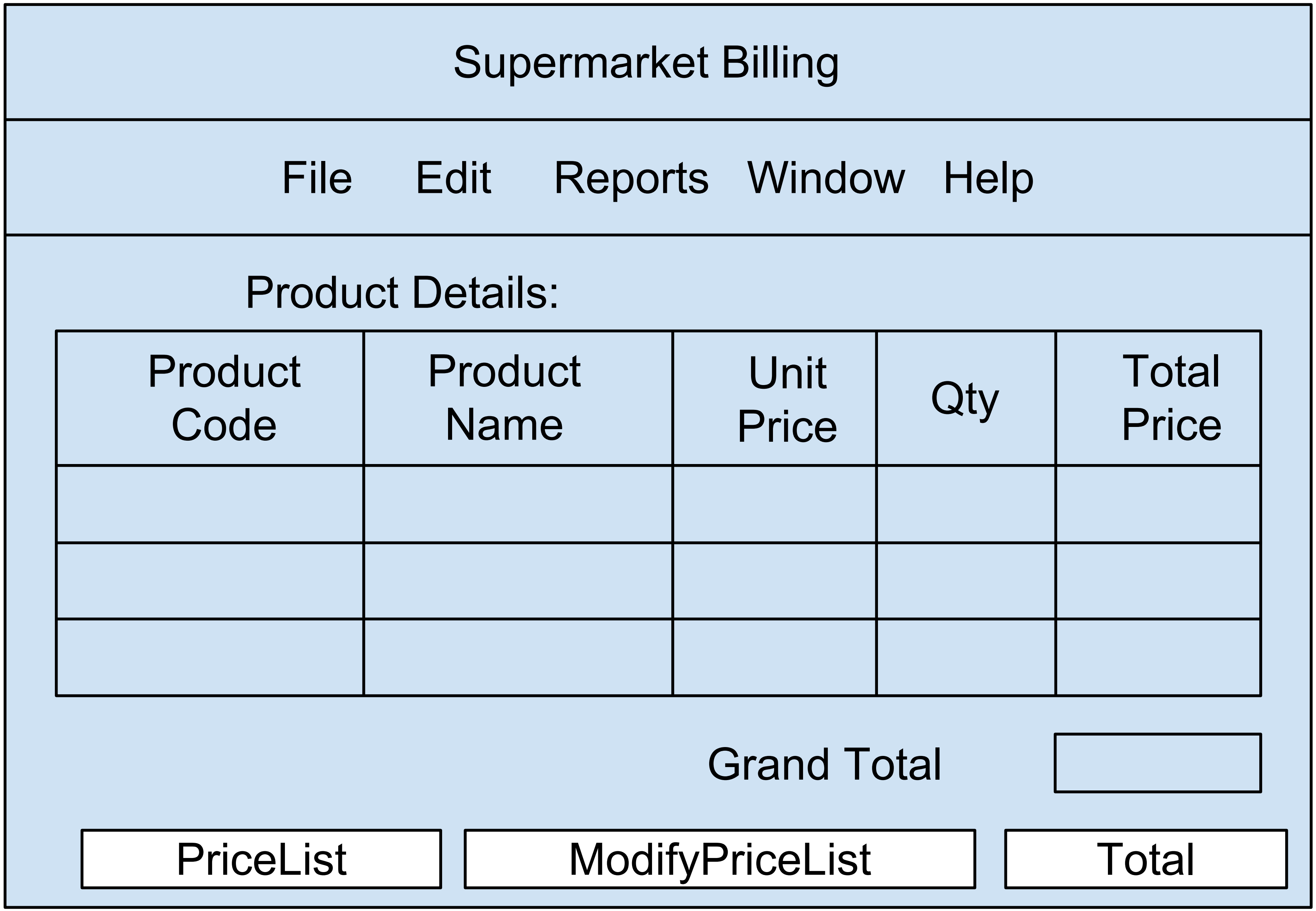
Interface Diagrams

- They depict the human interface of a system

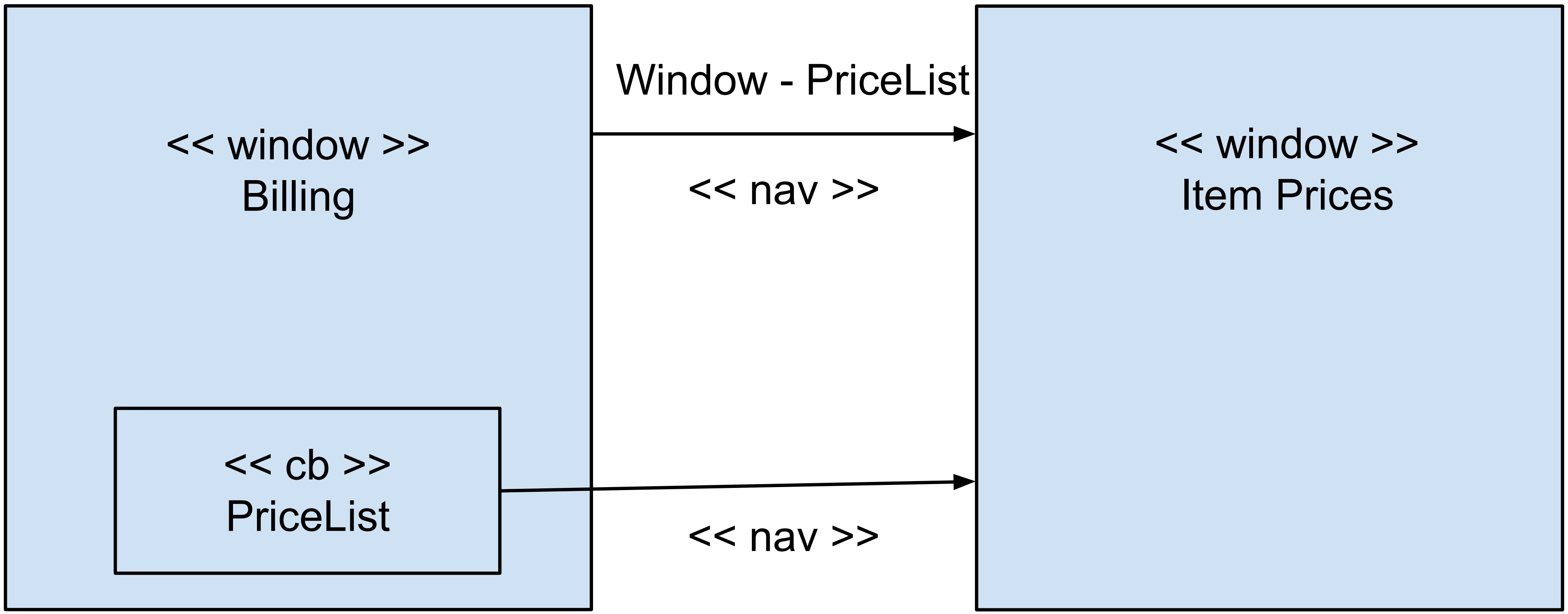
- There are two kinds of interface diagrams

  - Window Layout Diagram
    This is used for representing the properties of every window in the system
  - Window Navigation Diagram
    It is used for showing the navigation between various windows in the system

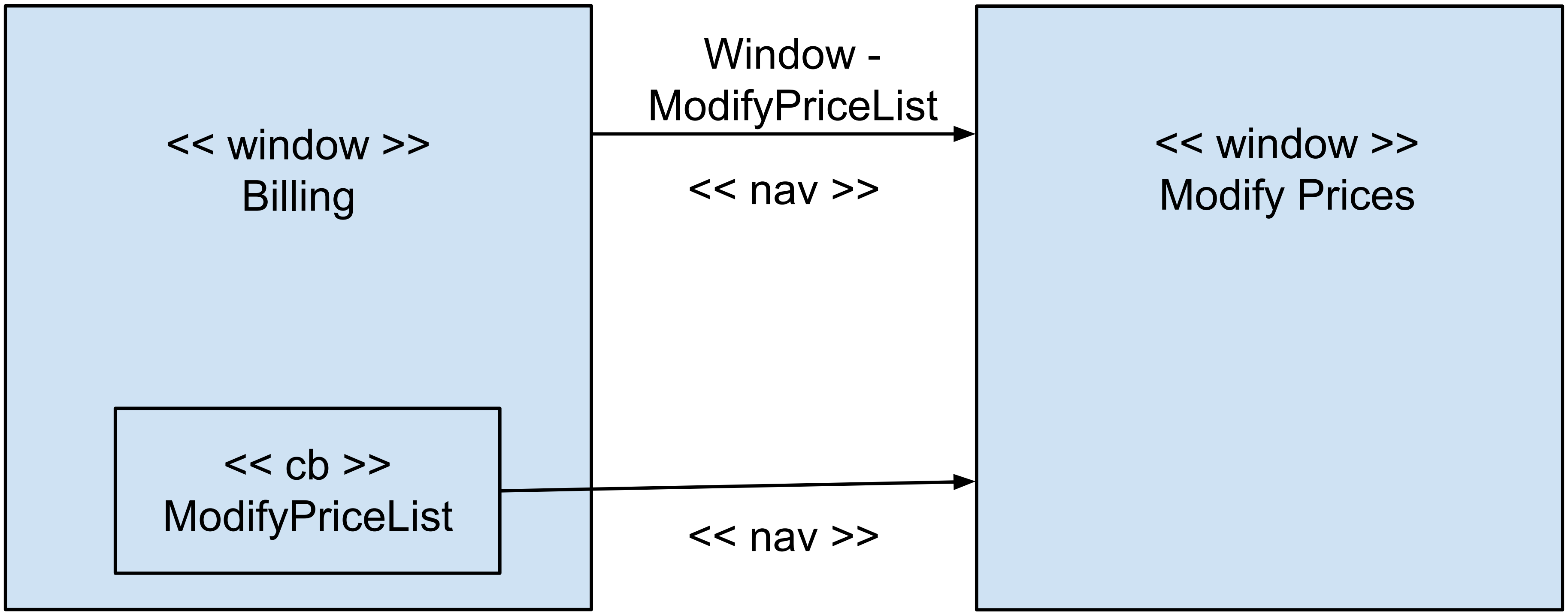
Window Layout Diagram - Supermarket Billing System



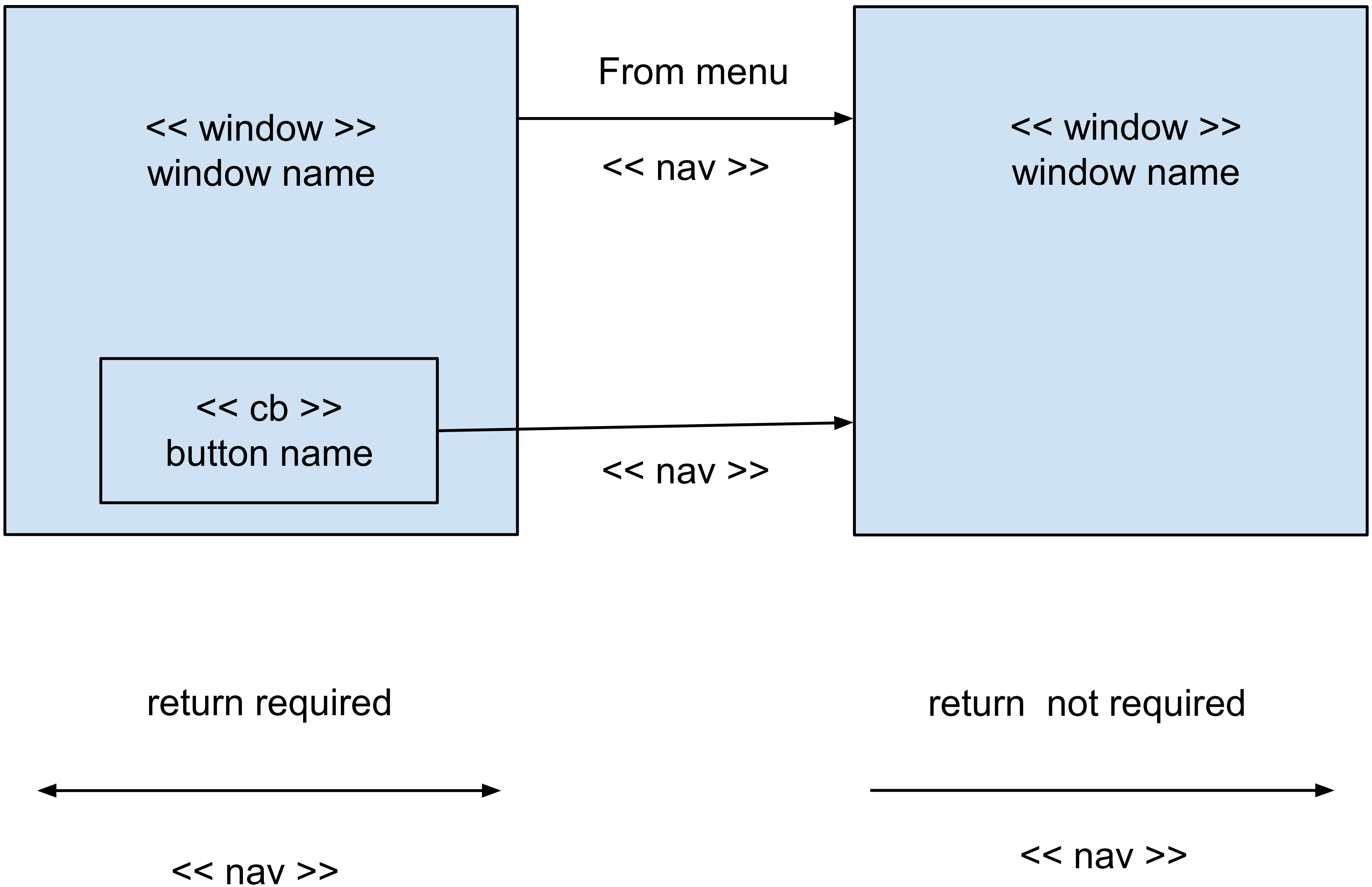
Window Navigation Diagrams - Supermarket Billing System



Window Navigation Diagrams - Supermarket Billing System-contd



Window Navigation Diagram - General Form



Module IV

- Encapsulation Structure
- Connascence
- Domains of Object Classes
- Encumbrance
- Class Cohesion
- State Space and behaviour of classes and subclasses
- class invariant
- preconditions and postconditions
- principle of type conformance
- principle of closed behaviour

Encapsulation Structure

- Encapsulation is a property associated with both object oriented and procedure oriented programming models

- Level - 0 encapsulation
  - raw lines of code with no encapsulation
- Level - 1 encapsulation
  - grouping several lines of code into a function
- Level - 2 encapsulation
  - grouping data and operations into a class
- Level - 3 encapsulation
  - grouping several classes into a package

Encapsulation Structure - contd

To / From	level - 0 (line of code)	level - 1 (operation)	level - 2 (class)
level - 0 (line of code)	structured programming	message fan-out	
level - 1 (operation)	cohesion	coupling	
level - 2 (class)		class cohesion	class coupling

interrelationship between various levels of encapsulation

Encapsulation Structure - contd

- structured programming
  - The principles of structured programming govern the relationship between a line of code and other lines of code within the same procedure
- message fan-out(fan-out)
  - It is a measure of the number of references to other procedures by lines of code within a given procedure
- cohesion
  - It is a measure of the degree of interconnectivity between various statements in a function

Encapsulation Structure - contd

- coupling
  - It is a measure of the degree of interconnectivity between various functions in a program
- class cohesion
  - It is a measure of the degree of interconnectivity between various functions in a class
- class coupling
  - It is a measure of the degree of interconnectivity between various classes in a program

Connascence

- It is another property associated with both object-oriented model and procedure oriented model

defn

connascence between two software elements A and B
means either

1. that you can postulate some change to A that would require B to be changed (or at least carefully checked) in order to preserve overall correctness, or
2. that you can postulate some change that would require both A and B to be changed together in order to preserve overall correctness

Connascence - contd

There are two types of connascence

- static connascence
  - connascence that is present at compile-time
- dynamic connascence
  - connascence that is present at run-time

Varieties of Static Connascence

- connascence of name
- connascence of type or class
- connascence of convention
- connascence of algorithm
- connascence of position

Varieties of Dynamic Connascence

- connascence of execution
- connascence of timing
- connascence of value
- connascence of identity

Varieties of Static Connascence-contd

1. connascence of name

example 1 : int i; // line A

i := 7 ; // line B

There is a connascence of name between A and B

example 2 :

connascence of name exists between a variable in the superclass and the inherited variable in the subclass

2. connascence of type or class

In the above example if i is changed to a char variable then we should assign a char value to it.If i is an object, then corresponding type values should be assigned to the object.

Varieties of Static Connascence-contd

3. connascence of convention

example 1 : int iNumber;
float fValue;

example 2 : #define PI 3.14
#define TRUE 1

4. connascence of algorithm

example : In a hash table a program that inserts symbols and another program that searches symbols should use the same hashing algorithm

5. connascence of position

example : The relative positions of actual arguments and formal arguments should be maintained in a function call

Varieties of Dynamic Connascence-contd

1.connascence of execution
It is the dynamic equivalent of connascence of position

example 1 - Initialising a variable before using it
example 2 - changing and reading the values of global variables in the correct sequence
example 3 - setting and testing semaphore values

2.connascence of timing
This occurs most often in real-time systems

example - An instruction to turn off an X-ray machine must be executed within 'n' milliseconds of the instruction to turn it on

Varieties of Dynamic Connascence-contd

3.connascence of value

It usually involves some arithmetic constraint.

example - The four corners of a rectangle must preserve a certain geometric relationship in their values

4.connascence of identity

If two objects obj1 and obj2 must point to the same object, then there is a connascence of identity between obj1 and obj2

example - If the sales report points to the March spreadsheet , then the operations report must also point to the March spreadsheet

Connascence - contd

Contranascence

Connascence can also indicate difference between sw elements. This is called connascence of difference or negative connascence or contranascence.

example 1 - int i;
int j;

Here i and j should differ. If i is changed to j , then j should be changed to something else

example 2 - If class C inherits from classes A and B, then the features of A and B should not have the same names.

Connascence and Encapsulation Boundaries

The property of connascence is affected by the level of encapsulation.

example 1 - Consider a system containing only a single procedure. Here whenever a new variable is added we have to make sure that it is different from other variables.Thus, contranascence of name affects the system

example 2 - Consider a system having level-1 encapsulation maintaining a hash table. Here there is a connascence of algorithm between hash table update procedure and hash table look up procedure

example 3 - In a system having level - 2 encapsulation, while maintaining a hash table , the connascence of algorithm is between the operations insertSymbol and lookupSymbol of the class Symbol Table

Connascence and Maintainability

Connascence offers three guidelines for improving system maintainabilty

1. Minimise overall connascence by breaking the system into encapsulated elements

2. Minimise any remaining connascence that crosses encapsulation boundaries

3. Maximise the connascence within encapsulation boundaries

Connascence abuses in object oriented systems

1. The Friend Function of C++

2. Unconstrained Inheritance

3. Relying on Accidents of Implementation

example - Consider a class set having operations add, remove, size and retrieve. We can implement a retrieve operation in several ways. We can either retrieve elements in a specific order or in a random order. If we choose different implementations in different parts of a system it will lead to an increase in connascence.

Domains of Object Classes

1. Foundation Domain
2. Architecture Domain
3. Business Domain
4. Application Domain

Domains of Object Classes-contd

1.Foundation Domain

It contains classes valuable across all businesses and architectures.

- Fundamental Classes - Integer, Boolean , Char etc.
- Structural Classes - Stack, Queue, List, BinaryTree, (Container Classes) Set etc.
- Semantic Classes - Date, Time, Angle, Mass, Money, Point, Line etc.

Domains of Object Classes-contd

2. Architecture Domain

It contains classes for one implementation architecture. They are usable in many applications from many different industries.

- Machine Communication classes - Port, RemoteMachine
- Database Manipulation Classes - Transaction, Backup
- Human Interface Classes - Window, Command Button

Domains of Object Classes-contd

3. Business Domain

It contains classes which are useful in many applications, but only those within a single industry such as banking, hospital etc.

- Attribute Classes - Balance , BodyTemperature
- Role Classes - Customer , Patient
- Relationship Classes - AccountOwnership,
  PatientSupervision

Domains of Object Classes-contd

4. Application Domain

It contains classes which are used within a single application.

- Event-recogniser classes

example - An object of class PatientTemperatureMonitor looks for the events patientdevelopsfever and patientbecomeshypothermic

- Event-manager classes -WarmHypothermicPatient,
  SchedulePatientForSurgery

Domains of Object Classes-contd

- Application Domain
- Business Domain
- Architecture Domain
- Foundation Domain

Low Reusability

Medium Reusability

High Reusability

Encumbrance

It measures the total number of classes that the given class must rely in order to work.

Defn

The direct encumbrance of a class is the size of its direct class reference set.

The indirect encumbrance of a class is the size of its indirect class reference set.

Encumbrance-contd

Defn- direct class reference set

The direct class reference set of a class C is the set of classes to which C refers directly.

A class C may refer directly to another class D in any of the following ways.

- C inherits from D
- C has an attribute of class D

Defn- direct class reference set-contd

- C has an operation with an input argument of class D
- C has a variable of class D
- C has a method that sends a message with a returned argument of class D
- C has a method containing a local variable of class D
- C has a friend class D

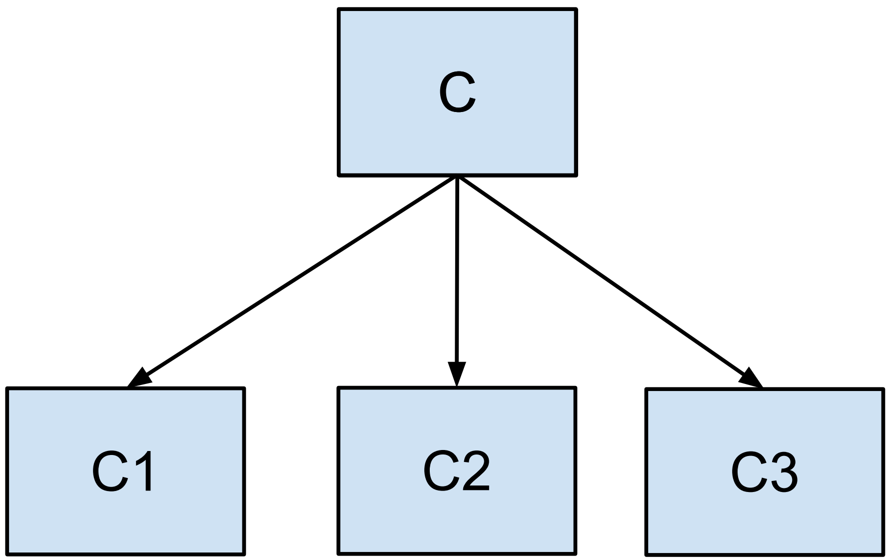
Defn- indirect class reference set

Let the direct class reference set of C comprise the classes C_1, C_2, C_3, ..., C_n.

Then, the indirect class reference set of C is the union of the direct class reference set of C and the indirect class reference sets of C_1, C_2, C_3, ..., C_n.

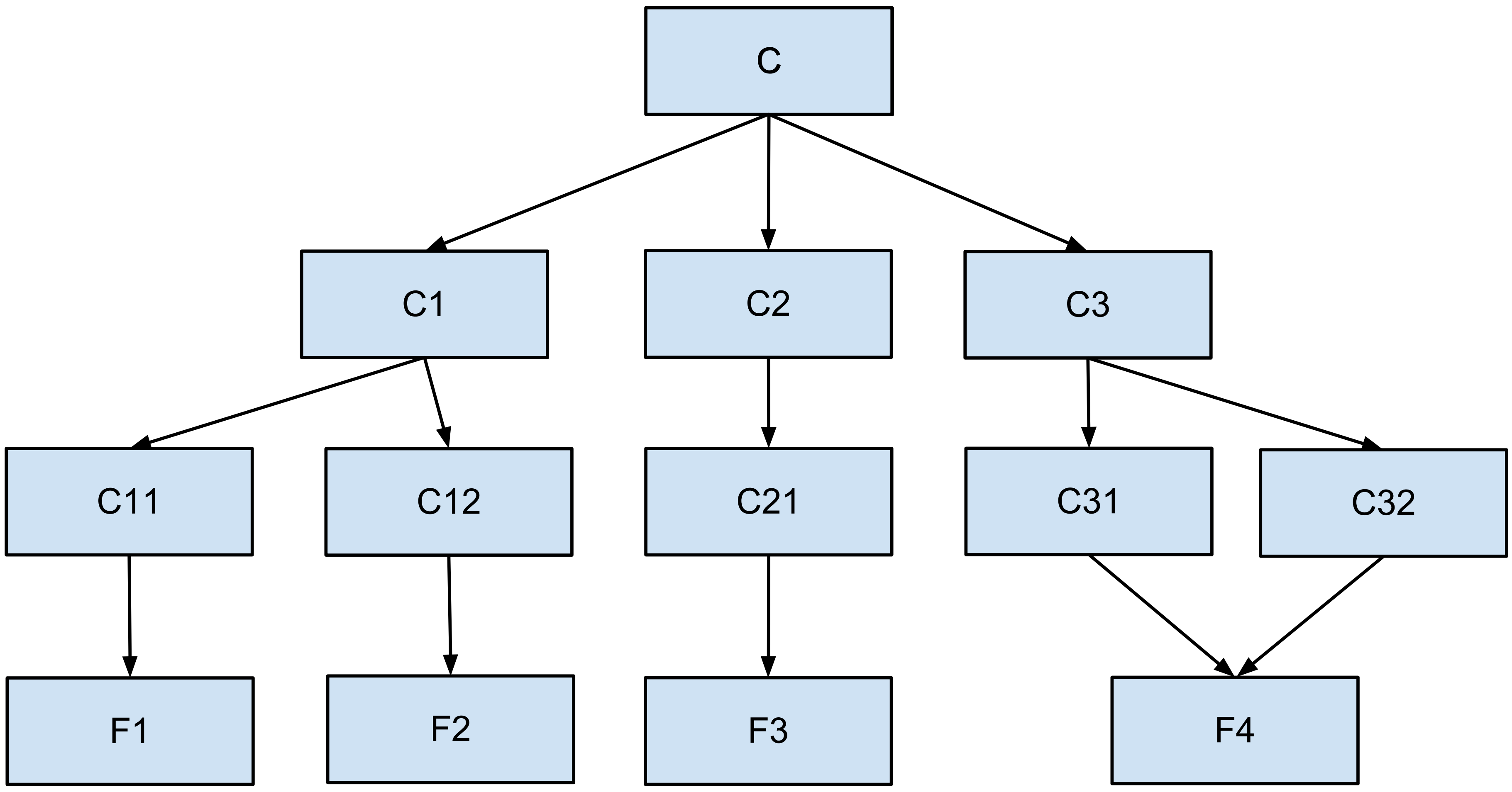
Indirect class reference set

Let the direct class reference set of C be C_1, C_2 and C_3.

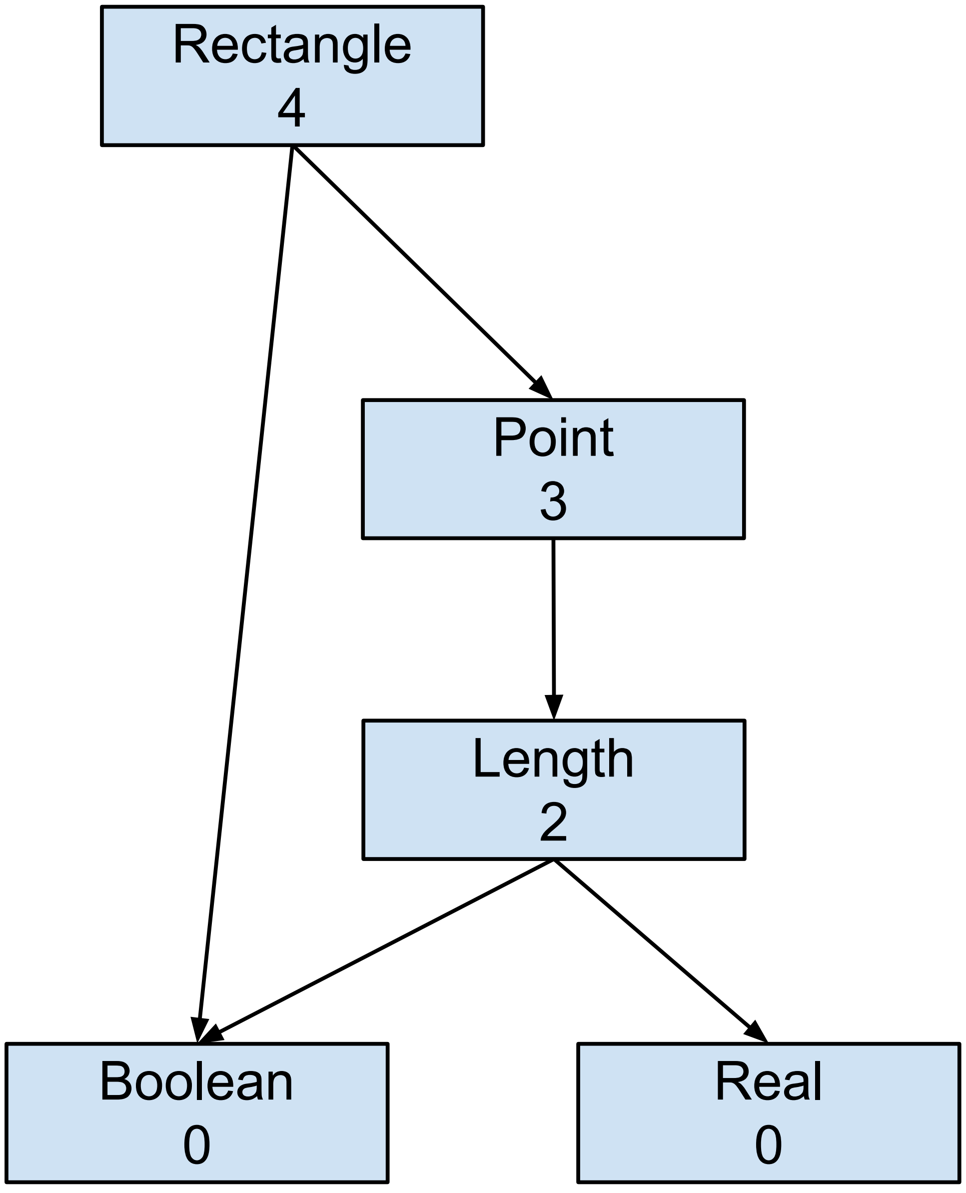


Indirect class reference set-contd

Then the indirect class reference set of C can be represented as given below.



Indirect class reference set-contd



The use of Encumbrance

Encumbrance gives us a measure of how high a class is above the fundamental domain. Thus, classes in higher domains have high indirect encumbrance and those in lower domains have low indirect encumbrance.

A good object oriented system should minimise encumbrance. Law of Demeter is a guiding principle for limiting encumbrance.

The Law of Demeter-defn

For an object of class C and for any operation op defined for obj, each target of a message within the implementation of op must be one of the following objects.

1. The object obj itself - this (in C++ and Java)
2. An object referred to by an argument within op's signature
3. An object referred to by a variable of obj
4. An object created by op
5. An object referred to by a global variable

The Law of Demeter-contd

There are two versions of the law , differing only in their interpretation of point 3.

The Strong Law of Demeter defines a variable as being only a variable defined in the class C itself.

The Weak Law of Demeter defines a variable as being either a variable of C or a variable that C inherits from its superclasses.

Class Cohesion

defn

It is the measure of the interrelatedness of the features(attributes and operations) located in the external interface of a class.

A class with low (bad) cohesion has a set of features that don't belong together.

A class with high (good) cohesion has a set of features that all contribute to the type abstraction implemented by the class.

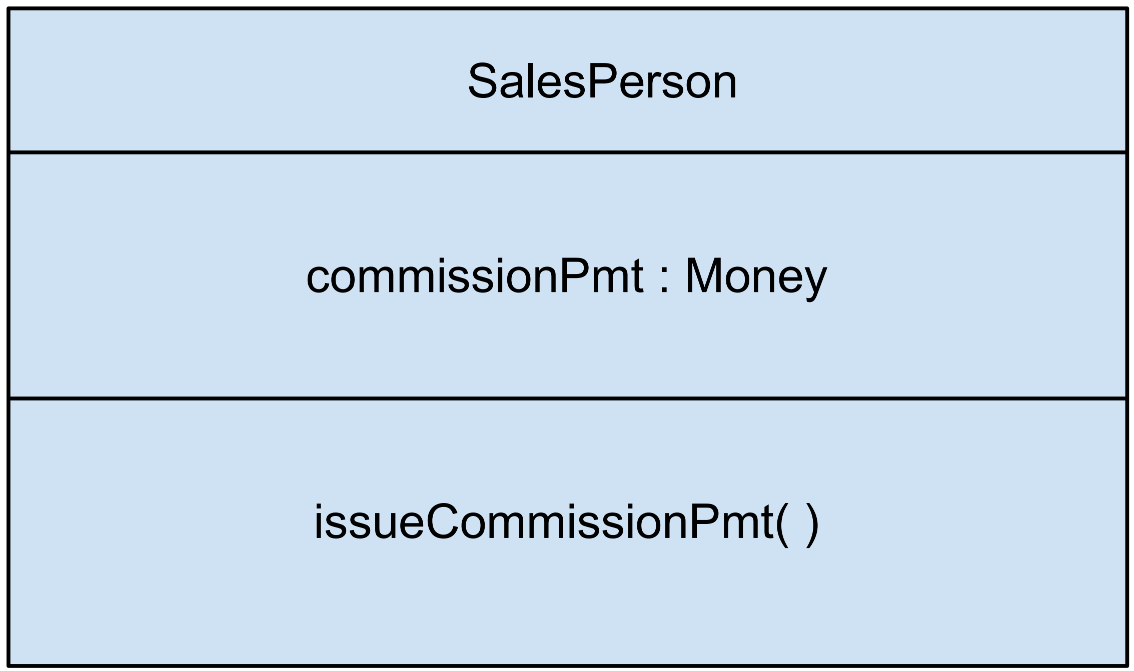
Class Cohesion-contd

There are three types of class cohesion which are problematic.

- Mixed-instance Cohesion (most problematic)
- Mixed-domain Cohesion
- Mixed-role Cohesion (least problematic)

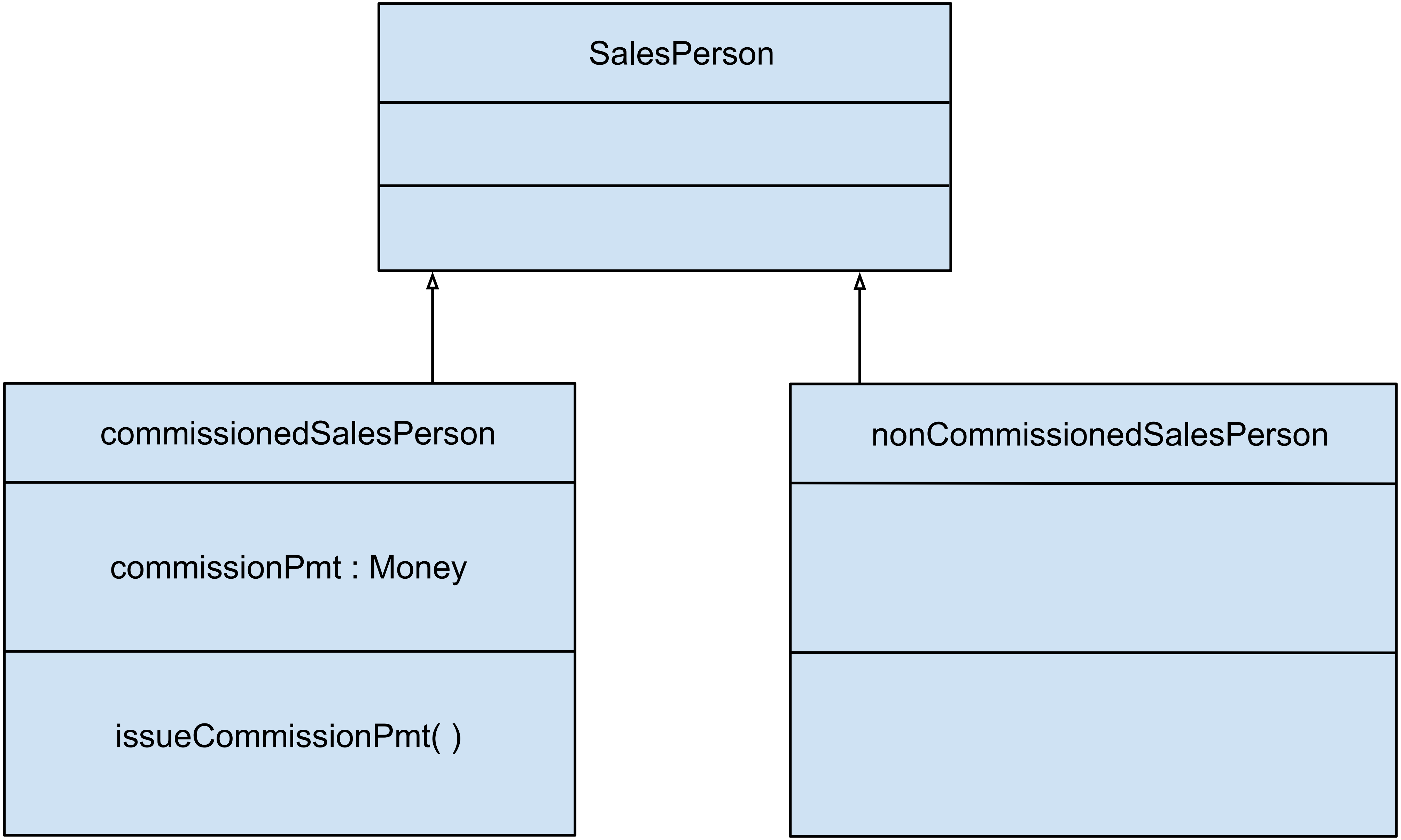
Mixed-instance Cohesion

A class with mixed-instance cohesion has some features that are undefined for some objects of the class.



Fred is a Commissioned SalesPerson
Mary is a nonCommissioned SalesPerson

Mixed-instance Cohesion-contd



Mixed-domain Cohesion

A class with mixed-domain cohesion contains an element that directly encumbers the class with an extrinsic class of a different domain.

Class B is extrinsic to class A if A can be fully defined with no notion of B.
example - Elephant is extrinsic to Person

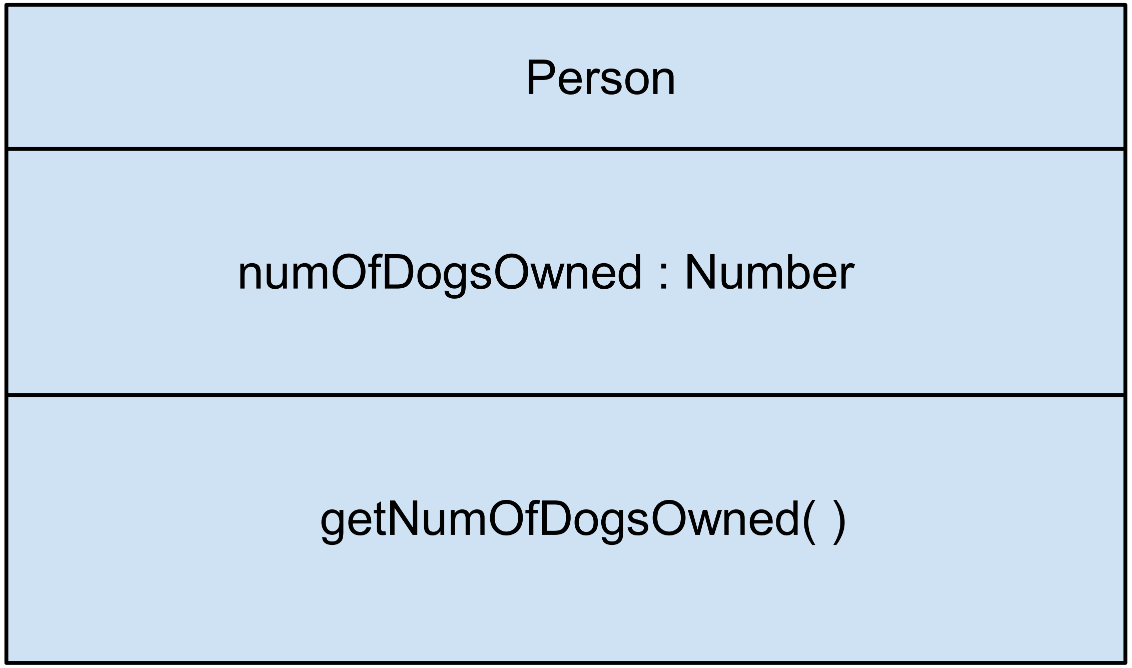
Class B is intrinsic to class A if B captures some characteristic inherent to A.
example - Date(as in date of birth) is intrinsic to Person

Mixed-domain Cohesion-contd

Real
arcTan : Angle euroAmount : Money tempValue : Temperature

Mixed-role Cohesion

A class with mixed - role cohesion contains an element that directly encumbers the class with an extrinsic class that lies in the same domain as C.



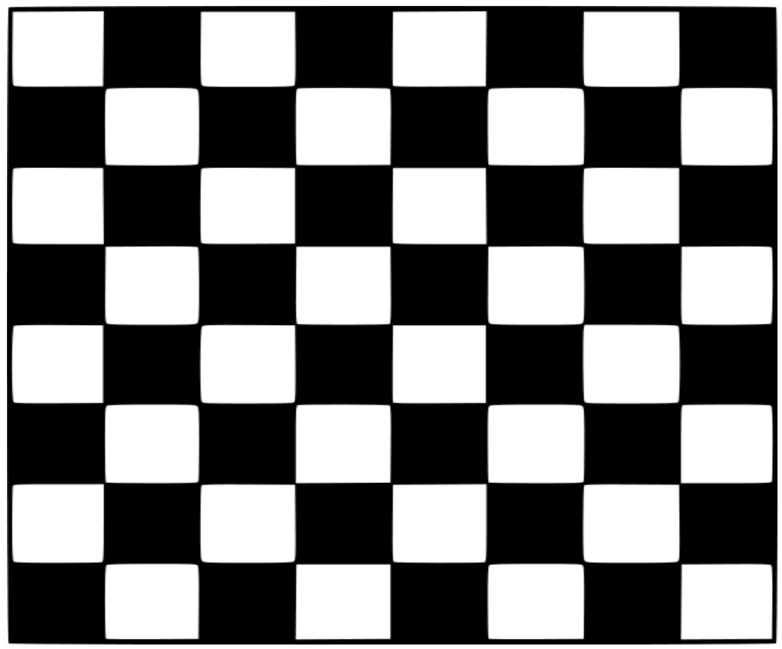
Mixed-role Cohesion-contd

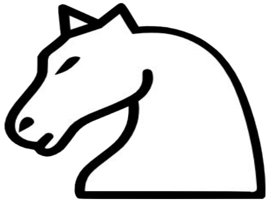
In pure design terms, mixed role cohesion is the least serious problem.When the classes are reused, then you should pay careful attention to mixed role cohesion. What if you wanted to reuse Person in an application that had no dogs?

If several attributes such as this like numOfCarsOwned, numOfBoatsOwned, numOfCatsOwned are added , then the degree of mixed role cohesion will increase.

State Space and Behaviour of a Class

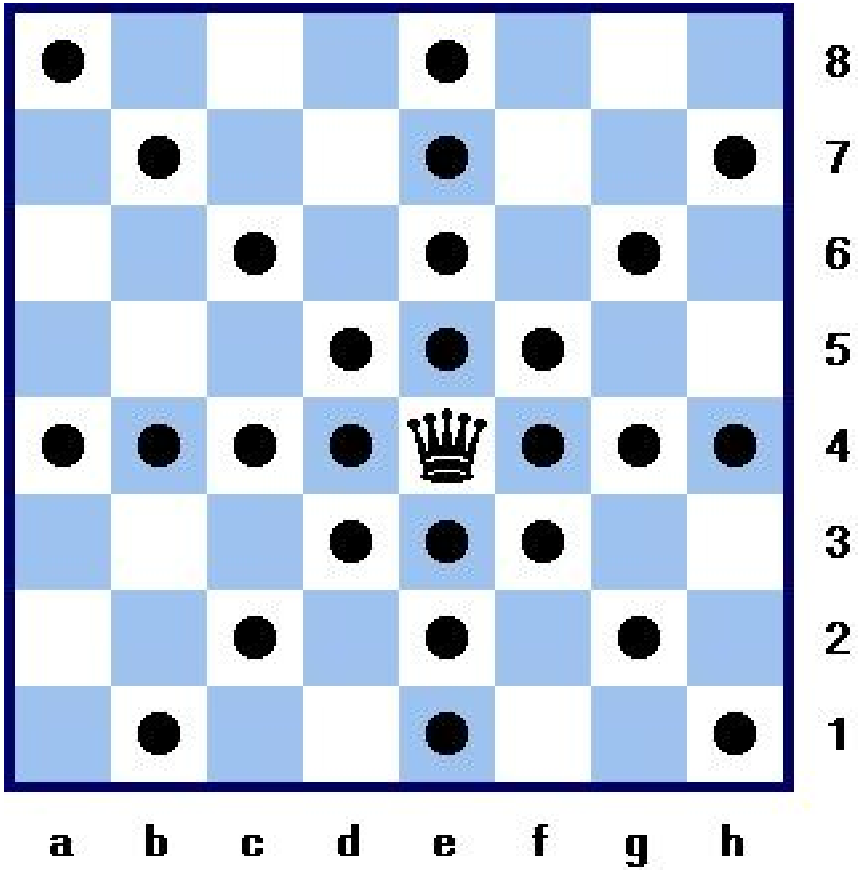
The total state space of a chess queen is the entire chessboard



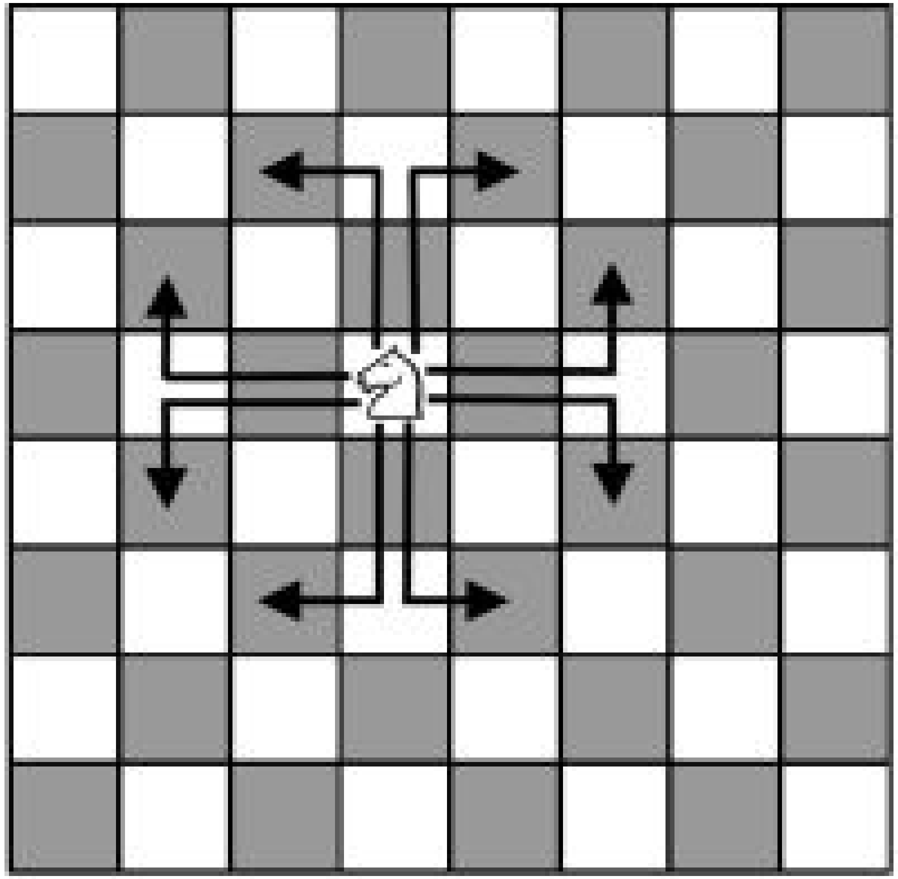


The total state space of a chess knight is also the entire chessboard

State Space and Behaviour of a Class - contd

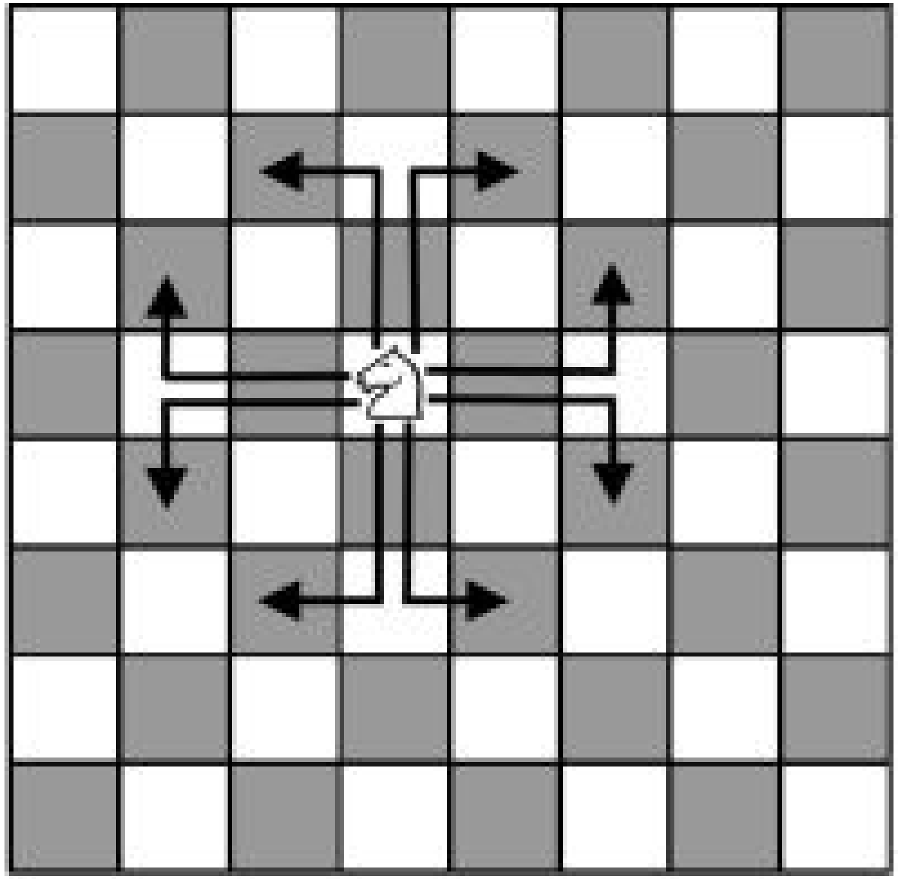


Queen Behaviour

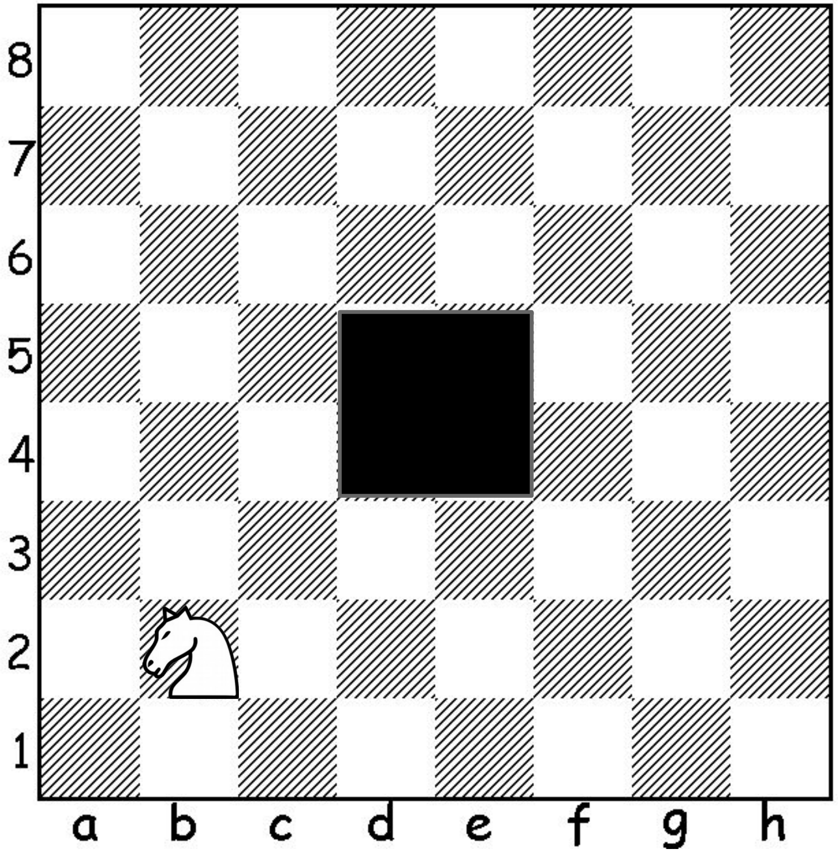


Knight Behaviour

State Space and Behaviour of a Class - contd



Traditional Knight



Special Knight

State Space and Behaviour of a Class - contd

From the above examples, we see that two classes may differ either in their state space or in their behaviour or both.

state space - defn
The state space of a class C is the ensemble of all the permitted states of any object of class C.
The dimensions of a state space are the coordinates needed to specify the state of a given object.

behaviour - defn
The behaviour of a class C is the set of transitions that an object of class C is permitted to make between states in C's state space.

State Space and Behaviour of a Class - contd

Informally, the state of an object is the value it has at a given time. More properly, an object's state is the ordered set of objects to which the object refers at a given time.

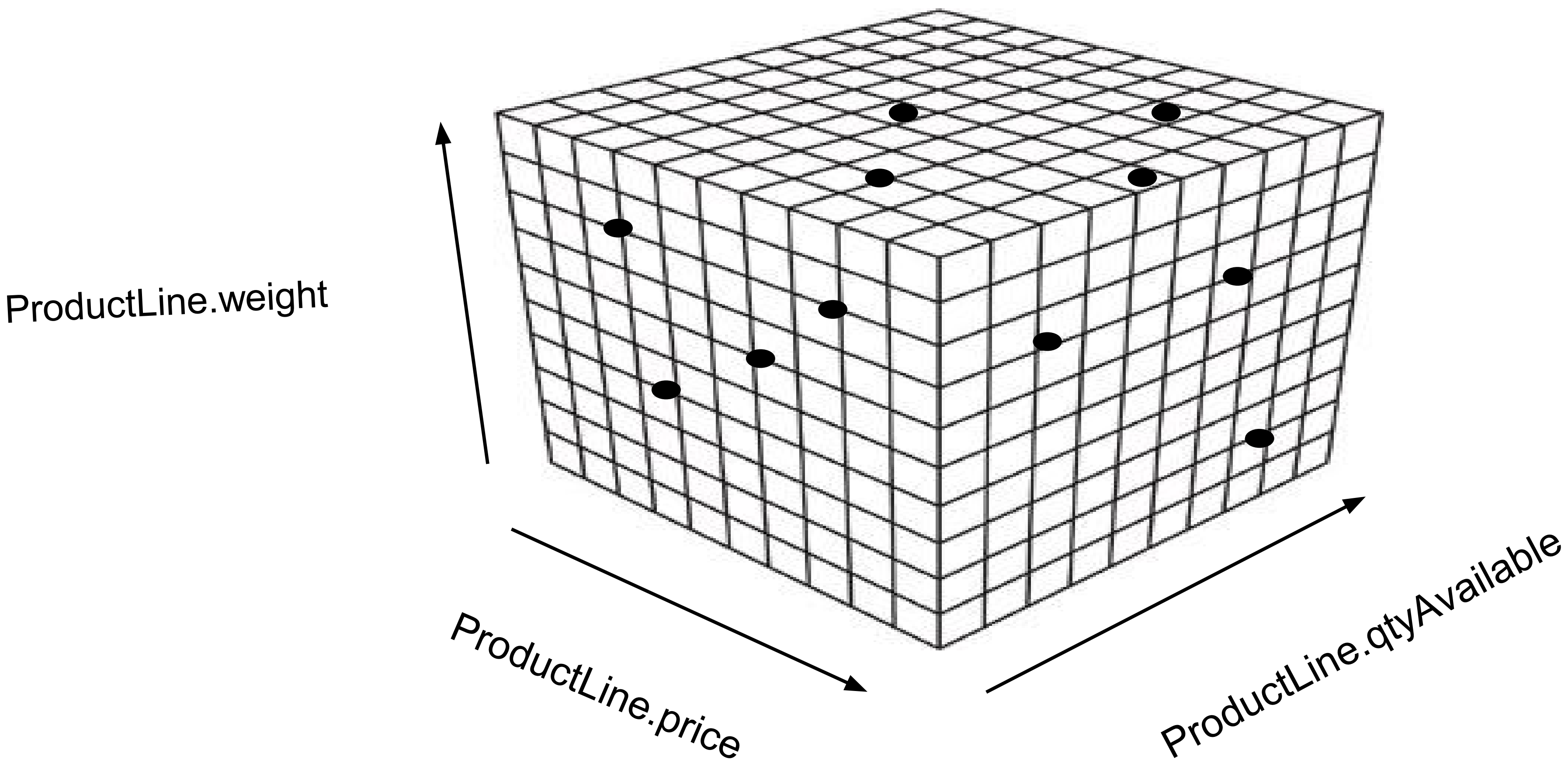
example

The state of a Swimming Pool object might be
(30, 2, 25) (length(m),depth(m),temperature(Celsius)).
In other words, this Swimming Pool object currently points to an object of class Length (30 m), another object of class Length (2 m) and one of class Temperature (25^{o} C)

State Space and Behaviour of a Class - contd

A class's state space can be specified as a grid of points, each point being a state.

The State Space for the class Product Line



State Space and Behaviour of a Class - contd

As another example, the Patient class may have such dimensions as age, height, weight, temperature and so on.

The dimensions of a class's state space are roughly equivalent to the attributes defined on the class. Most attributes that could be derived from others are not normally considered dimensions.

example - The dimensions of a Cube may be length, breadth and height, but not volume or area.

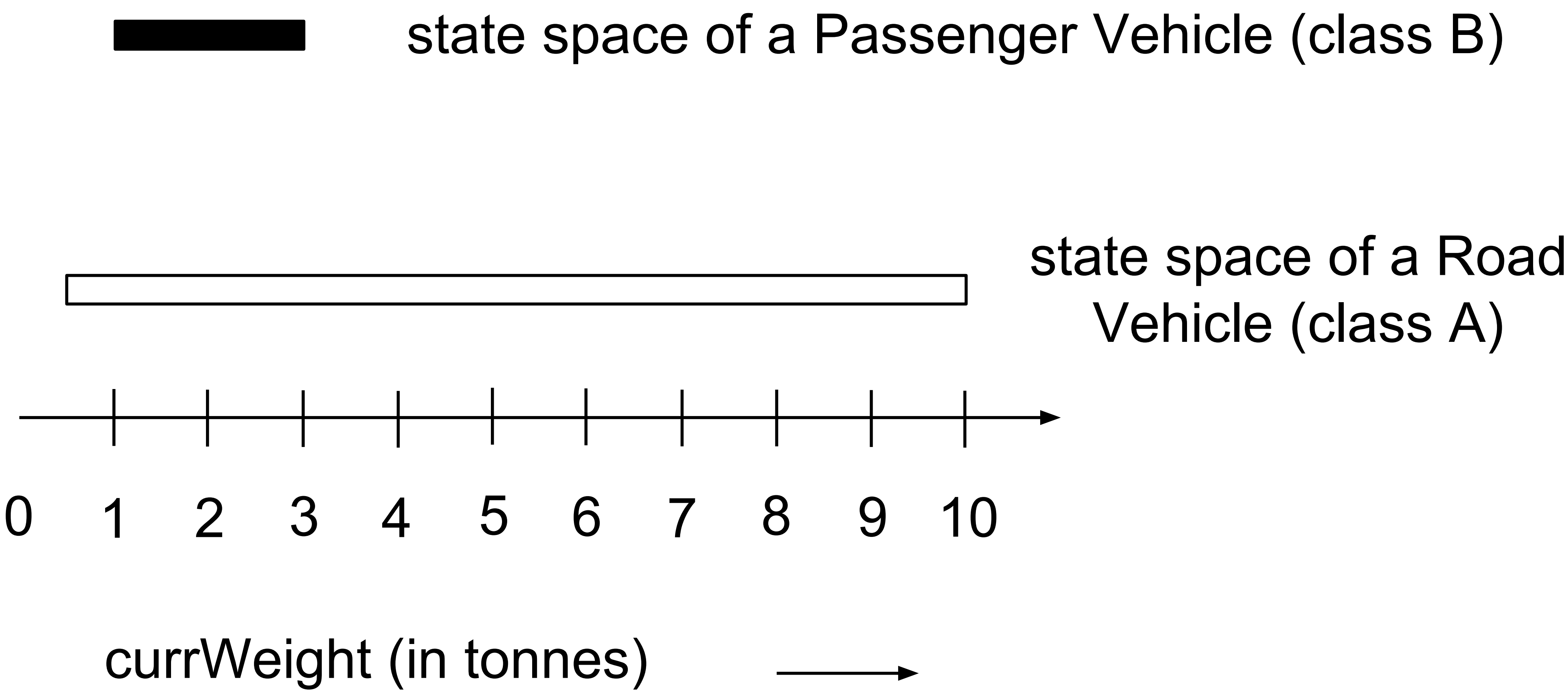
State Space and Behaviour of a Class - contd

The classes that mark dimensions are not always from such a lowly domain as Money and Length.

example - Hammer's state space has two dimensions, Handle and Head, because you make a hammer (a composite object) by selecting one Handle and one Head (the two component objects). Each of these two classes is itself from the business domain and itself has several dimensions (such as length, weight and so on).

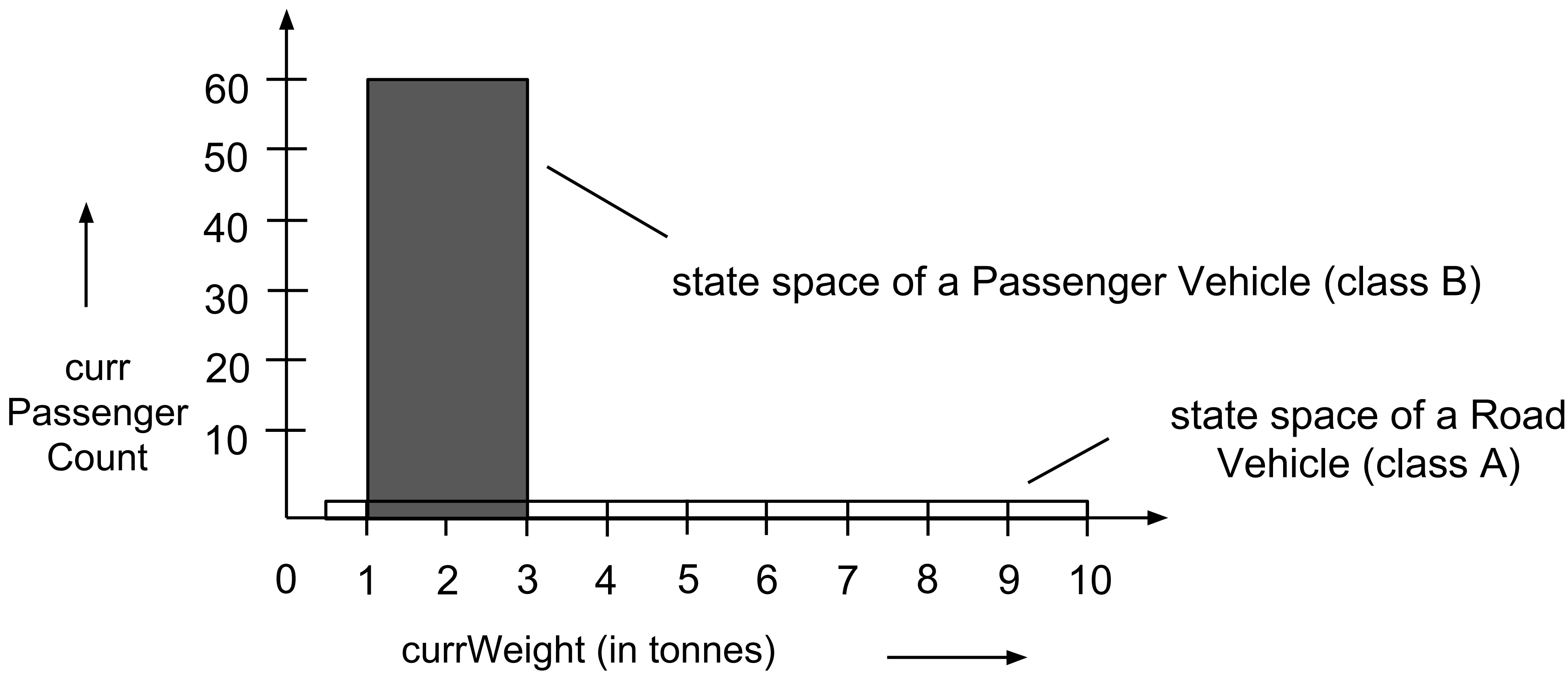
State Space of a subclass

If B is a subclass of A, then the state space of B must be entirely contained within the state space of A. We say that B's state space is confined by A's state space.



State Space of a subclass-contd

If B is a subclass of A, then B's state space must comprise at least the dimensions of A's - but it may comprise more. If it comprises more dimensions, we say that B's state space extends from A's.



The Behaviour of a SubClass

Passenger Vehicle will have behaviour, facilitated by operations such as pickUpPassenger and dropOffPassenger.This example shows that PassengerVehicle may extend the behaviour of its superclass, RoadVehicle .

Passenger Vehicle's behaviour may also be confined by the behaviour of RoadVehicle.
We can possibly add five tonnes to the weight of a general RoadVehicle, as long as it doesn't exceed the ten tonne limit. However, we can never add five tonnes to a Passenger Vehicle's weight because its maximum weight is three tonnes.

Class Invariant

The legal state space of a class is defined by its class invariant.

defn

A class invariant is a condition that every object of that class must satisfy at all times (when object is in equilibrium).

An object is in equilibrium when it is not in the middle of changing states.

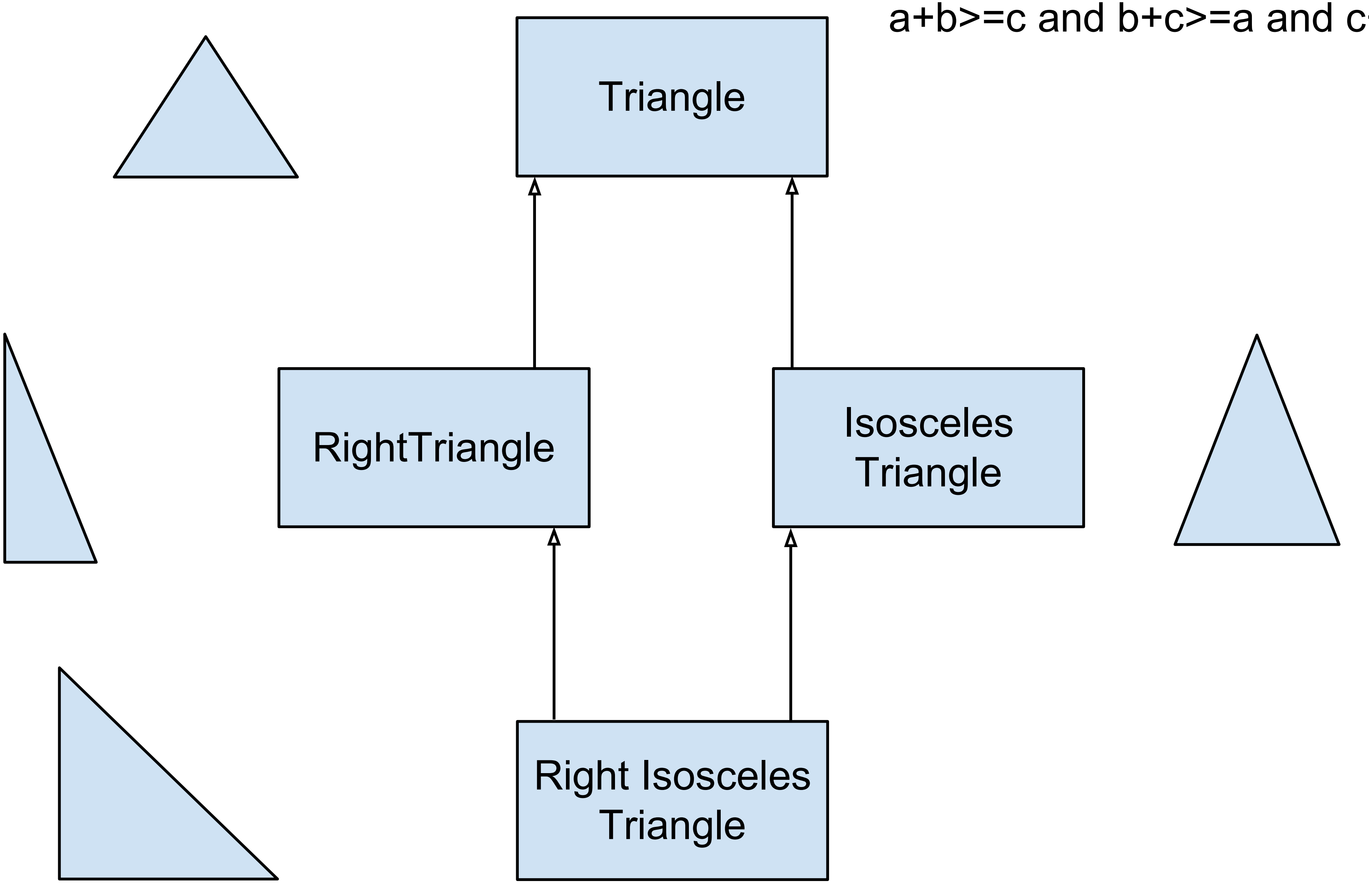
example - The class invariant of a Triangle object is

$$a+b>= c \text{ and } b+c>=a \text{ and } c+a>=b$$

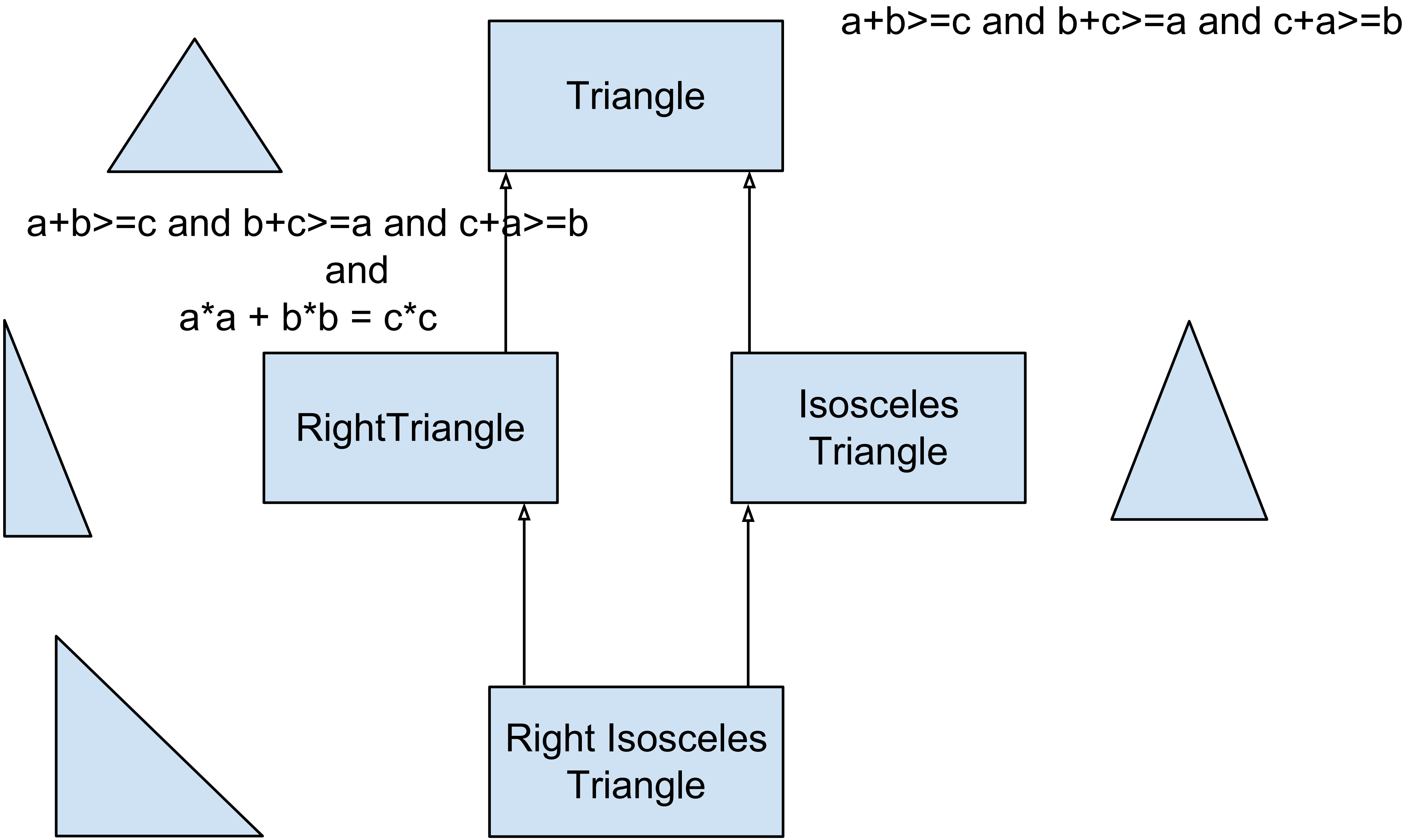
where a,b and c are the sides of a triangle

Class Invariant - contd

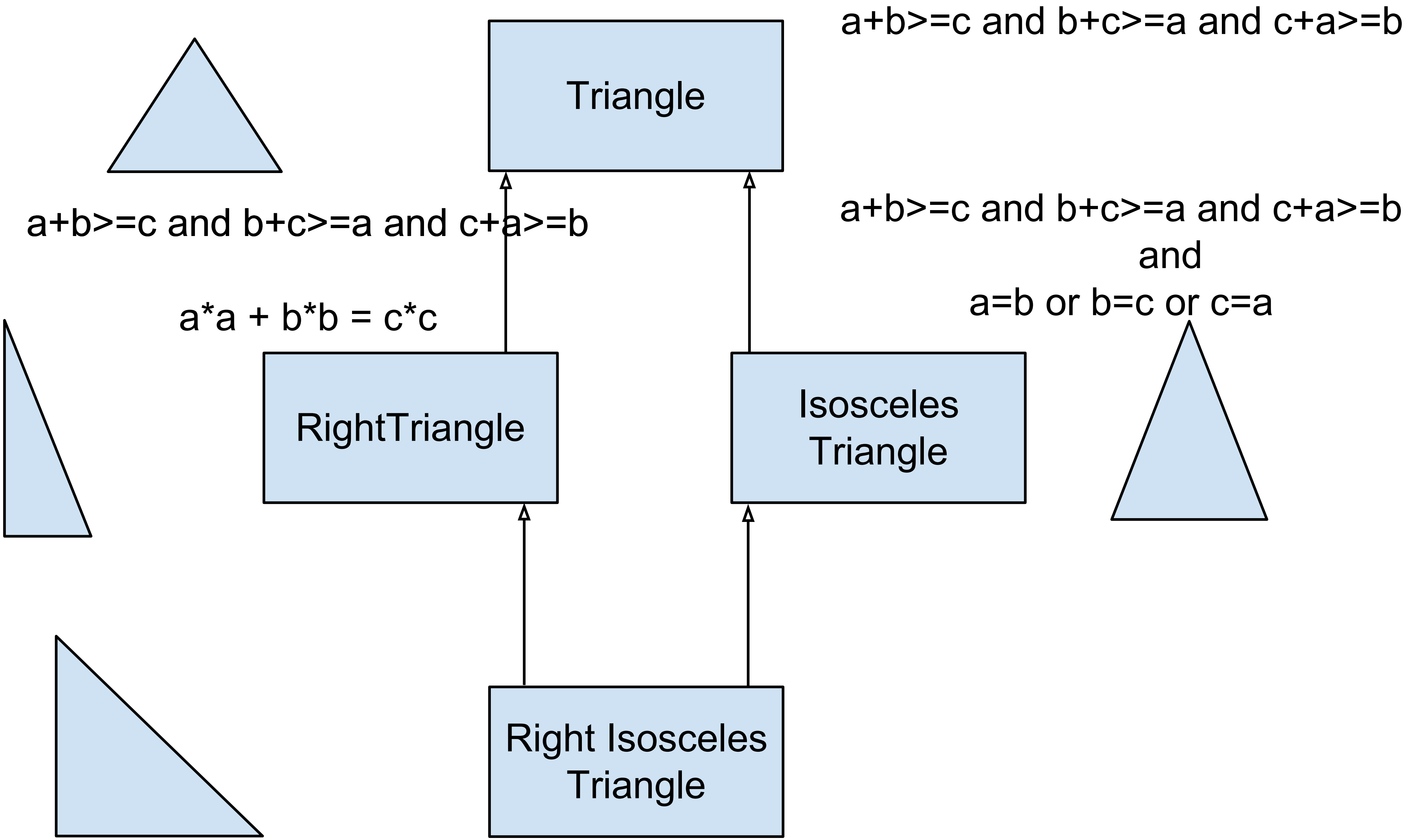
a+b>=c and b+c>=a and c+a>=b



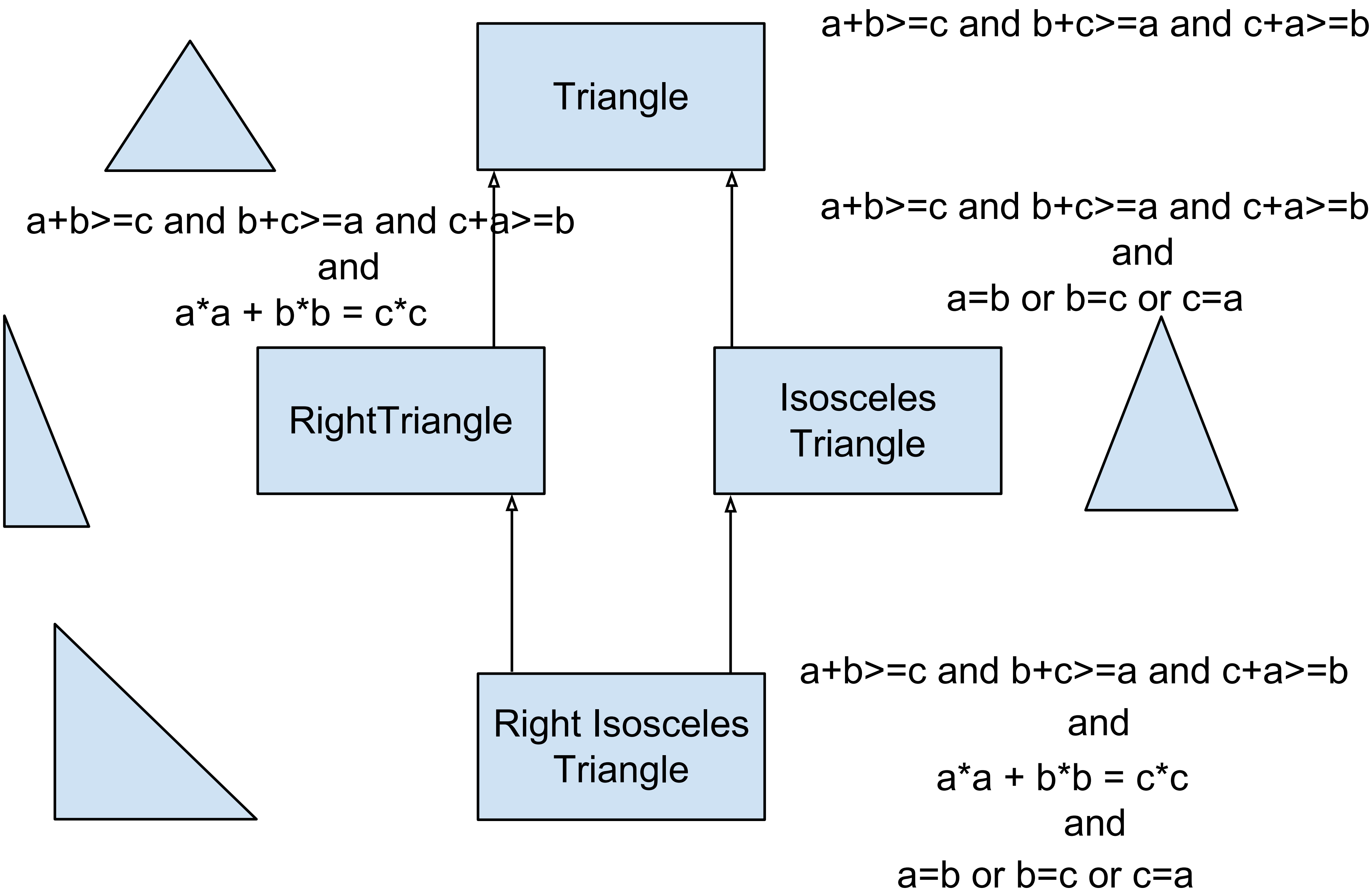
Class Invariant - contd



Class Invariant - contd



Class Invariant - contd



Preconditions and Postconditions

Every operation has a precondition and postcondition.

The precondition is a condition that must be true when the operation begins to execute. If it is not true, then the operation may refuse to execute and possibly raise some exception condition.

The postcondition is a condition that must be true when the operation ends its execution. If it is not true, then the operation's implementation is defective and must be corrected.

Preconditions and Postconditions-contd

operation

stack.pop

precondition

not empty

postcondition

(numElements = old numElements - 1) and not full

Preconditions and Postconditions-contd

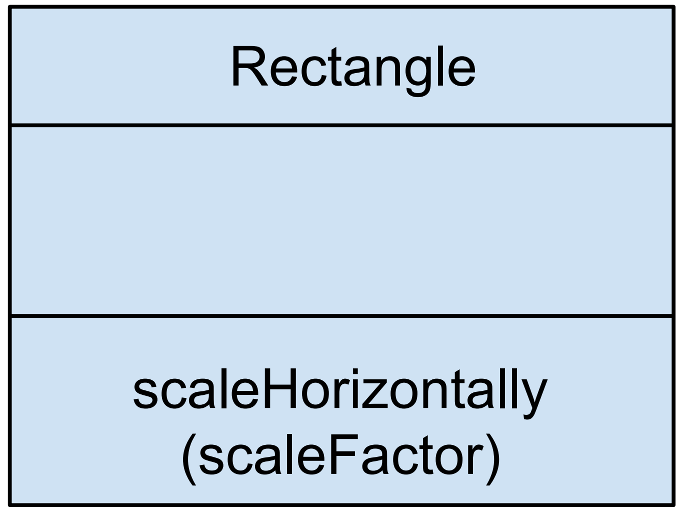
The execution of an operation takes place between two compound conditions.

class invariant and operation precondition

operation executes

class invariant and operation postcondition

Preconditions and Postconditions-contd



class invariant

w1 = w2 and h1 = h2

precondition

maxAllowedWidth >= w1 * scaleFactor

postcondition

w1 = old w1 * scaleFactor

Preconditions and Postconditions-contd

Overall ,

w1 = w2 and h1 = h2 and maxAllowedWidth >= w1 * scaleFactor
// class invariant and precondition

scaleHorizontally(scaleFactor)//operation executes

w1 = w2 and h1 = h2 and w1 = old w1 * scaleFactor
// class invariant and postcondition

Type Conformance and Closed Behaviour

The principle of type conformance and the principle of closed behaviour are vital to the construction of healthy class hierarchies.

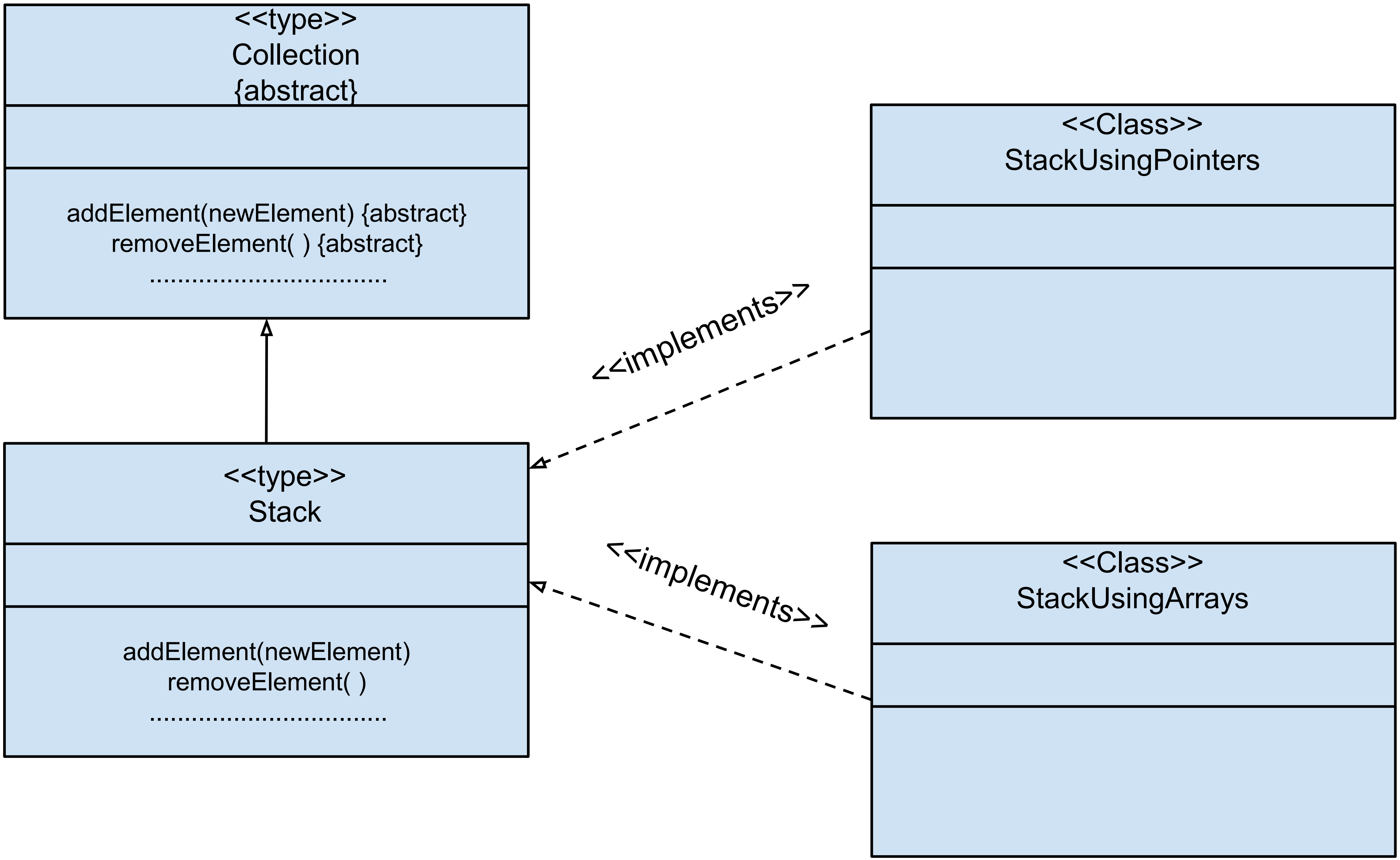
Class Vs Type

Type includes the purpose of the class, together with its state space and behaviour.Type is the abstract or external view of a class.

Class is an implementation of a type. A class includes the design of the variables of the class and the design of the algorithms for the operations' method.

A single type may be implemented as several classes, with each class having its own particular internal design.

Class Vs Type - contd



The Principle of Type Conformance

defn

If S is a true subtype of T , then S must conform to T. In other words, an object of type S can be provided in any context where an object of type T is expected, and correctness is still preserved when any accessor operation of the object is executed.

example - Circle is a subtype of Ellipse. Any object that is a Circle is also an Ellipse - a very round Ellipse. So any operation that is expecting to receive an Ellipse as an argument in a message should be very happy to get a Circle .

In a sound OO design, the type of each class should conform to the type of its superclass. In other words, the class/subclass inheritance hierarchy should follow the principle of type conformance.

The Principle of Type Conformance-contd

To ensure principle of type conformance, the following conditions are to be satisfied.

1. The class invariant of the subclass is at least as strong as that of the superclass.

example - Rectangle has the invariant

$$w1 = w2 \text{ and } h1 = h2$$

Square has the invariant

$$w1 = w2 \text{ and } h1 = h2 \text{ and } w1 = h1$$

Clearly square's invariant is stronger than rectangle's.

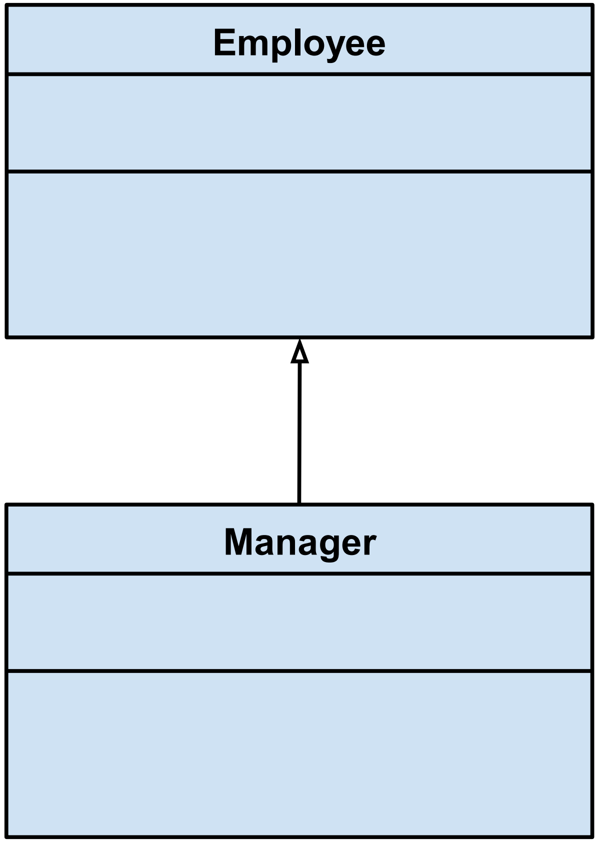
The Principle of Type Conformance-contd

2. Every operation of the superclass has a corresponding operation in the subclass with the same name and signature.

3. Every operation's precondition is no stronger than the corresponding operation's precondition in the superclass - Principle of Contravariance.

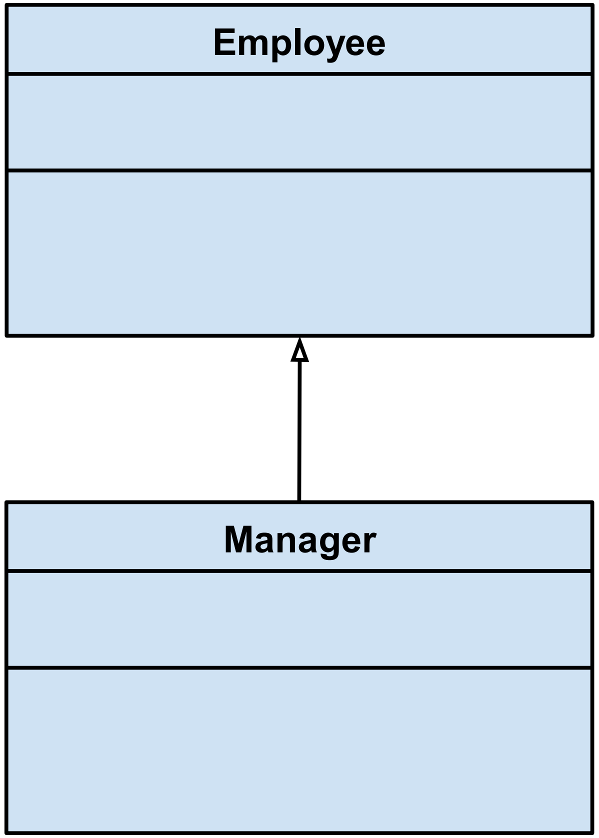
4. Every operation's postcondition is at least as strong as the the corresponding operation's postcondition in the superclass - Principle of Covariance.

An Example of Contravariance and Covariance



What must we do to ensure that Manager is a valid subtype of Employee?

An Example of Contravariance and Covariance-contd

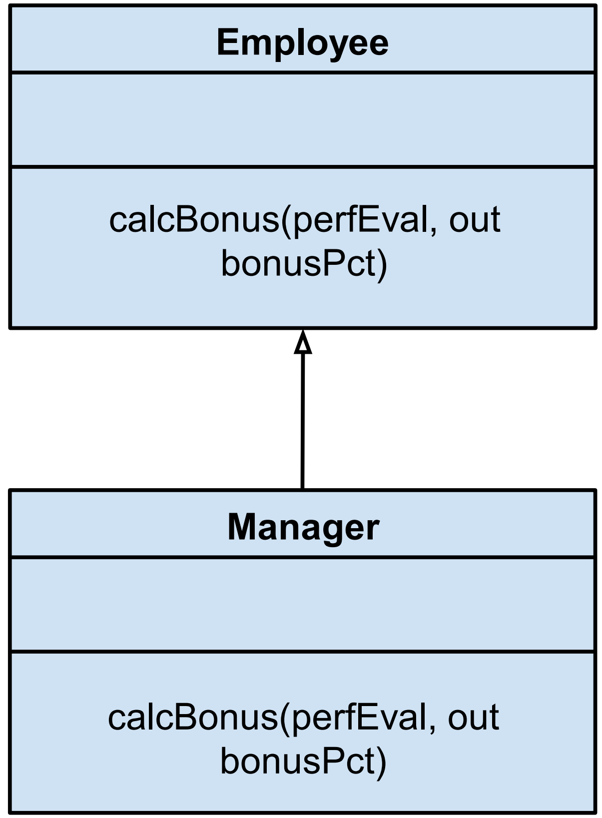


class invariant
gradeLevel > 0

class invariant
gradeLevel > 20

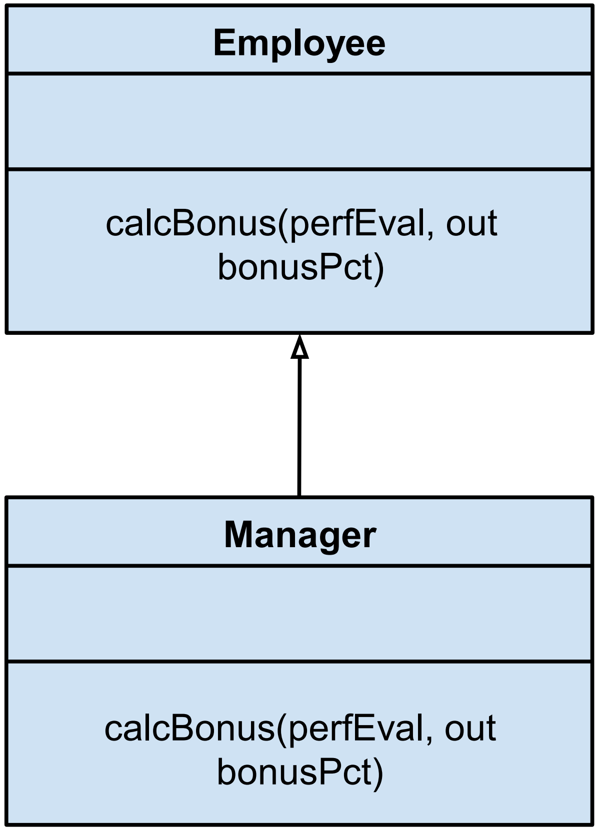
That makes Manager's class invariant stronger than Employee's and that is the way it should be.

An Example of Contravariance and Covariance-contd



We will say for simplicity that perfEval is an integer between 0 and 5. The output argument bonusPct is between 0% and 10%.(in Employee Class)

An Example of Contravariance and Covariance-contd



legal ranges of perfEval

0 to 5

0 to 8

-1 to 9

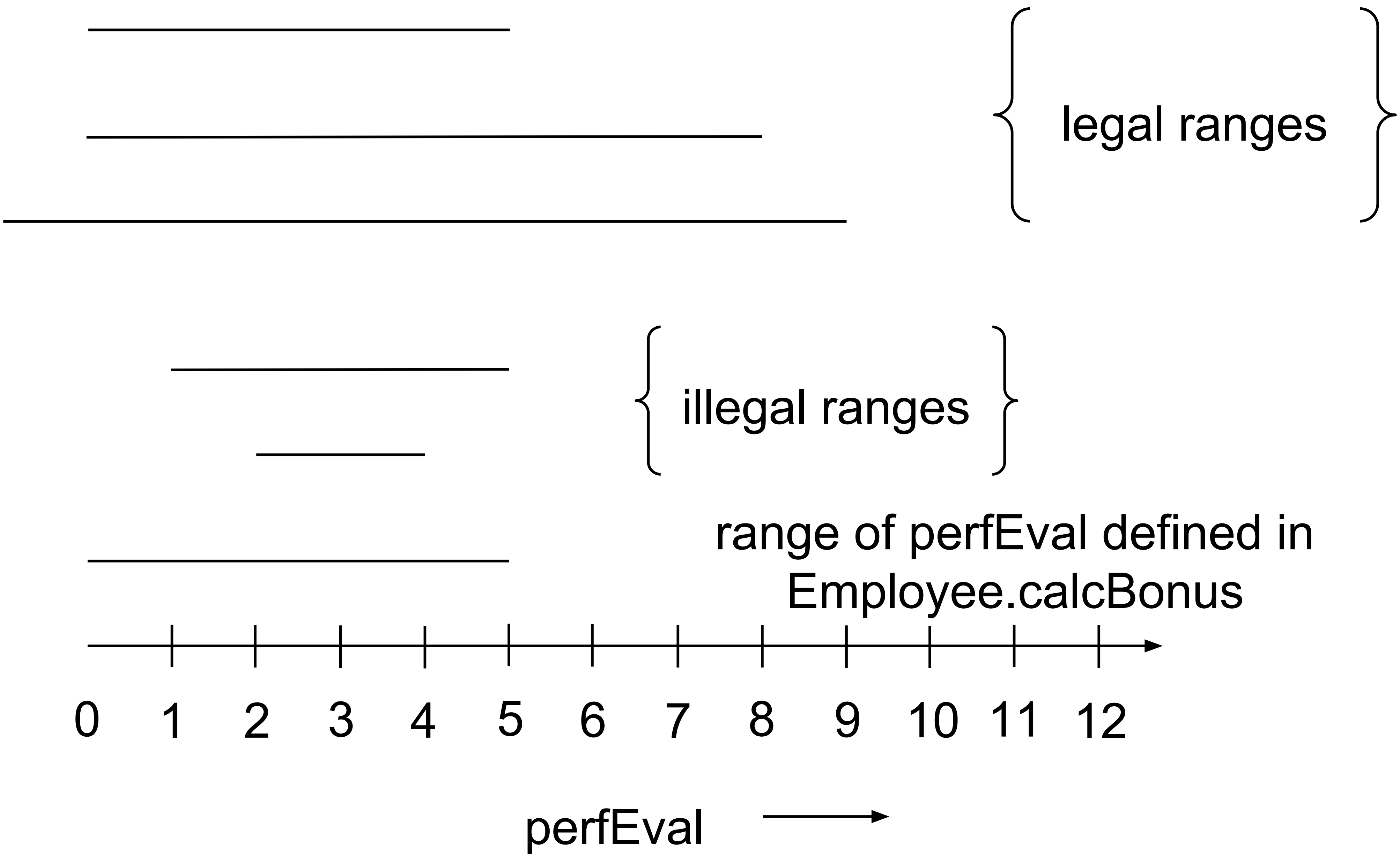
illegal ranges of perfEval

1 to 5

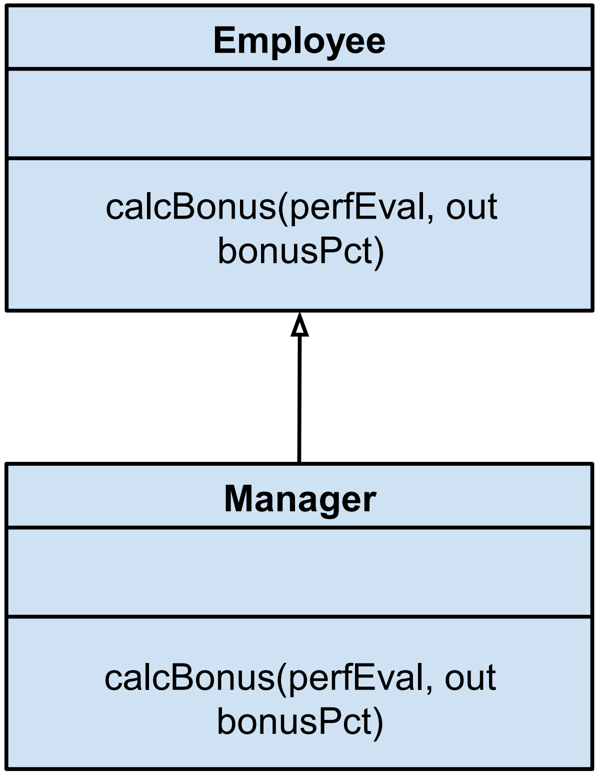
2 to 4

We will say for simplicity that perfEval is an integer between 0 and 5. The output argument bonusPct is between 0% and 10%.(in Employee Class)

An Example of Contravariance and Covariance-contd



An Example of Contravariance and Covariance-contd



legal ranges of bonusPct

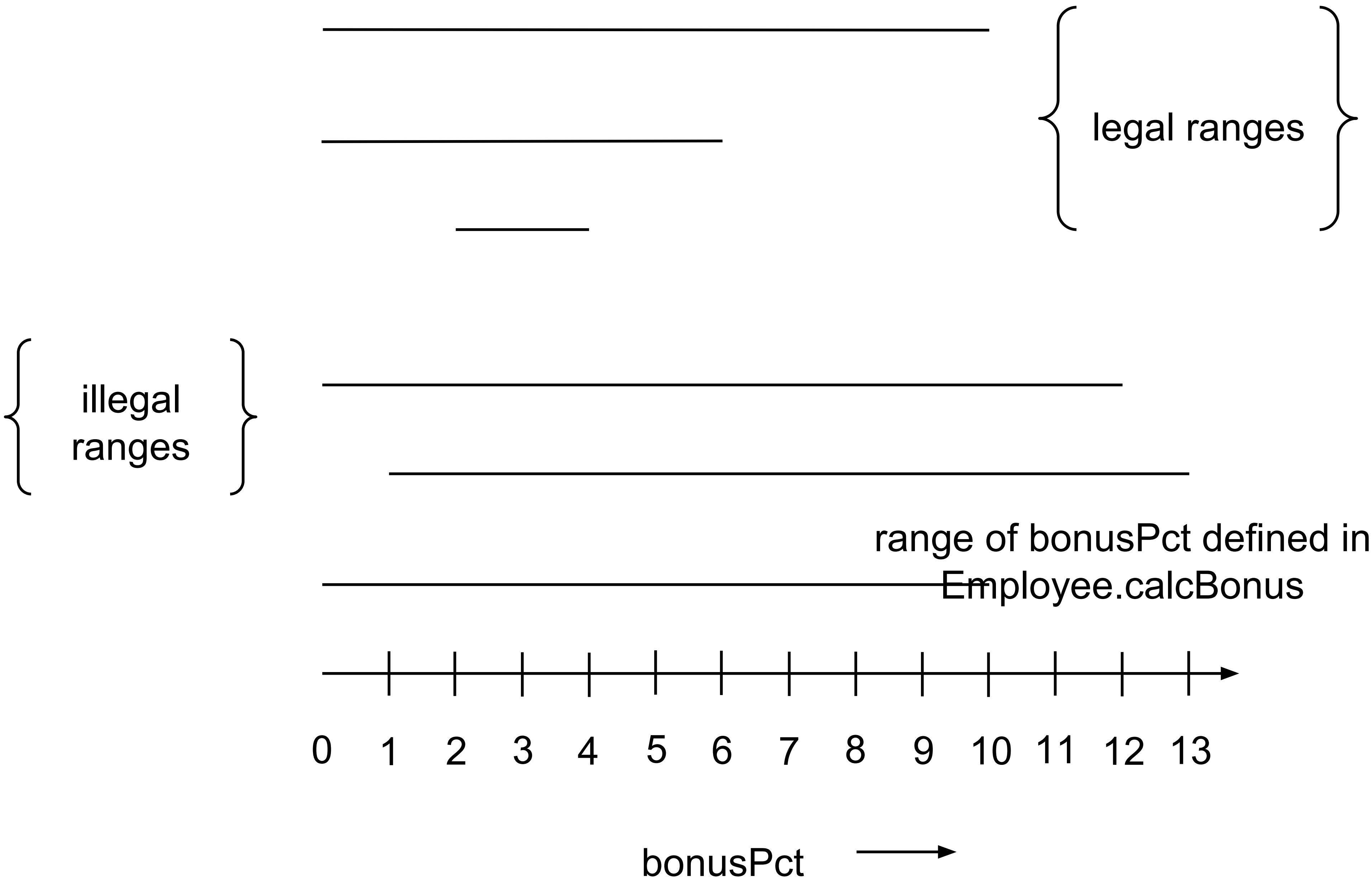
- 0% to 10%
- 0% to 6%
- 2% to 4%

illegal ranges of bonusPct

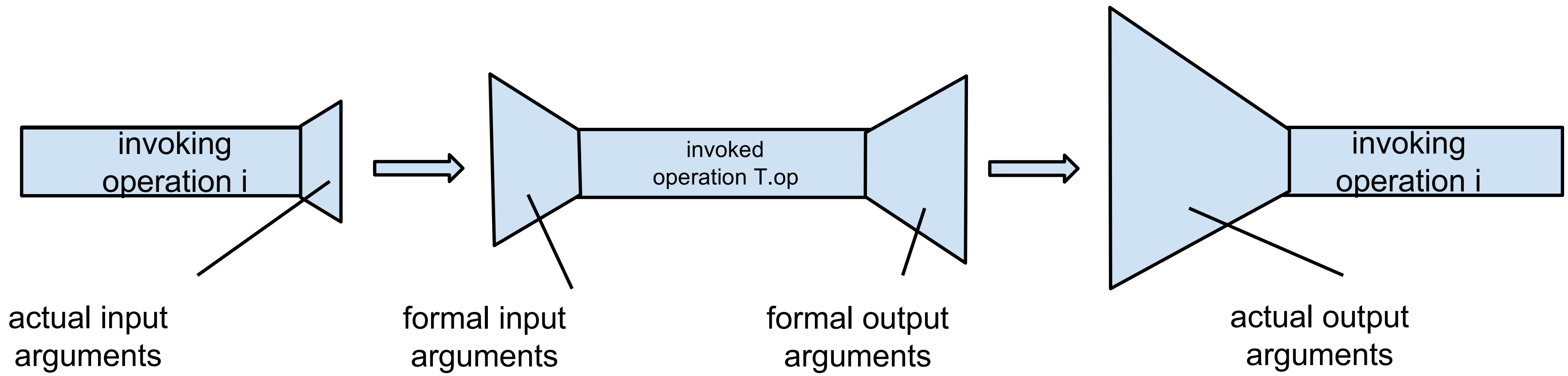
- 0% to 12%
- 1% to 13%

We will say for simplicity that perfEval is an integer between 0 and 5. The output argument bonusPct is between 0% and 10%.(in Employee Class)

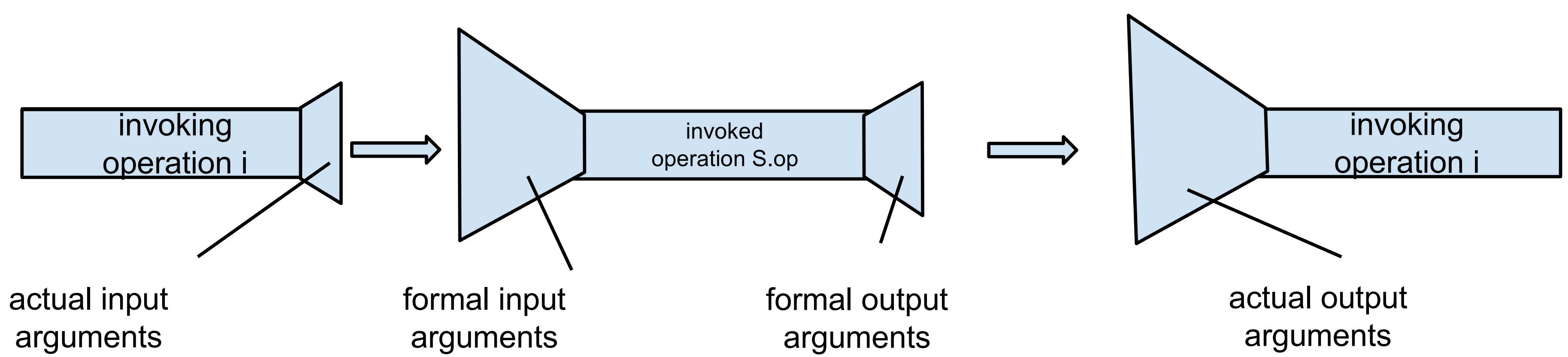
An Example of Contravariance and Covariance-contd



Graphic Illustration of Contravariance and Covariance



invoking an operation op in supertype T



invoking an operation op in subtype S

Summary of the Requirements for Type Conformance

The first two apply to whole classes; the last four apply to individual operations.

1. The state space of S must have the same dimensions as T. (But S may have additional dimensions that extend from T's state space).

2.The class invariant of S must be equal to or stronger than that of T.

Summary of the Requirements for Type Conformance -contd

For each operation of T (say T.op) that S overrides and redefines with S.op,

3. S.op must have the same name as T.op

4. S.op must have the same signature as T.op

5. The precondition of S.op must be equal to or weaker than that of T.op. (Principle of Contravariance)

6. The postcondition of S.op must be equal to or stronger than that of T.op. (Principle of Covariance)

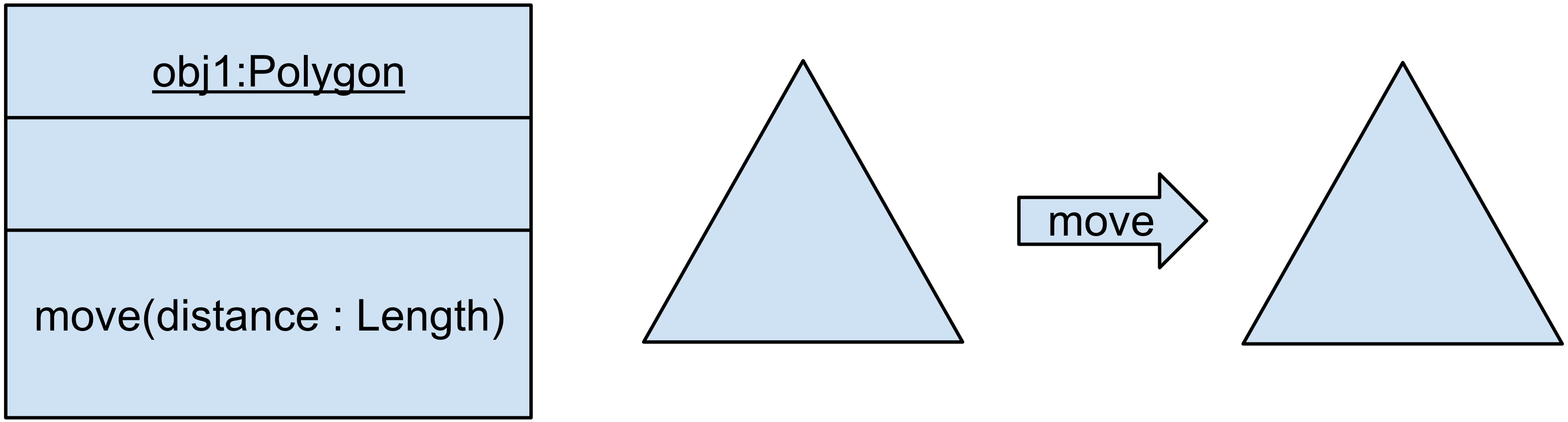
The Principle of Closed Behaviour

The principle of type conformance leads to sound designs only in read-only situations, that is , when accessor operations are executed.To handle situations in which modifier operations are executed, we also need the principle of closed behaviour.

defn

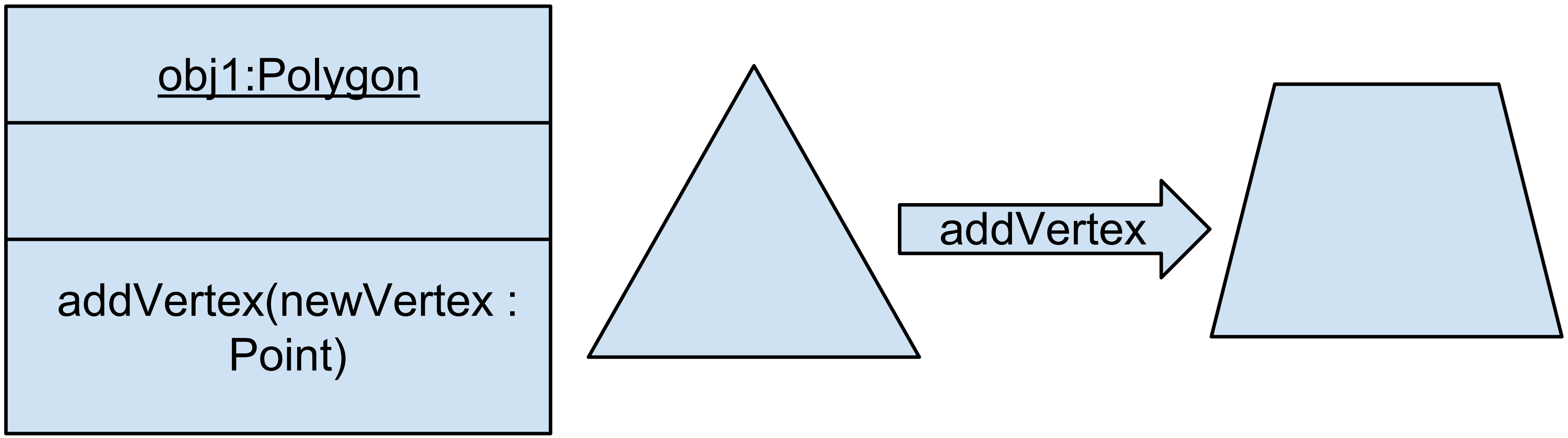
In an inheritance hierarchy based on a type/subtype hierarchy, the execution of any operation on an object of class C - including any operation inherited from C's super classes - should obey C's class invariant.

The Principle of Closed Behaviour-contd



The subclass Triangle is closed under the behaviour defined by the superclass's operation move().

The Principle of Closed Behaviour-contd



The subclass Triangle is not closed under the behaviour defined by the superclass's operation addVertex().
Here you must take one of the following corrective actions.
1. avoid inheriting addVertex().
2. overide addVertex() so that it has no effect (possibly also raising an exception).

Module V

- Abuses of Inheritance
- Danger of Polymorphism
- Mix-in Classes
- Rings of Operations
- Class Cohesion and Support of States and Behaviour

Module V

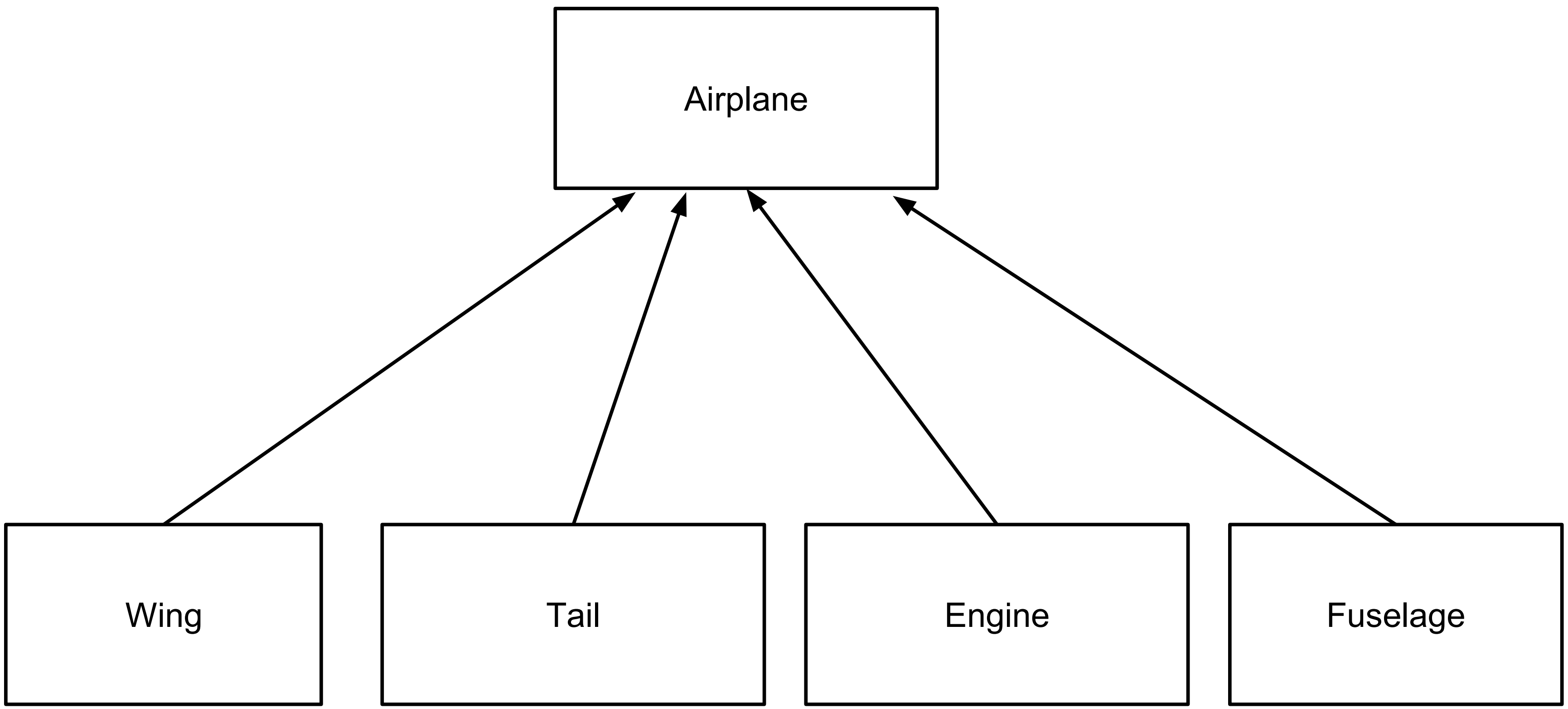
- Components and Objects
- Design of a Component
- Lightweight and Heavyweight Components
- Advantages and Disadvantages of using Components

Abuses of Inheritance

1. Mistaken Aggregates
2. Inverted Hierarchy
3. Confusing Class and Instance
4. Misapplying is a

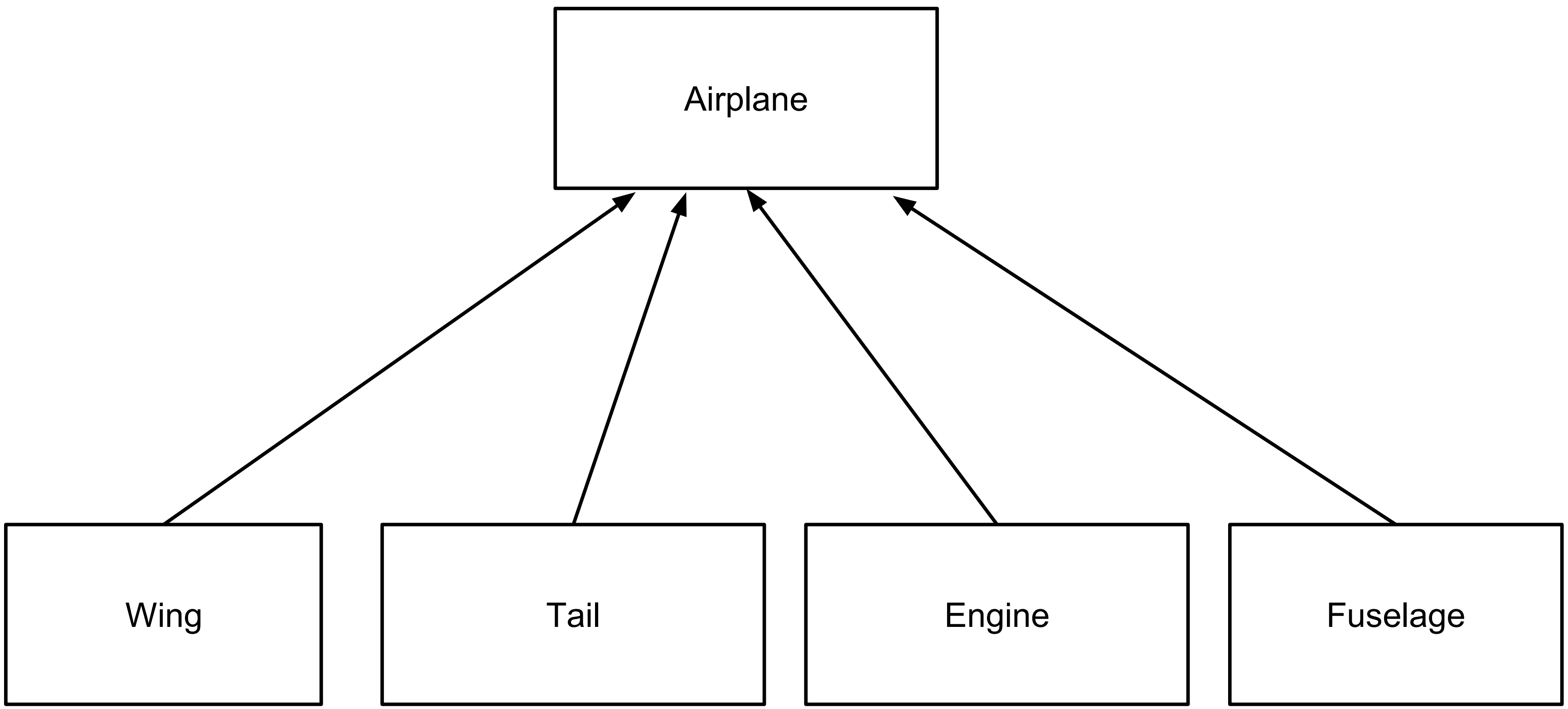
Abuses of Inheritance

- Mistaken Aggregates



Abuses of Inheritance

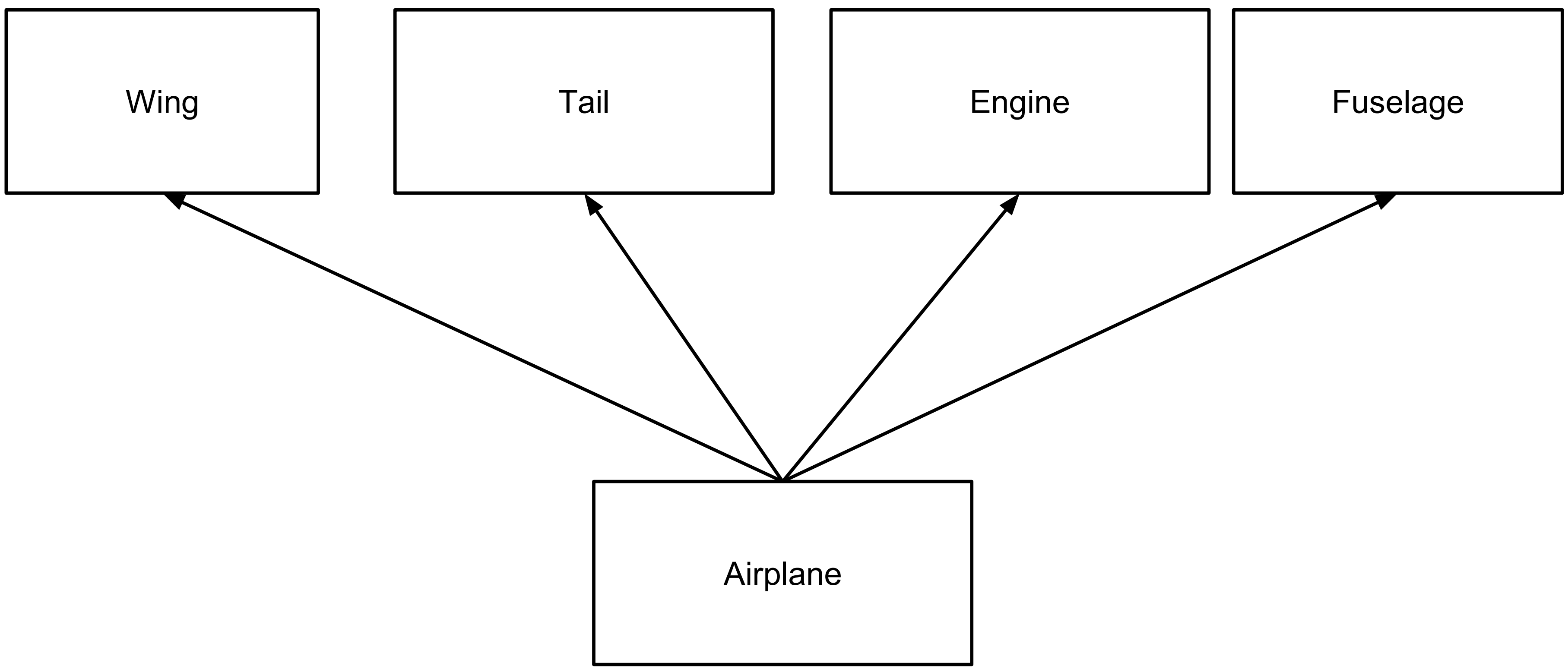
- Mistaken Aggregates



wrong representation

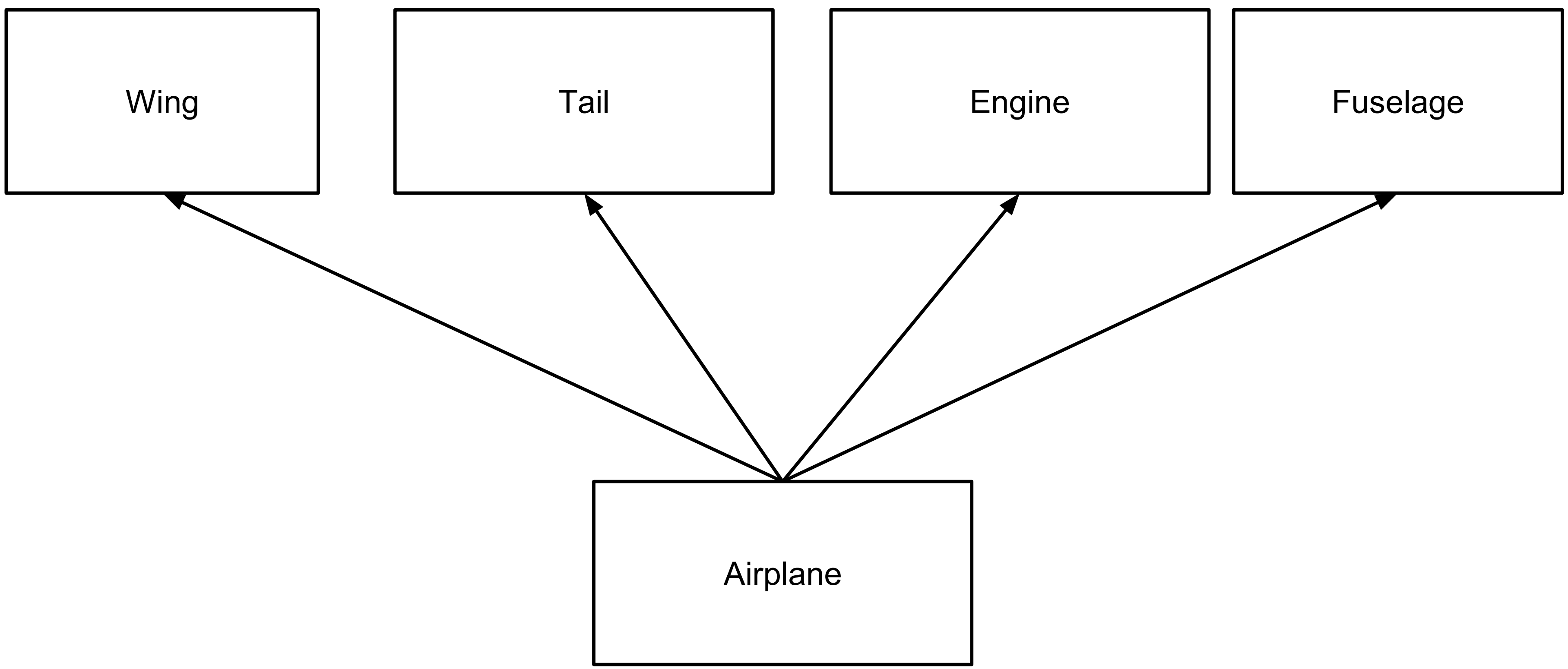
Abuses of Inheritance

- Mistaken Aggregates



Abuses of Inheritance

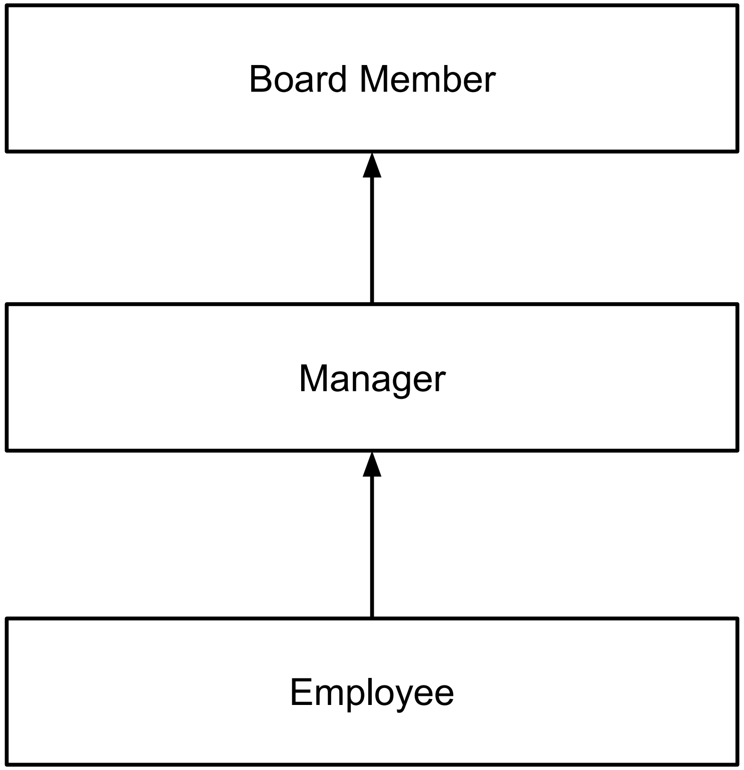
- Mistaken Aggregates



wrong representation

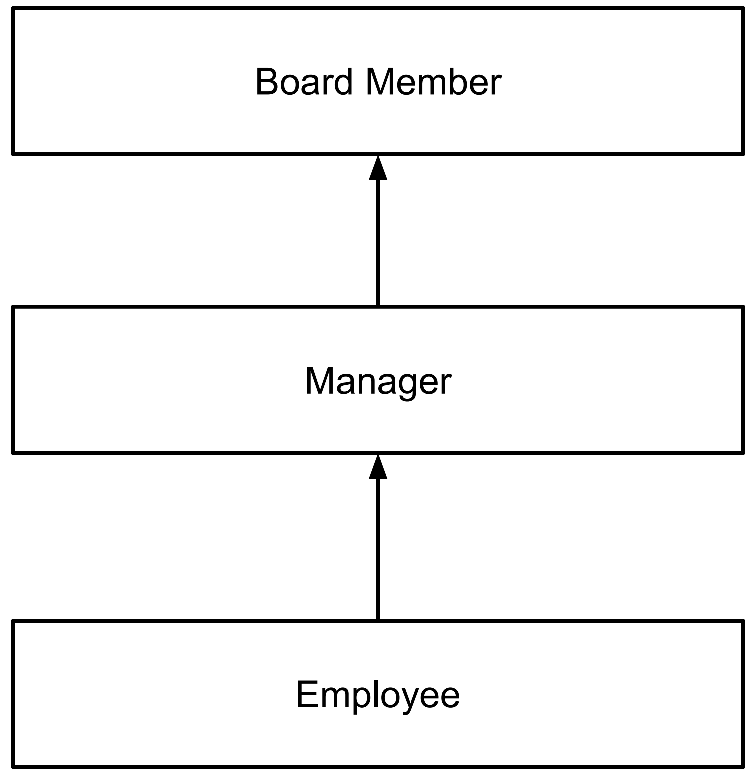
Abuses of Inheritance

- Inverted Hierarchy



Abuses of Inheritance

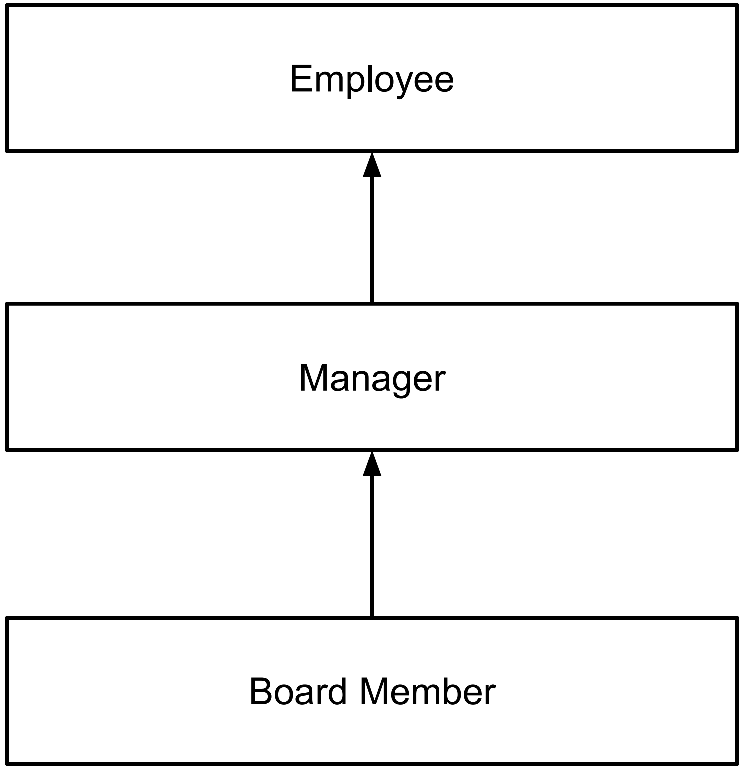
- Inverted Hierarchy



wrong representation

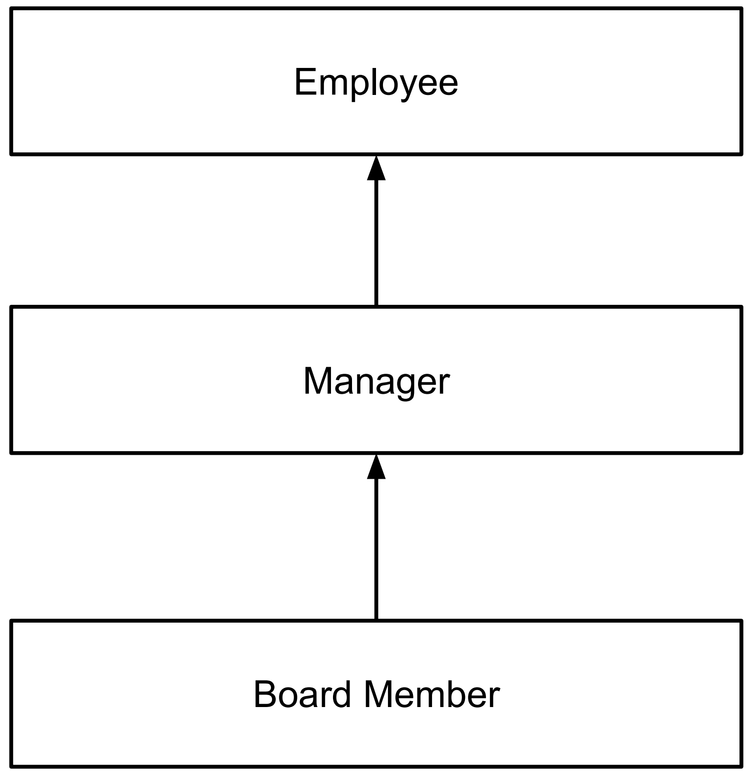
Abuses of Inheritance

- Inverted Hierarchy



Abuses of Inheritance

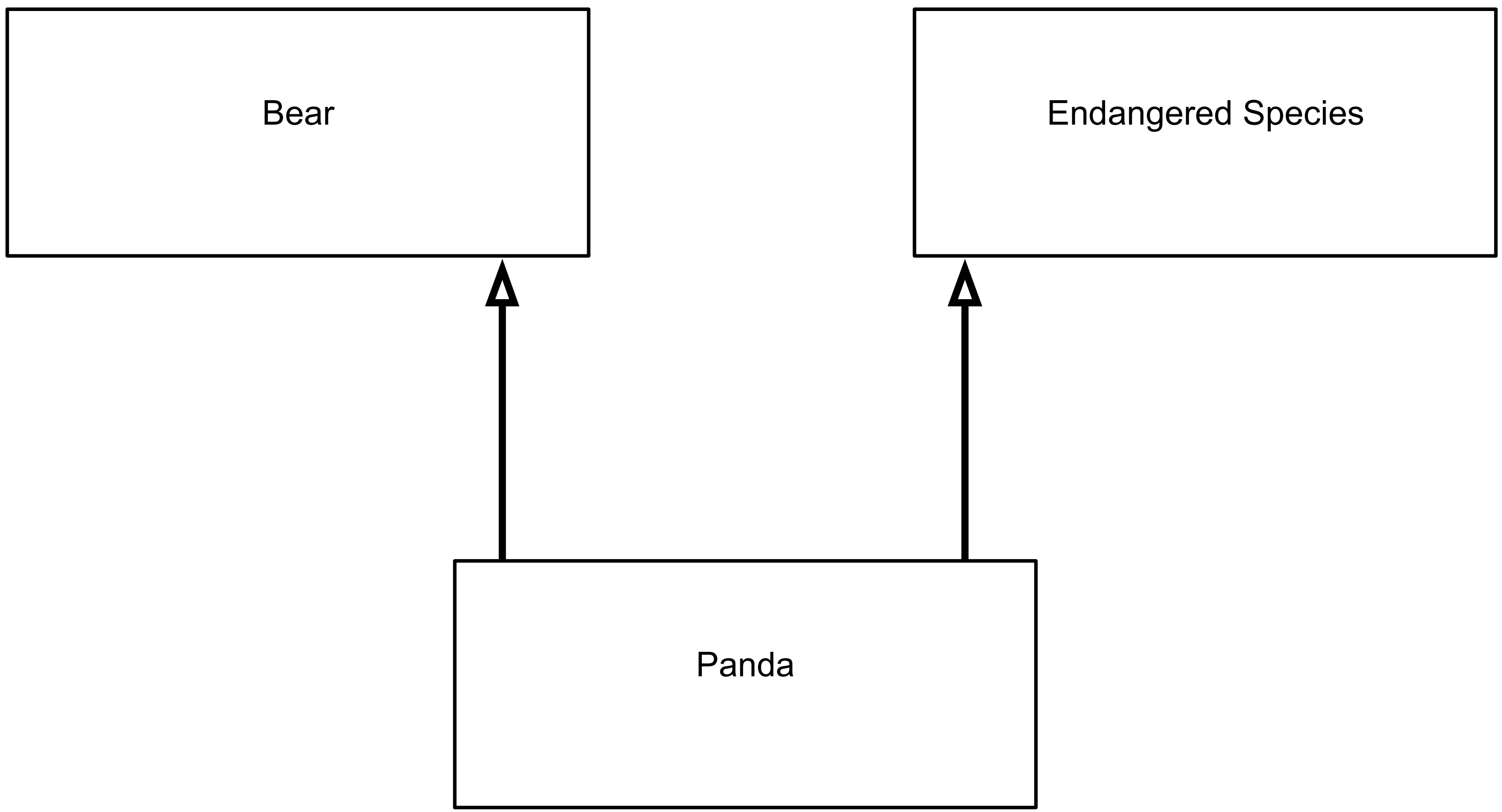
- Inverted Hierarchy



Correct Representation

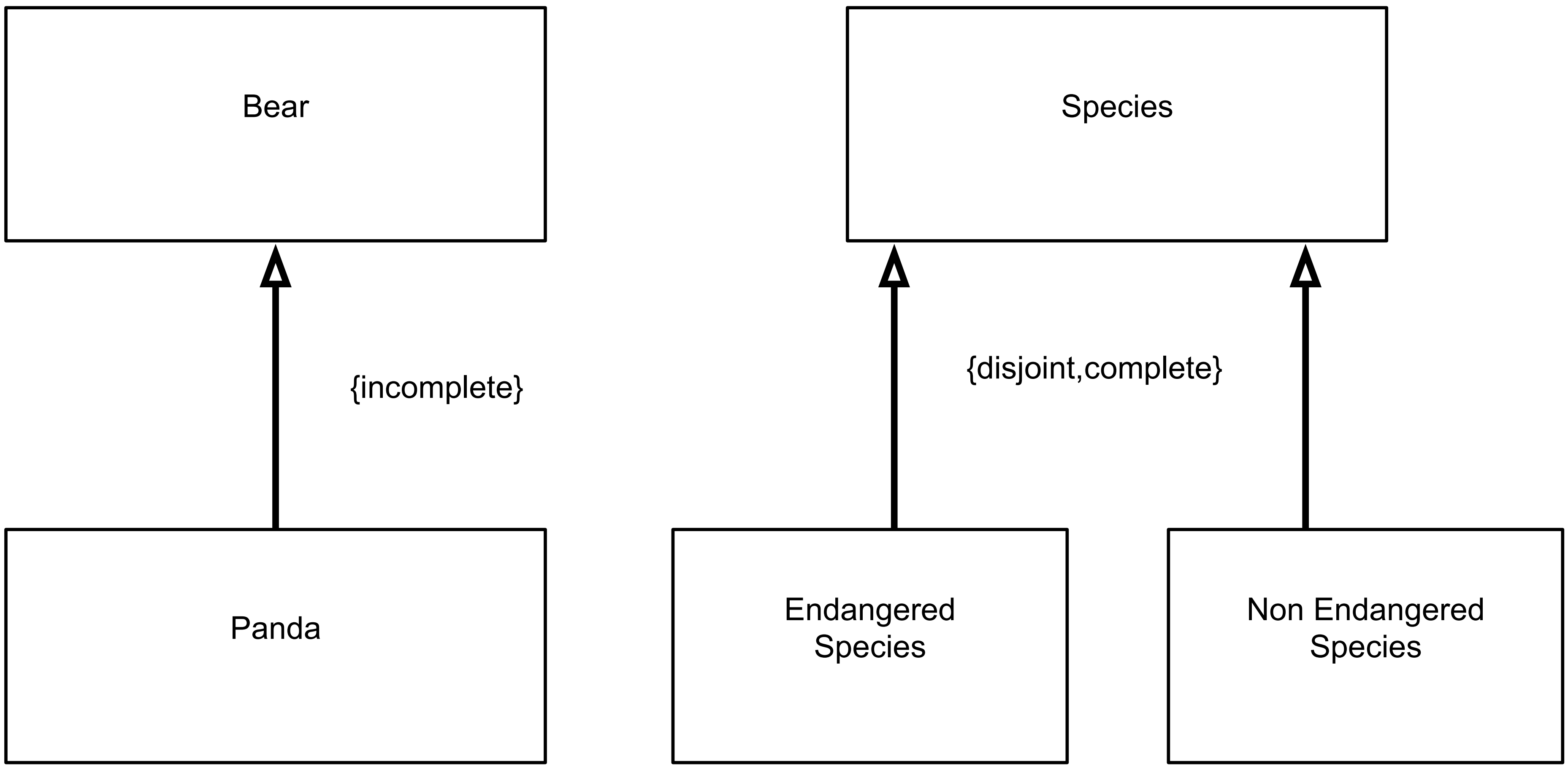
Abuses of Inheritance

- Confusing Class and Instance



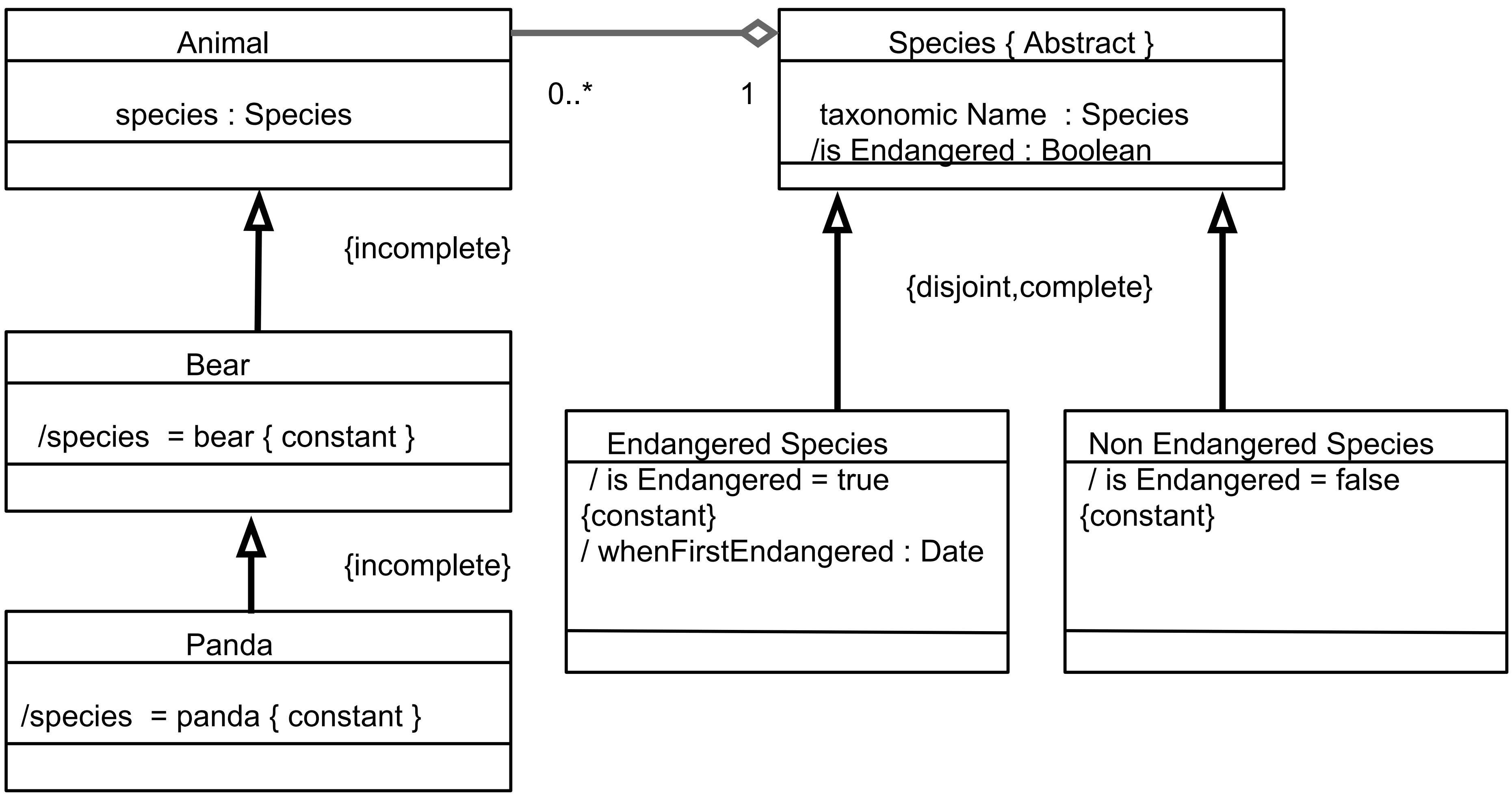
Abuses of Inheritance

- Confusing Class and Instance



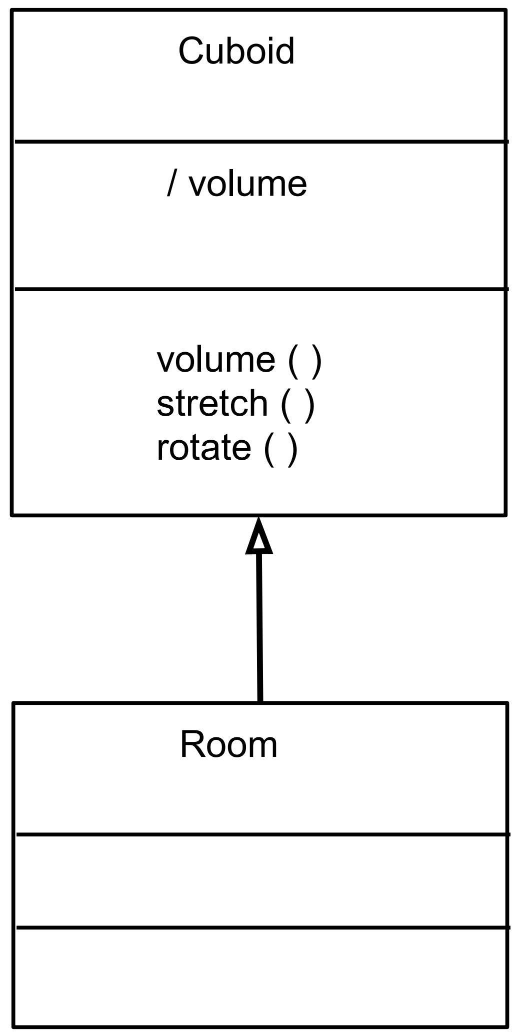
Abuses of Inheritance

- Confusing Class and Instance



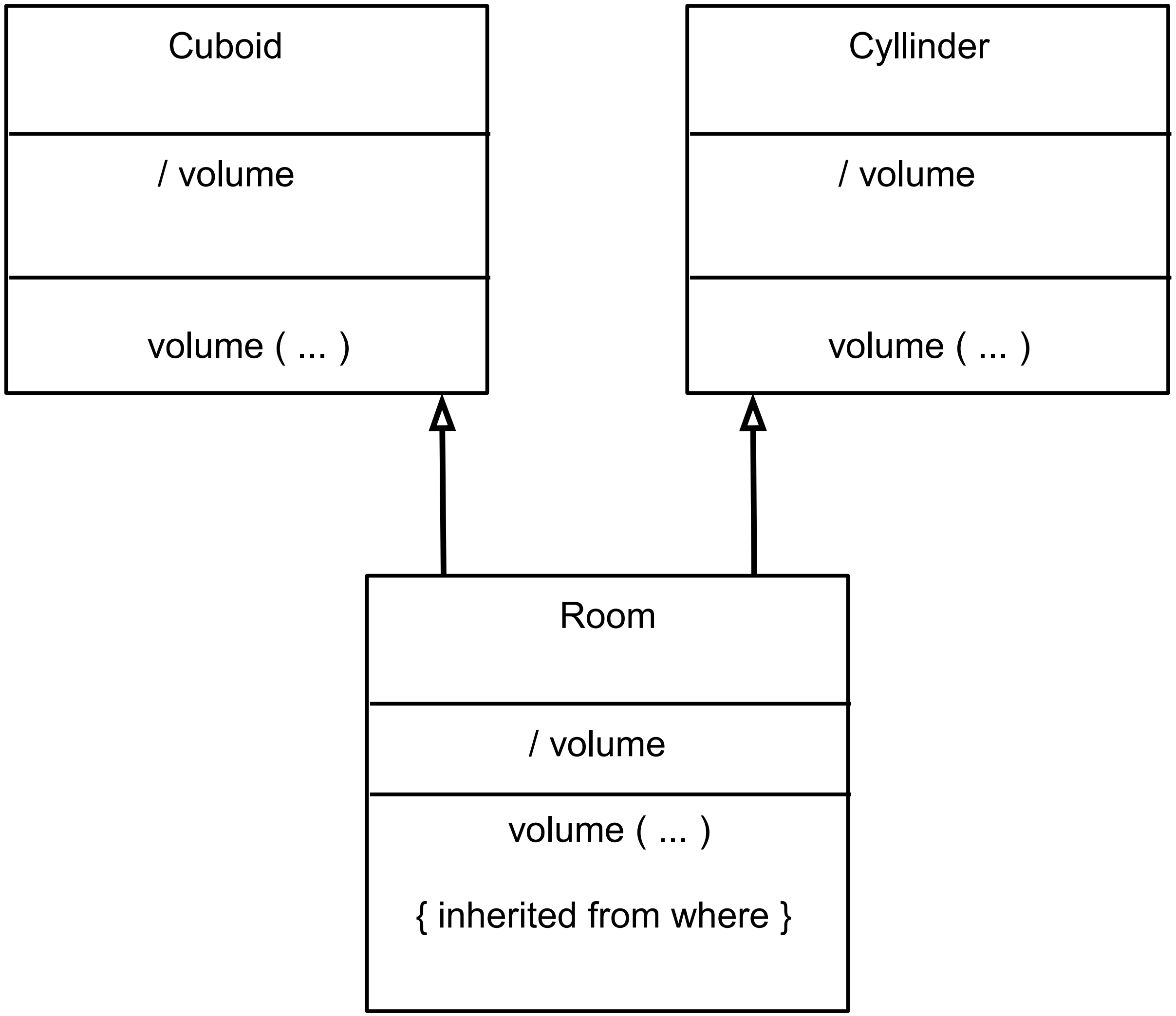
Abuses of Inheritance

- Misapplying is a



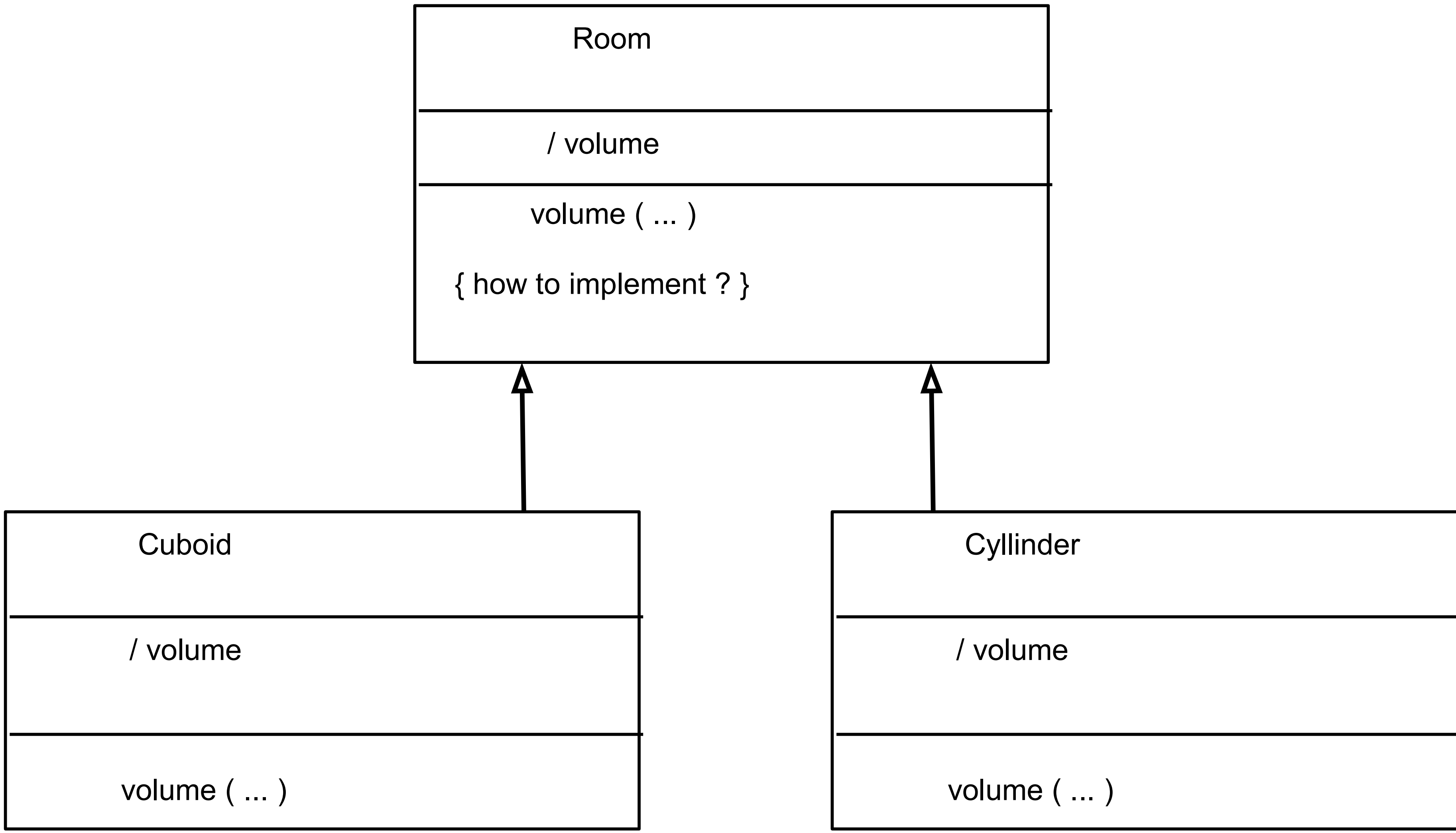
Abuses of Inheritance

- Misapplying is a



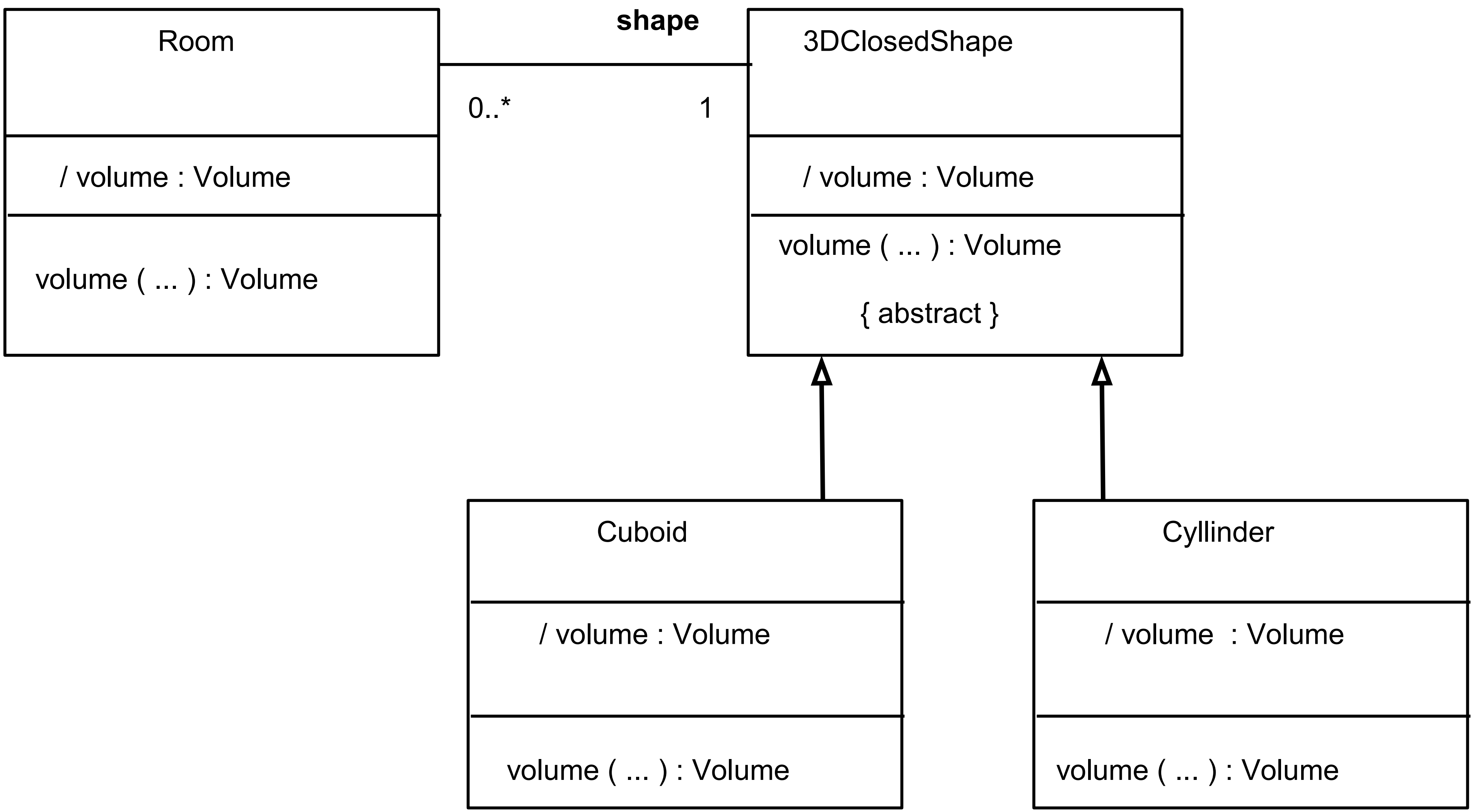
Abuses of Inheritance

- Misapplying is a



Abuses of Inheritance

- Misapplying is a



Abuses of Inheritance

- Misapplying is a

  - Room should also contain the declaration

    var shape :   3DClosedShape

  - At the initialisation of a particular room, the variable shape is assigned to an object of the correct shape for the given room

Abuses of Inheritance

- Misapplying is a
    - The operation Room.volume now works by asking the object pointed to by the variable shape to compute volume
    - This technique of accessing the code in another class is called message forwarding

Danger of Polymorphism

- Polymorphism of Operations
- Polymorphism of Variables
- Polymorphism in Messages
- Polymorphism and Genericity

Danger of Polymorphism

- Polymorphism of Operations

  Defn - Scope of Polymorphism (SOP)

  The scope of polymorphism of an operation op is the set of classes upon which op is defined.

Danger of Polymorphism

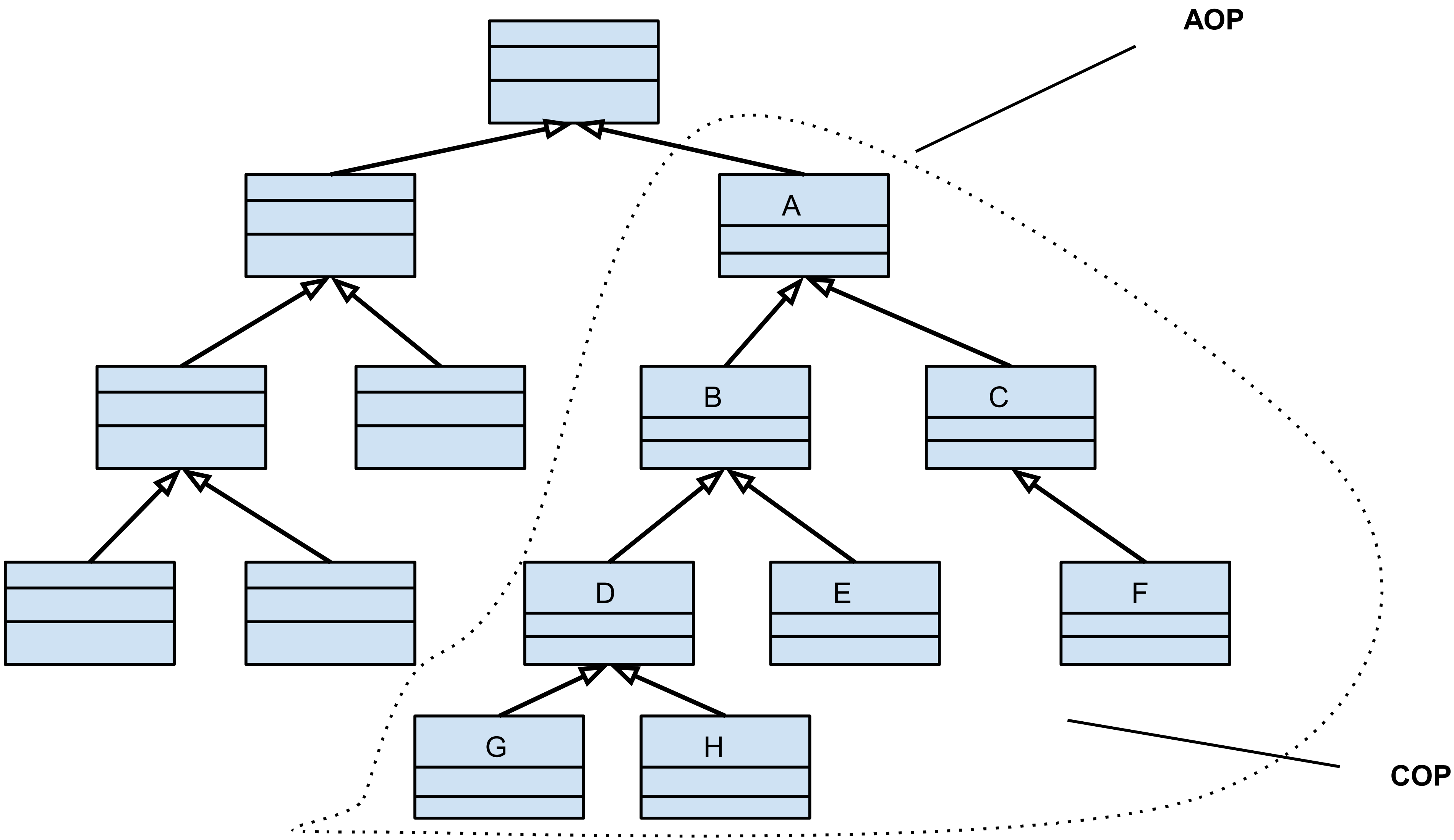
- Polymorphism of Operations

  Defn - Cone of Polymorphism (COP)

  A Scope of Polymorphism that forms a branch of the inheritance hierarchy - that is , a class A together with all of its subclasses - is termed a Cone of Polymorphism, with A as the Apex of Polymorphism (AOP)

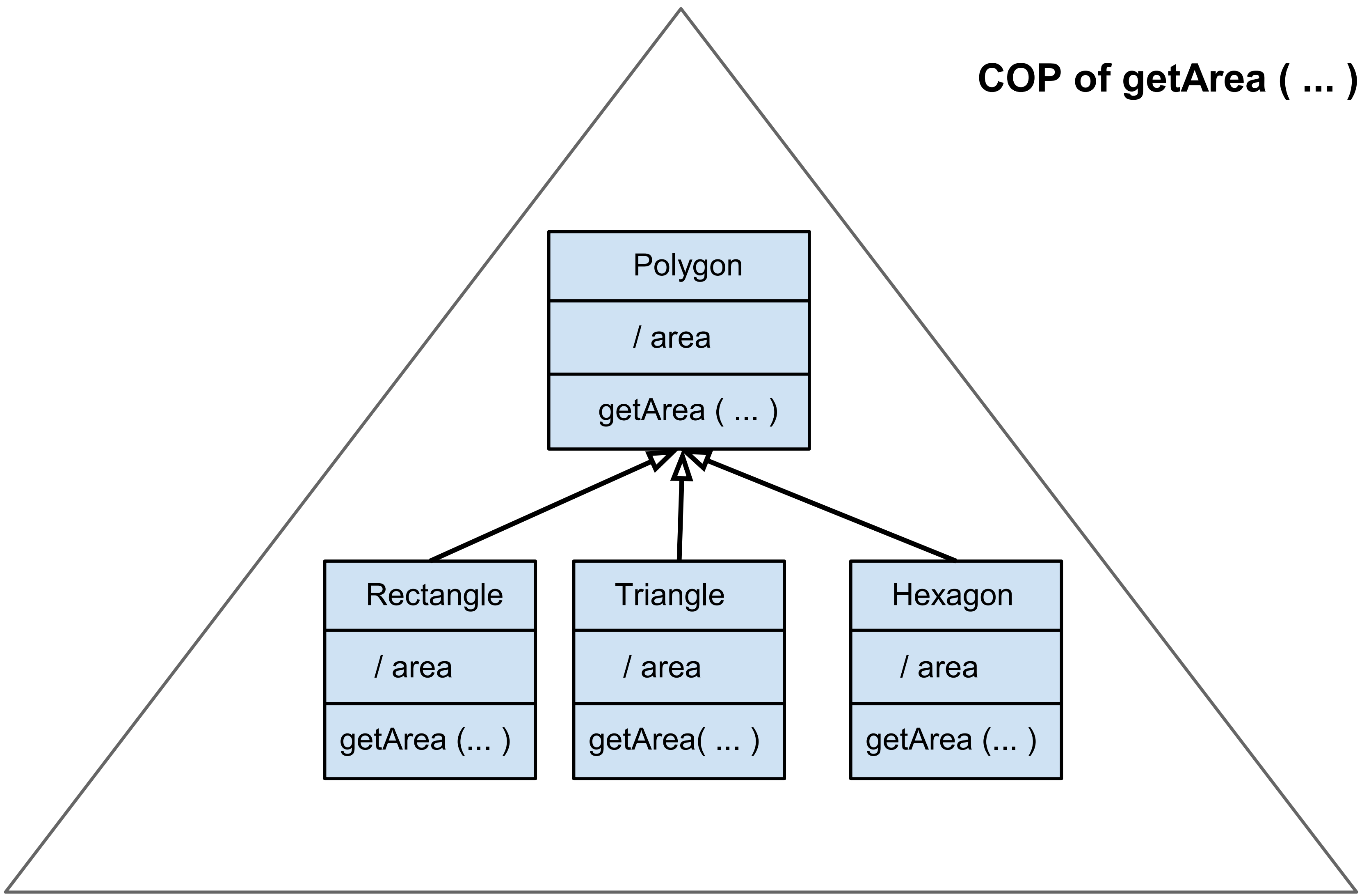
Danger of Polymorphism

- Polymorphism of Operations



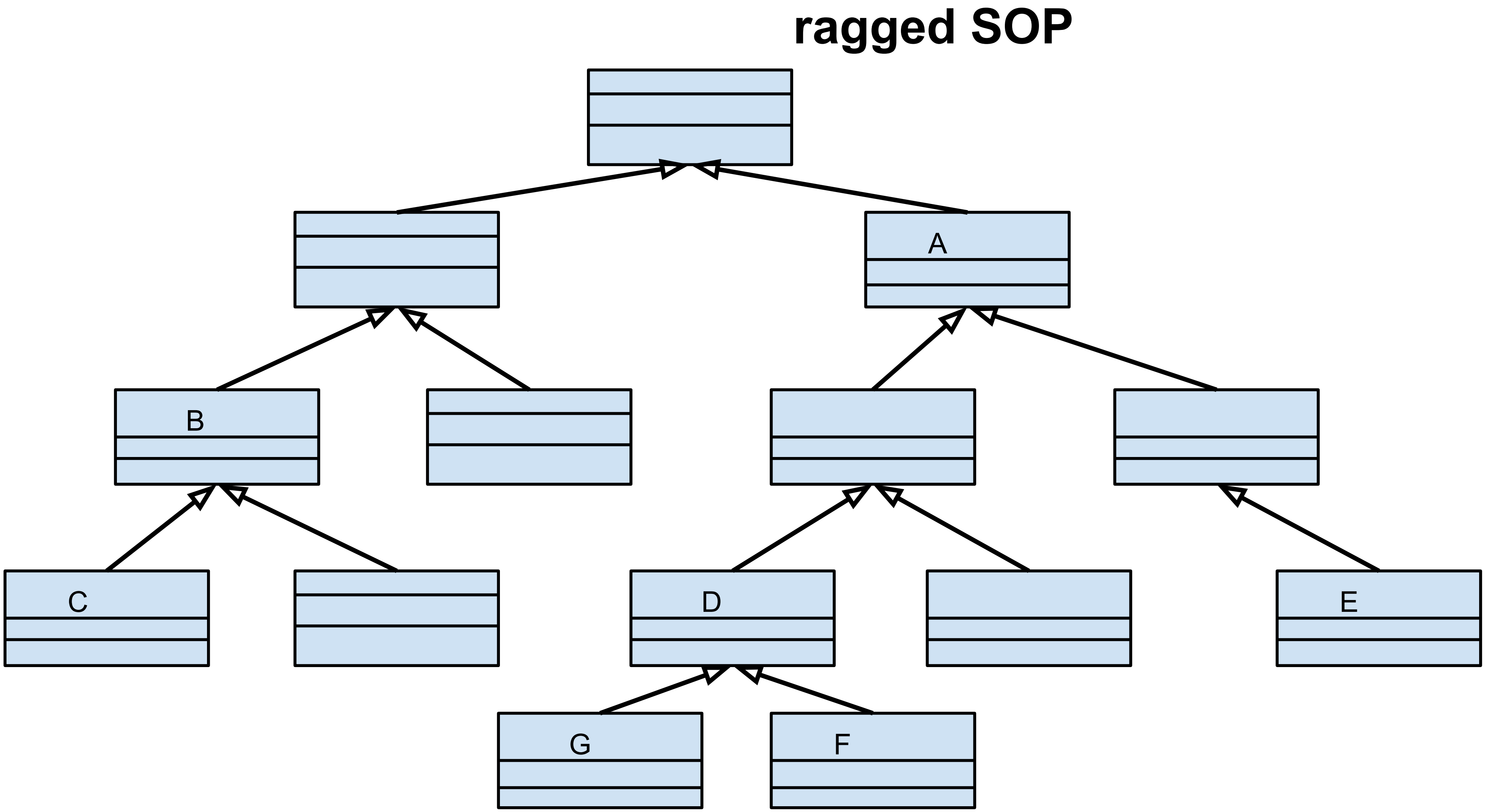
Danger of Polymorphism

- Polymorphism of Operations



Danger of Polymorphism

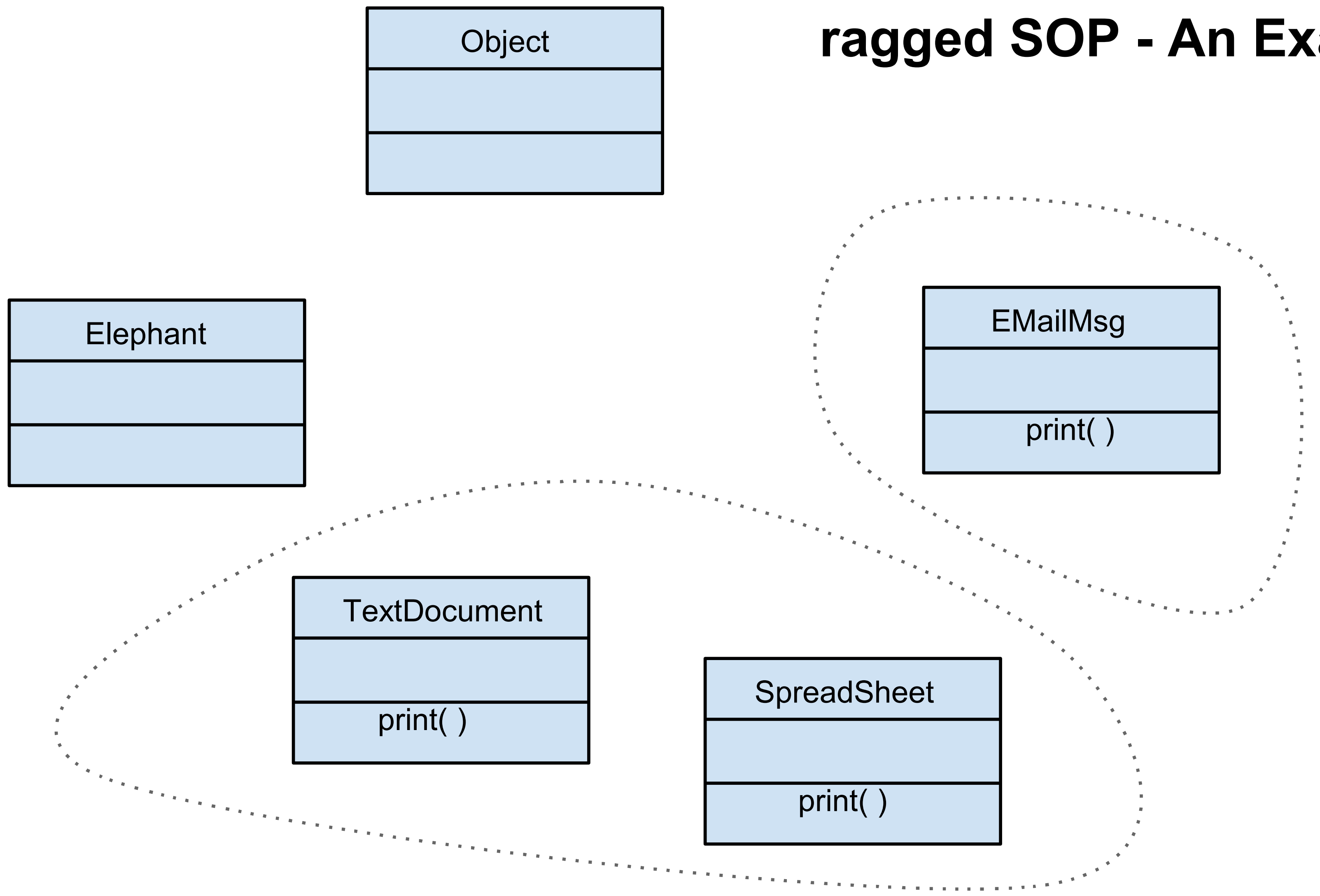
- Polymorphism of Operations



Danger of Polymorphism

- Polymorphism of Operations

ragged SOP - An Example



Danger of Polymorphism

- Polymorphism of Variables

Defn - Scope of Polymorphism

The scope of polymorphism of a variable v is the set of classes to which objects pointed to by v (during v's entire life time) may belong.

Danger of Polymorphism

- Polymorphism of Variables

Examples

- Let us say that the declaration

  var t : Triangle

  allows the variable t to point to any object of class Triangle or of Triangle's descendents.
- In this example, therefore , the variable's SOP forms a cone, with the class Triangle as the apex.

Danger of Polymorphism

- Polymorphism of Variables

Examples

- Let us say that a variable v is allowed, at various times , to point to an object of class Horse, Circle, or Customer.
- In this example, then, the variable's SOP is not a cone in that these classes do not have a common intermediate superclass to form an AOP.

Danger of Polymorphism

- Polymorphism of Variables

  Examples

- Let us say that we have a declaration

  var x : Object

  where the class Object is the top of the class hierarchy.
- This time, the variable's SOP does form a Cone.
- Indeed, this Cone is the largest one of all, because its apex is the top class in the class hierarchy.

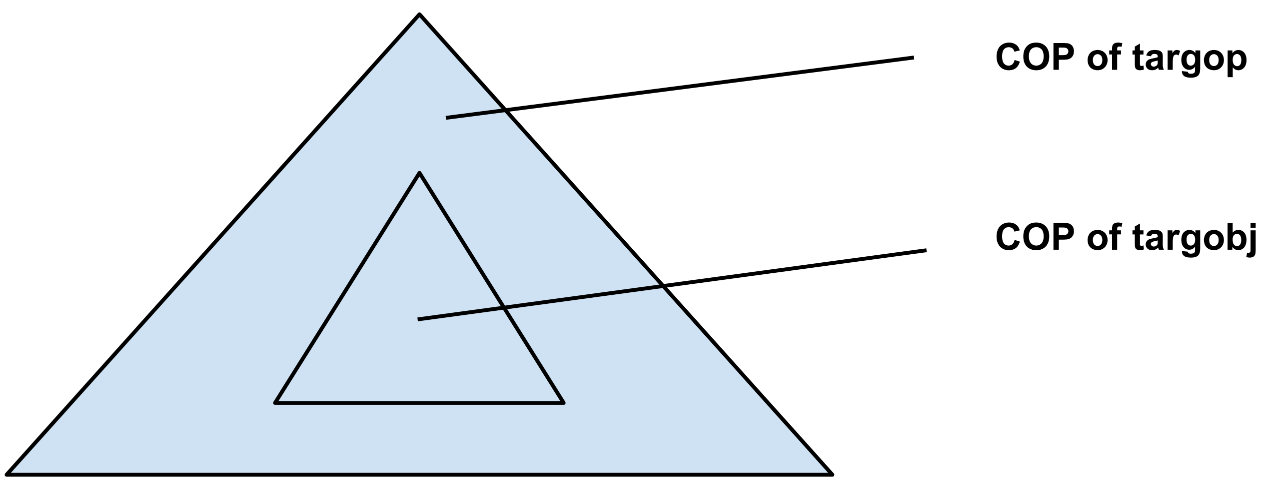
Danger of Polymorphism

- Polymorphism in Messages
  - A message is composed of a variable that points to the target object and an operation name that states the operation to be invoked.
  - Both the variable and the operation have an SOP
  - The relationship between these two SOPs has a significant impact on system reliability

Danger of Polymorphism

- Polymorphism in Messages

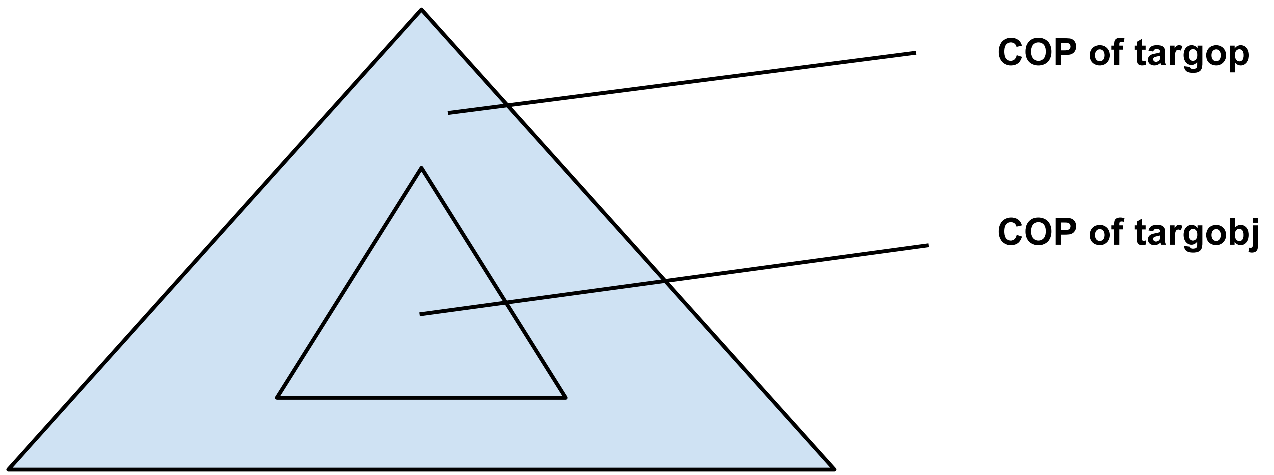
  case 1



Danger of Polymorphism

- Polymorphism in Messages

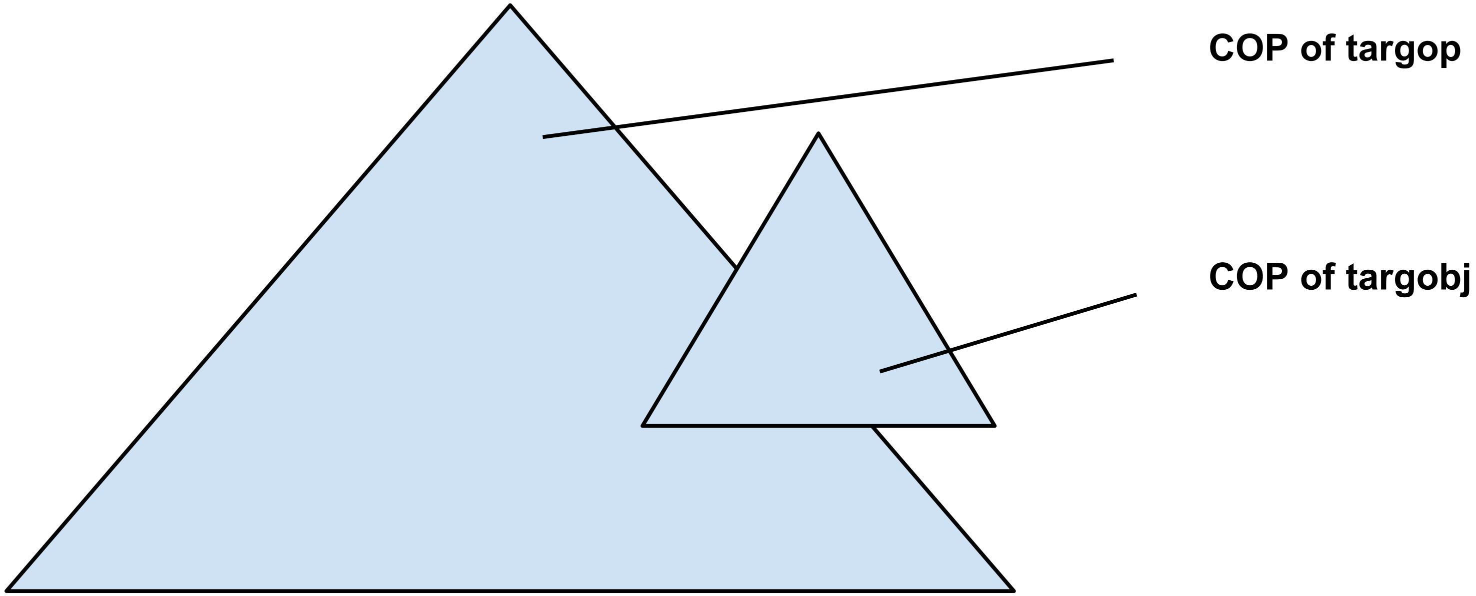
case 1 good design



Danger of Polymorphism

- Polymorphism in Messages

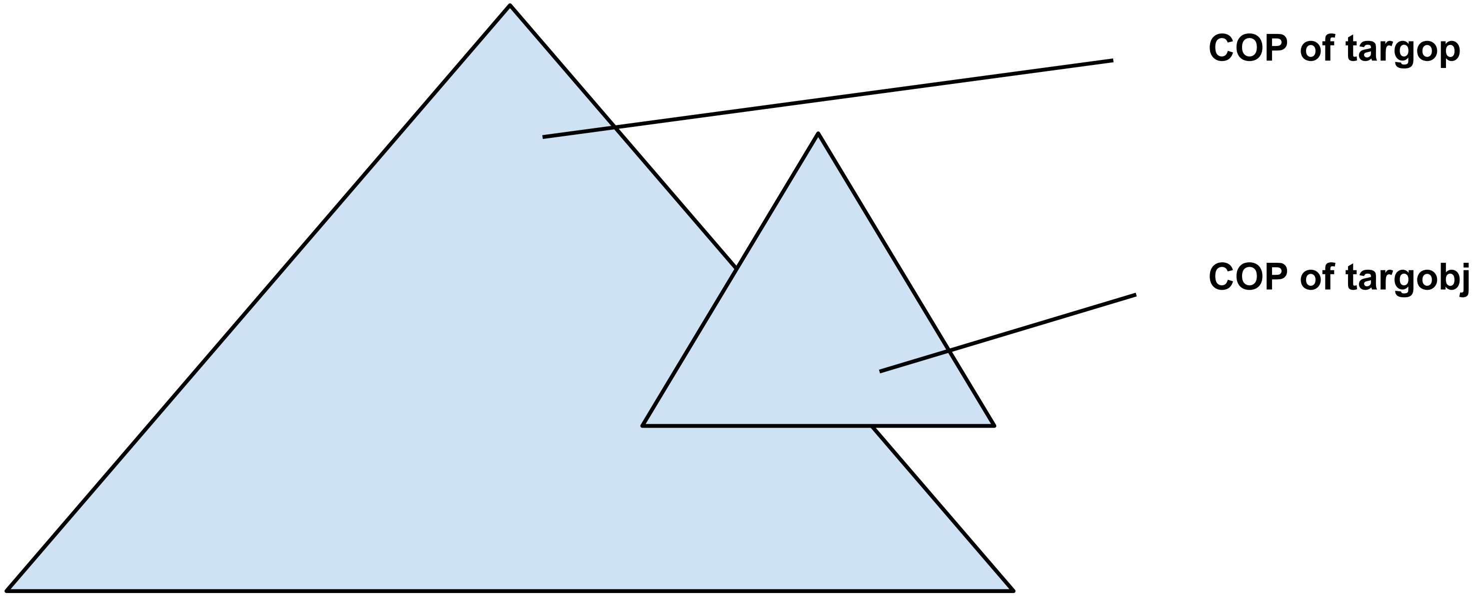
  case 2



Danger of Polymorphism

- Polymorphism in Messages

case 2 miserable , non-robust design



Danger of Polymorphism

- Polymorphism in Messages

  Example

  Consider the message

  factoryDevice . SwitchOn

Danger of Polymorphism

- Polymorphism in Messages

  Case 1

- factoryDevice always points to an object of class Tap, Motor, or Light , all of which can switch on.

- Then, factoryDevice's SOP is within SwitchOn's SOP and everything is fine.

Danger of Polymorphism

- Polymorphism in Messages

  Case 2

- factoryDevice refers to any piece of hardware in the factory, including objects of class Tap, Motor, Light, Pipe, Tank, Door, Lever and so on.
- Not all of these devices " know how to switch on ".

Danger of Polymorphism

- Polymorphism in Messages

  Case 2

- This time, therefore, much of FactoryDevice's SOP falls outside SwitchOn's SOP.

- There is a significant chance of a run-time problem, when, for instance, a plain old door is told to switch on.

Danger of Polymorphism

- Polymorphism and Genericity
  - A parameterised class (template class in C++) is a class that takes a class name as an argument whenever one of its objects is instantiated
  - Designers often use parameterised classes to construct containers such as lists, stacks and sorted trees

Danger of Polymorphism

- Polymorphism and Genericity

  - Consider the parameterised class

    Sorted Tree < NodeClass>

Danger of Polymorphism

- Polymorphism and Genericity
    - The following statements instantiates specific sorted trees

```
RealNumTree := SortedTree<Real>.New
CustTree    :=      SortedTree<Customer>.New
FuselageTree :=  SortedTree<Fuselage>.New
ComplexTree := SortedTree<Complex>.New
AnimalTree := SortedTree<Animal>.New
```

Danger of Polymorphism

- Polymorphism and Genericity
  - The scope of NodeClass is unlimited
  - The SOP of operations within SortedTree is actually very small
  - It is the intersection of the SOPs of the individual operations (such as print, lessThan and so on)

Danger of Polymorphism

- Polymorphism and Genericity
  - There are two solutions to this design problem
  - The first solution is that every user of a parameterised class should ensure that the actual runtime class provided is within the SOP intersection of individual operations

Danger of Polymorphism

- Polymorphism and Genericity
  - The second solution is to provide some kind of guard at the beginning of the parameterised class' code
  - The guard checks that the actual class supplied can understand the required messages
  - This method is supported in Eiffel programming language

Danger of Polymorphism

- Polymorphism and Genericity
  - For example, you would first write at the top of a parameterised class

    NodeClass -> Printable

    where Printable is the AOP.

Danger of Polymorphism

- Polymorphism and Genericity
    - This means that the parameterised class will only accept the class Printable or one of its descendents as a supplied runtime class
    - Next, you would design a class called Printable with an operation called print (which should be an abstract operation)

Danger of Polymorphism

- Polymorphism and Genericity
  - Now, since everyone who supplies a class to SortedTree has to supply a descendent class of Printable, the supplied class is guaranteed to have the operation print defined on it
  - Similarly, the class Comparable might be one with the operations lessThan, greaterThan and equal To defined on it

Mix-in Classes

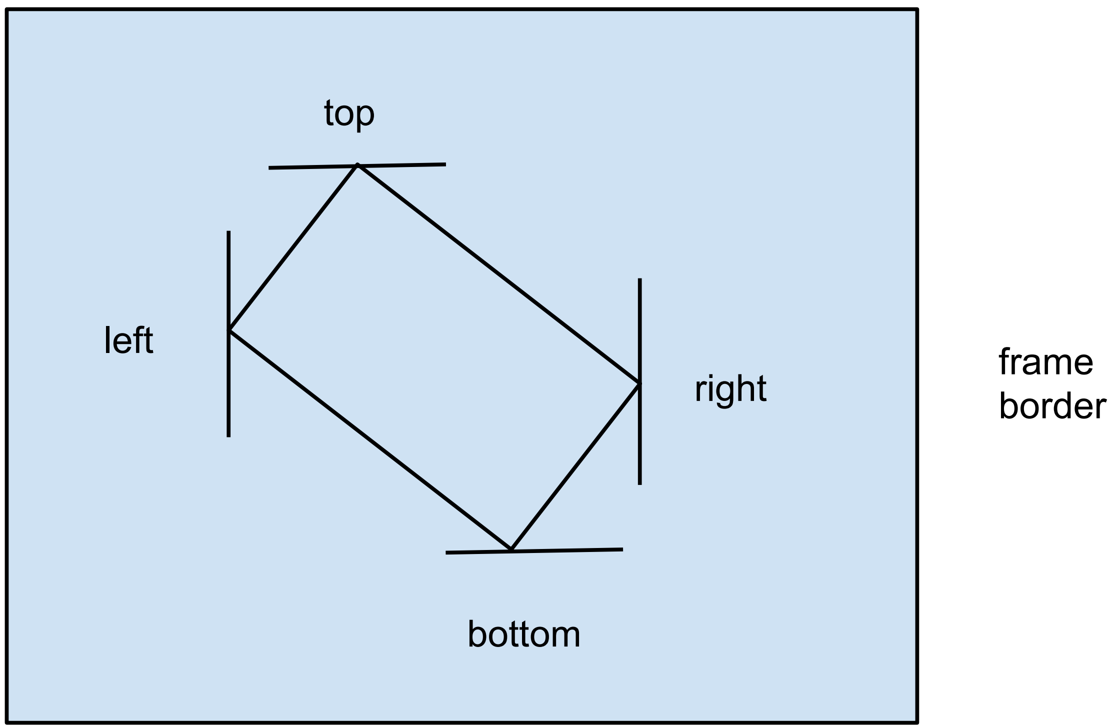
- They are used for increasing the reusability in an object oriented system

  defn

- A class from which objects are not normally instantiated, but which is designed to have its capabilities inherited by ("mixed in" with) other classes

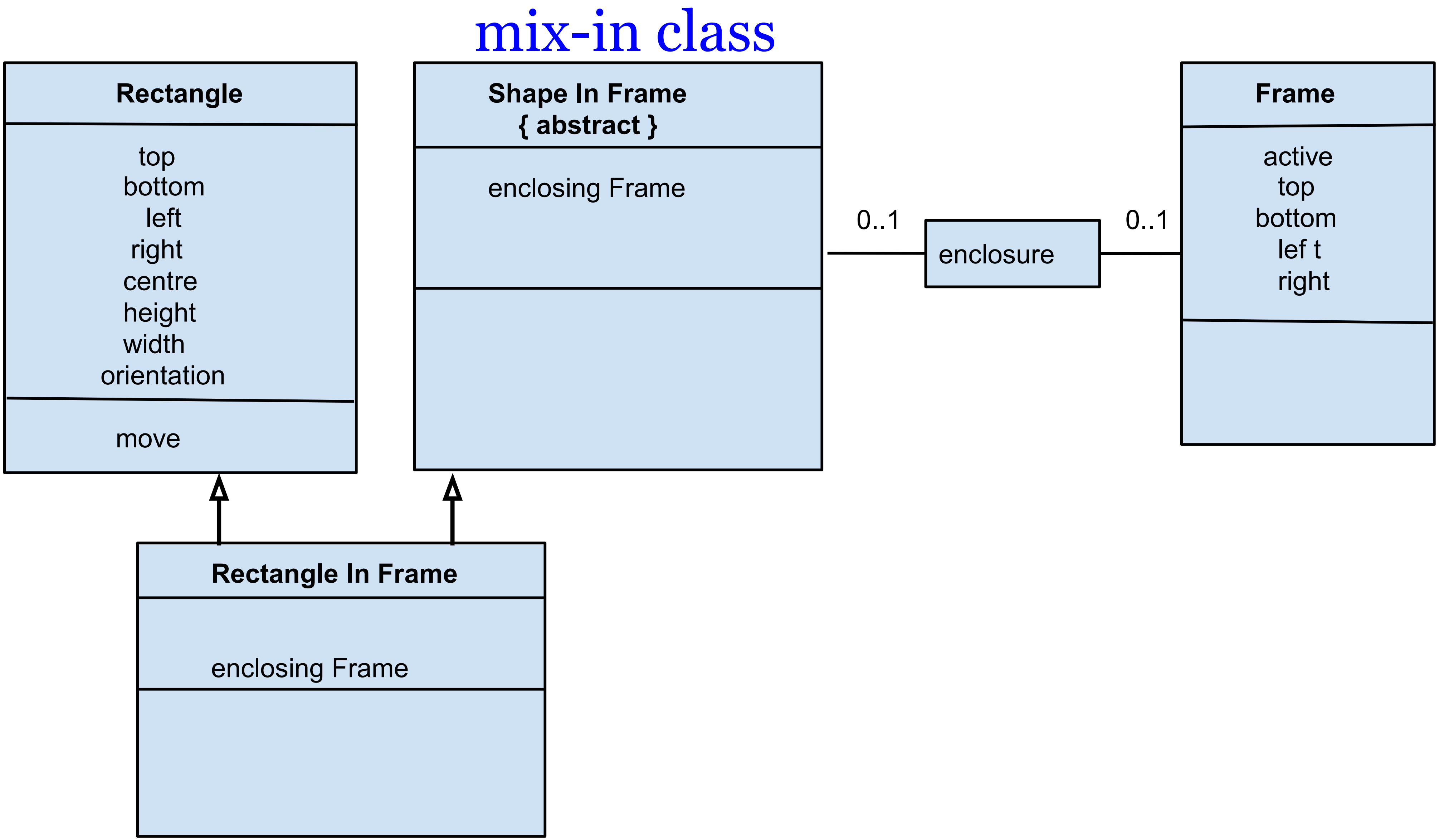
Mix-in Classes

Example



Mix-in Classes

Class Diagram

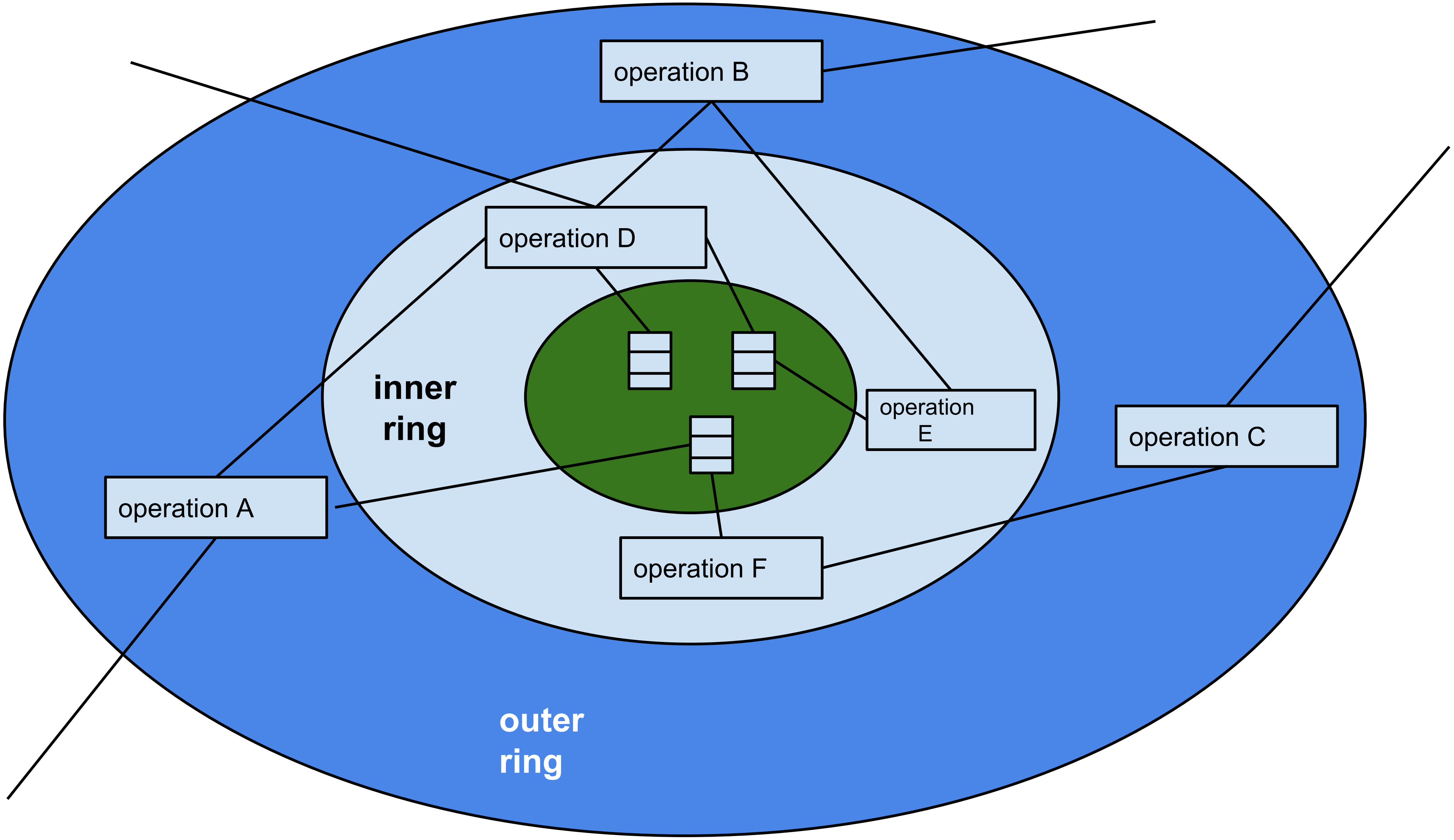


Mix-in Classes

- If we use Rectangle class to store information about the frame in which it is enclosed , we are reducing its reusability in other applications

- Similarly we can mix Triangle class with Shape In Frame to produce Triangle In Frame, Ellipse class with Shape In Frame to produce Ellipse In Frame etc

Rings of Operations



Rings of Operations

- Advantages
  - It may avoid duplication of code in the two operations
  - It limits the knowledge of some variables' representations to fewer operations
  - If one of the operations is in a subclass, then sending a message - rather than directly manipulating the superclass' variables - decreases the connascence between the two classes

Class Cohesion and Support of States and Behaviour

- State Support in a Class Interface
- Behaviour Support in a Class Interface
- Operation Cohesion in a Class Interface

State Support in a Class Interface

- Illegal States
- Incomplete States
- Inappropriate States
- Ideal States

State Support in a Class Interface

- Illegal States
  - A class interface that allows illegal states enable an object to reach states that violate that object's class invariant

State Support in a Class Interface

- Illegal States

  example 1

  Consider an operation movePoint defined on Rectangle that allows a single corner of a rectangle to be moved independently of the other corners

State Support in a Class Interface

- Illegal States

  example 2

  Consider a rectangle implemented by lines for its sides, with each of these lines directly manipulable

State Support in a Class Interface

- Incomplete States
    - In a class interface design with incomplete states, there are legal states in Rectangle's state space that an object cannot reach

State Support in a Class Interface

- Incomplete States

  example

  Consider a poor design of Rectangle in which all rectangles must be wider than they are high

State Support in a Class Interface

- Inappropriate States
  - A class interface design with inappropriate states typically offers the outside users of an object some states that are not formally part of the object's class abstraction

State Support in a Class Interface

- Inappropriate States

  example

  Consider a stack implemented by means of an array and an array pointer which is publicly visible

State Support in a Class Interface

- Inappropriate States
  - Identification of inappropriate states is a difficult problem
  - For example, depth of a stack is considered as inappropriate information while length of a queue is considered as appropriate information

State Support in a Class Interface

- Ideal States
  - In a class interface with ideal states, an object of a class may reach any state legal for that class

Behaviour Support in a Class Interface

- Illegal Behaviour
- Dangerous Behaviour
- Irrelevant Behaviour
- Incomplete Behaviour
- Awkward Behaviour
- Replicated Behaviour
- Ideal Behaviour

Behaviour Support in a Class Interface

- Illegal Behaviour
  - A class interface that supports illegal behaviour has an operation that allows an object to make illegal transitions from one state to another

Behaviour Support in a Class Interface

- Illegal Behaviour

  example
  A design of the class stack in which a user of a stack object could pull out an element from the middle of the stack

Behaviour Support in a Class Interface

- Dangerous Behaviour
  - When a class has an interface with dangerous behaviour, multiple messages are needed to carry out a single piece of an object's behaviour and at least one of the messages takes the object to an illegal state

Behaviour Support in a Class Interface

- Dangerous Behaviour

  example

  Suppose we want to move a rectangle to the right. In order to do that, for a bad Rectangle design, we have to send four messages to the rectangle object because each message moves one corner

Behaviour Support in a Class Interface

- Irrelevant Behaviour
  - A behaviour that simply doesn't belong to that class and objects

Behaviour Support in a Class Interface

- Irrelevant Behaviour
  - A behaviour that simply doesn't belong to that class and objects

example

add operation in a stack

Behaviour Support in a Class Interface

- Incomplete Behaviour
  - A class interface with incomplete behaviour does not allow all behaviour needed by objects of that class to be carried out

Behaviour Support in a Class Interface

- Incomplete Behaviour
  - A class interface with incomplete behaviour does not allow all behaviour needed by objects of that class to be carried out

example

A stack with push operation and without pop operation

Behaviour Support in a Class Interface

- Awkward Behaviour
    - An object whose class has an interface with awkward behaviour may require two or more messages to carry out a single piece of legal behaviour
    - However, none of the messages takes the object to an illegal state

Behaviour Support in a Class Interface

- Awkward Behaviour

  example

  The rotate operation which rotates a triangle by 90° in the clockwise direction can be performed by two simultaneous rotate operations which rotate a triangle by 45° in the clockwise direction

Behaviour Support in a Class Interface

- Replicated Behaviour
  - An interface to a class has replicated behaviour if the same piece of behaviour in an object may be carried out in more than one way

Behaviour Support in a Class Interface

- Replicated Behaviour

  example
  In a rectangle the following operations are defined.

  turnRight //turn to the right the rectangle by 90°
  turnClockwise (turnAngle) // turn the rectangle
  // in the clockwise direction by the specified angle

Behaviour Support in a Class Interface

- Ideal Behaviour
  - An interface to a class supports ideal behaviour if it enforces the following three properties
    - An object in a legal state can move only to another legal state
    - An object can move to another state only in a legal way
    - There is only one way to use the interface in order to carry out a piece of behaviour

Behaviour Support in a Class Interface

- Ideal Behaviour

  example

  Consider the interface of a class stack containing the following operations

  top : Integer // returns the top element of a stack

  pop( ) //removes the top element of the stack

  push(element) // places a new element on top of // the stack

  isEmpty : Boolean // returns whether the stack is // empty

  isFull : Boolean // returns whether the stack is full

Operation Cohesion in a Class Interface

- Alternate Cohesion
- Multiple Cohesion
- Functional Cohesion (Ideal Cohesion)

Operation Cohesion in a Class Interface

- Alternate Cohesion
  - It arises when a designer combines several pieces of behaviour into a single operation
  - The operation on the receipt of a message applies only one piece of behaviour to the object
  - A flag is supplied as part of the message, that tells which piece of behaviour to execute this time

Operation Cohesion in a Class Interface

- Alternate Cohesion

  Example (Bad Design)

  Rectangle. scaleOrRotate (scaleFactor:Real, rotateAngle:Angle, whichToDo:Boolean)

Operation Cohesion in a Class Interface

- Alternate Cohesion

  Example (Worse Design)

  Rectangle. scaleOrRotate (amount:Real, whichToDo:Boolean)

Operation Cohesion in a Class Interface

- Multiple Cohesion
  - It also combines several pieces of behaviour into a single operation
  - Here it applies all pieces of behaviour to the object

Example

Person. changeAddressAndPhoneNo(...)

Operation Cohesion in a Class Interface

- Functional Cohesion (Ideal Cohesion)
  An operation has functional cohesion if it is dedicated to a single piece of behaviour

  - An “and” name implies multiple cohesion
  - An “or” name implies alternate cohesion
  - A strong name with neither “and” nor “or” in it implies an operation with functional cohesion

Operation Cohesion in a Class Interface

- Functional Cohesion (Ideal Cohesion)
  An operation has functional cohesion if it is dedicated to a single piece of behaviour

  examples
  Tank.fill, Rectangle.calculateArea
  ProductItem.getWeight, AirPlane.turn
  Account.makeDeposit, Customer.getPhoneNo

- A feeble name such as Customer.doSomeStuff is not much good

Components And Objects

Design Of A Component

Lightweight Components

Heavyweight Components

Advantages Of Using Components

Disadvantages Of Using Components