Components Based Design and Development

Computer Engineering Studies

Universidad Carlos III de Madrid

Unit 6: Components Models

Juan Llorens

Högskolan på Åland – Finland / Universidad Carlos III de Madrid - Spain

Juan.llorens@uc3m.es

Components Models

- SW Models to define components
  - Structured models
    - Modules
    - Functions
  - Object models
    - Objects
    - Interactions between objects
  - Distributed Models

- SW Models to interact with components
  - Structured Models
  - Object Models
  - Interface Models - Interfacing and Typing
  - Distributed Models

Ian Sommerville 2004 - Software Engineering, 7th edition. Chapter 8

Components Based Design and Development

Computer Engineering Studies

Universidad Carlos III de Madrid

Unit 5.1: Objects Models

Object models

System - Up Definition

- Natural ways of representing a System, Subsystem, Component, Functionality,.. Using just objects and their relationships

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Real World – Down definition

- Natural ways of reflecting the real-world entities manipulated by the system
- More abstract entities are more difficult to model using this approach
- Object class identification is recognised as a difficult process requiring a deep understanding of the application domain
- Object classes reflecting domain entities are reusable across systems

Ian Sommerville 2004 - Software Engineering, 7th edition. Chapter 8

Object models: Creation

- The hard part about object-oriented design is decomposing a system into objects.
- The task is difficult because many factors come into play:
  - Encapsulation
  - Granularity
  - Dependency
  - Flexibility
  - Performance
  - Evolution
  - Reusability
  - And on and on.
- They all influence the decomposition, often in conflicting ways.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Object models: Creation

Different approaches within Object-oriented design methodologies

- You can write a problem statement, single out the nouns and verbs, and create corresponding classes and operations.
- Or you can focus on the collaborations and responsibilities in your system.
- Or you can model the real world and translate the objects found during analysis into design.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

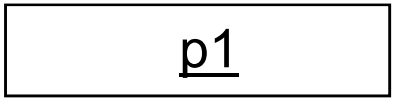
Object models and the UML

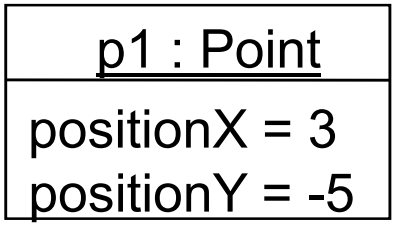
- The UML is a standard representation devised by the developers of widely used object-oriented analysis and design methods.
- It has become an effective standard for object-oriented modelling.
- Notation
  - Object classes are rectangles with the name at the top, attributes in the middle section and operations in the bottom section;
  - Relationships between object classes (known as associations) are shown as lines linking objects;
  - Inheritance is referred to as generalisation and is shown ‘upwards’ rather than ‘downwards’ in a hierarchy.

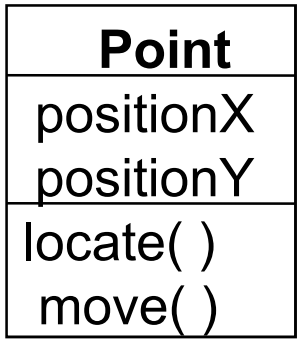
Ian Sommerville 2004 - Software Engineering, 7th edition. Chapter 8

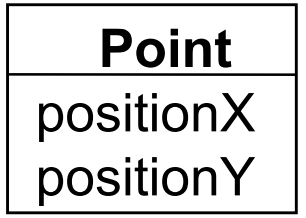
Objects and classes

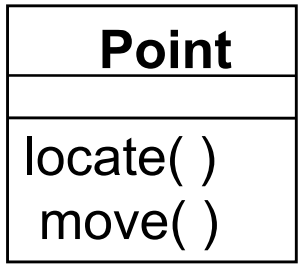
- Two levels of abstraction:
  - **object**: representation of a concrete entity with identity, state and behaviour
  - **class**: specification of a set of entities with structure and common behaviour
- **Types of classes**:
  - physical objects: airplane, person, book exemplar...
  - logical objects : bank account, subject, complex number, book (type)
  - historical objects: account operation, booking of a room
  - abstract objects : product for sale, ingredients of a recipe
- Optional suppression of compartments











Attributes and operations

- **Attribute**: a property “shared” by the objects of a class
  - each attribute has a **value** (probably different) for each object
  - redundant properties that can be calculated on the basis of other properties
    - **/area** (= base * height)
    - can be implemented as operations
- **Operation**: a function or transformation that can be applied to an object (the objects of a class have all the same operations)
  - can be **invoked** by other objects, or by the same object
  - **method**: procedural specification (implementation) of an operation
- **Notation** (can suppress all the elements except the name):
  - visibility **attributeName** multiplicity : Type = initialValue
    - balance : Money = 0
    - officePhone [0..2]
  - visibility **operationName** (argument : Type = defaultValue,…) : ReturnedType
    - getBalance ( ) : Money
    - call (number : Phone; attemptsNb : Integer)

Visibility

- **Levels of visibility** (different in each programming language):
  - \+ public: visible to all the classes (option by default for **operations**)
  - ~ package: visible to all the classes that are in the same package
  - # protected: visible to the subclasses
  - – private: visible only to the class (option by default for **attributes**)

<table>
<tr><th colspan="2">UML</th><th colspan="2">Java</th><th colspan="2">.NET</th></tr>
<tr><td colspan="2">Public</td><td colspan="2">Public</td><td colspan="2">Public</td></tr>
<tr><td rowspan="2">Package</td><td rowspan="2">Protected</td><td colspan="2">Protected</td><td colspan="2">Protected-Internal</td></tr>
<tr><td>friendly</td><td></td><td>Internal</td><td>Protected</td></tr>
<tr><td colspan="2">Private</td><td colspan="2">Private</td><td colspan="2">Private</td></tr>
</table>

Connections and associations

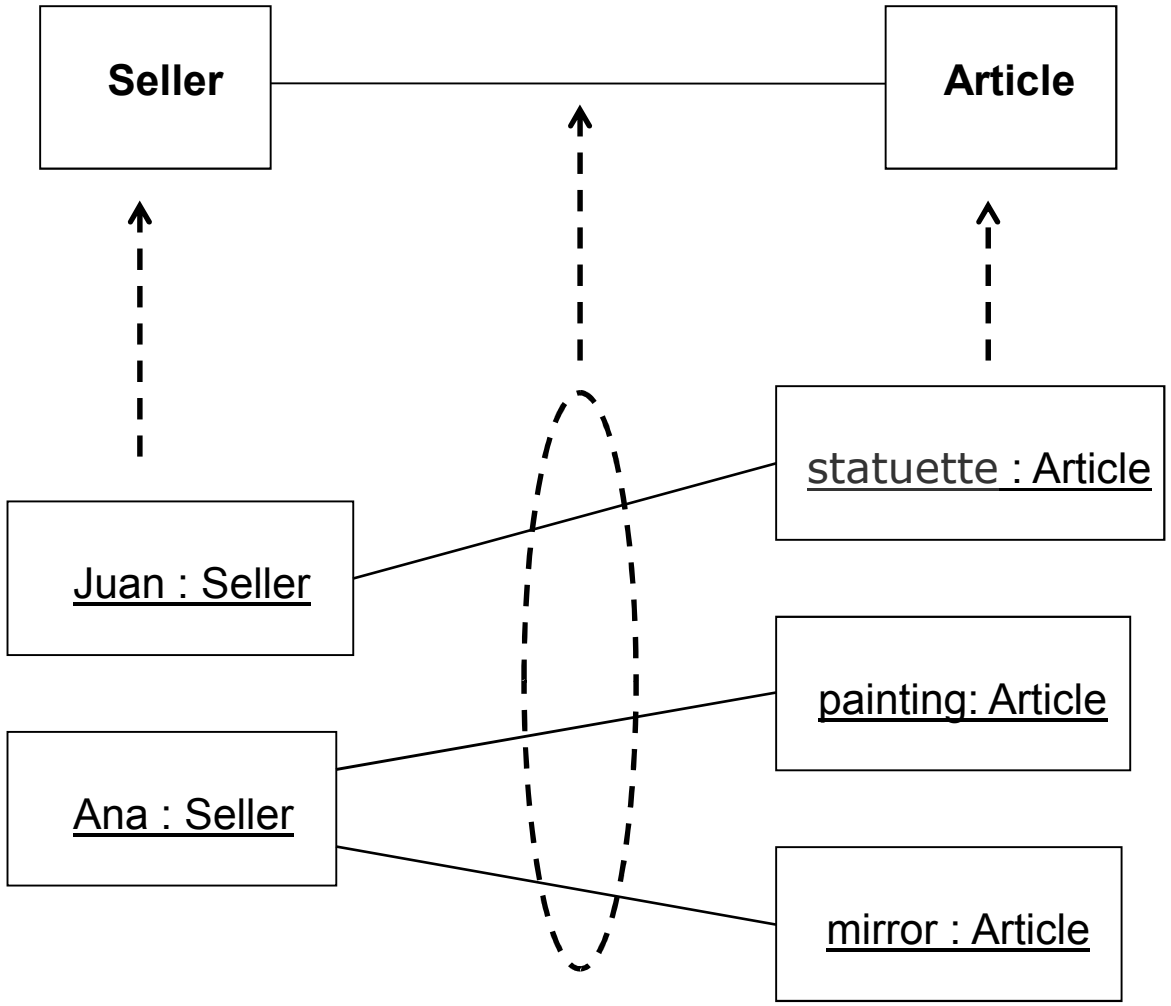
Association:
specification of a set of connections

represents the **structure** and the ***behaviour*** of the system

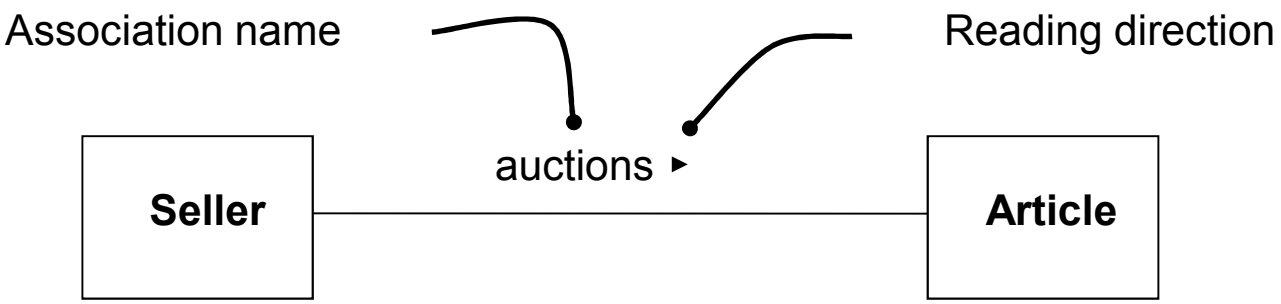
Connection:
A connection between objects determines a tuple of objects, instance of an association

state of the connected objects
state of the system

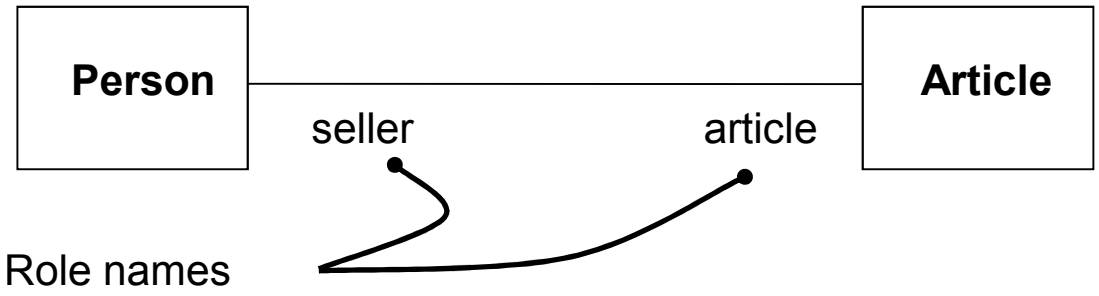
Fact + possibility of **communication**



Association names and role names



Association names can be repeated in a model,
except for associations between the same classes



Role names can be repeated in different associations, and can be equal to the names of the associated classes

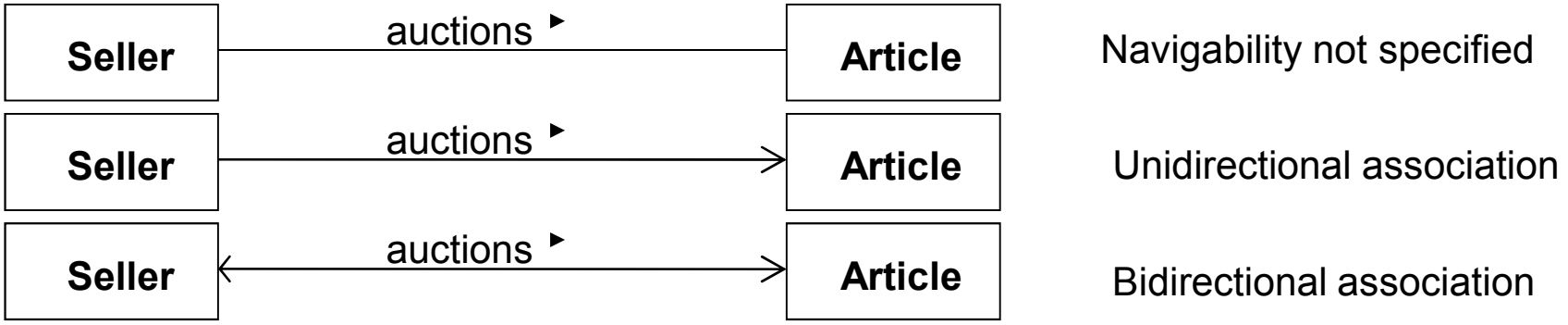
Multiplicity of an association

- In a **binary association**, the multiplicity of an **extremity of the association** specifies the number of **destination** instances that can be connected with a unique **origin** through the association

- **Typical values**:
  - 0..1 zero or one
  - 1..1 one and only one (abbreviated as “1”)
  - 0..* from zero to “many” (abbreviated as “*”)
  - 1..* from one to “many”
- Other multiplicity values:
  - ranges between: (2..*), (0..3), etc.
  - list of ranges separated by comas: (1, 3, 5..10, 20..*), (0, 2, 4, 8), etc.

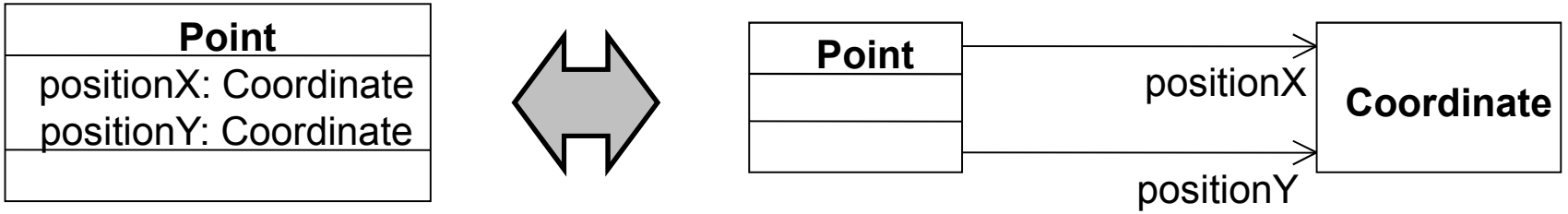
Navigability of an association

- The navigability of a **binary association** specifies the capacity of an instance of the **origin** class to access instances of the **destination** class through the **instances of the association** that connects those classes.
- **To access** = to name, designate or reference the object to...
  - read or modify the value of an **attribute** of the object (not advised)
  - ➔ invoke an **operation** of the object **(to send it a message)**
  - use the object as **argument** or **return** value in a message
  - modify (to assign or to delete) the **connection** with the object
- Do not confuse:
  - **reading direction of the name** of the association: linguistic asymmetry
  - **navigability** or directionality of the association: asymmetry of communication

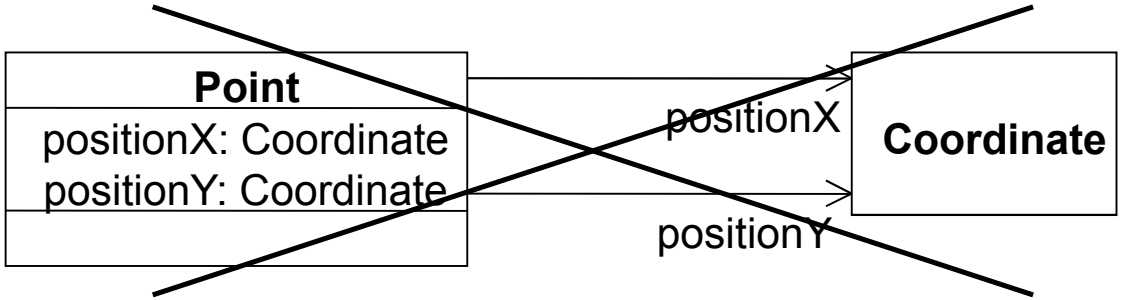


Association vs. Attribute

- An **attribute** is equivalent to a **unidirectional association**

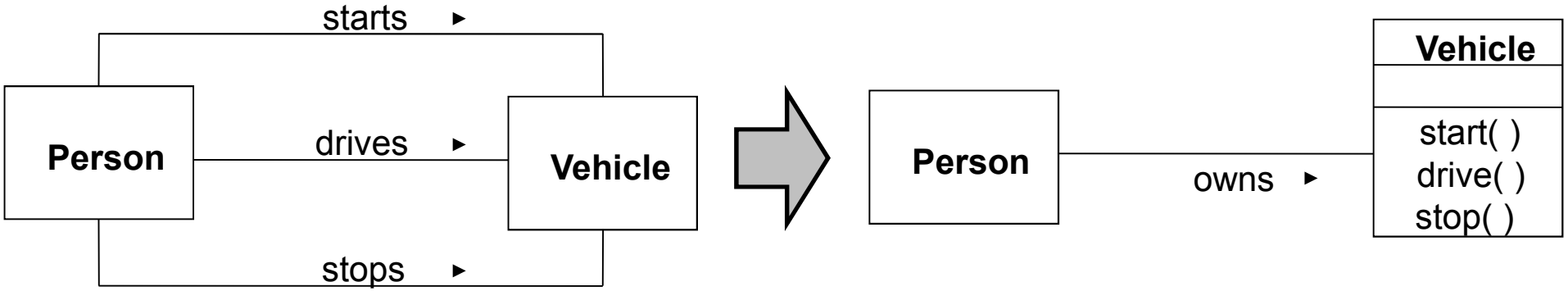


- It is **incorrect** to duplicate the representation



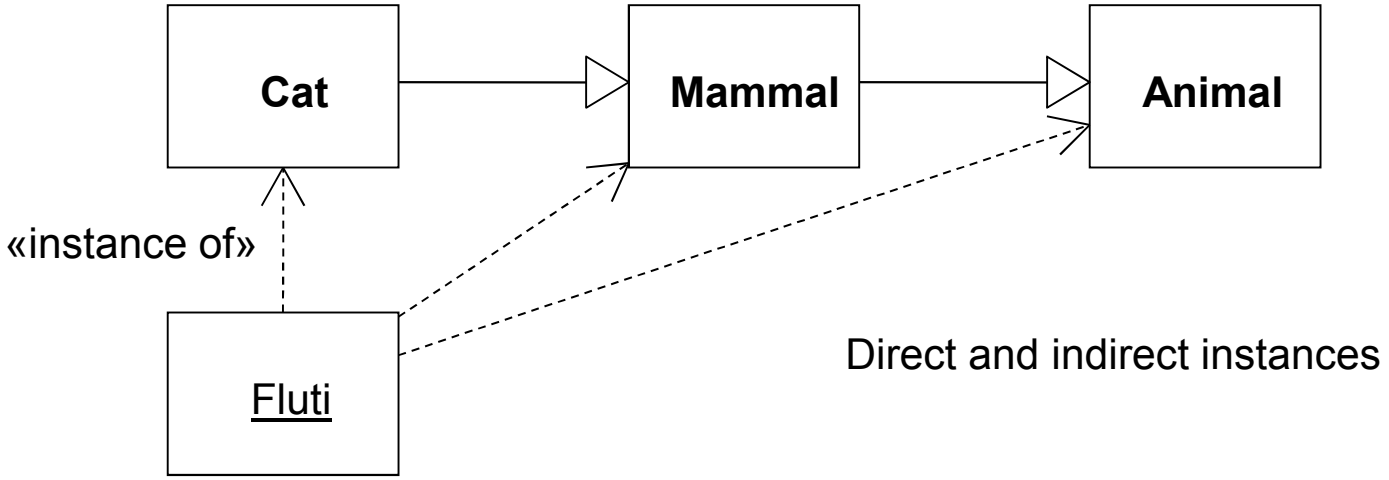
Association vs. Operation

- Every association has a **double meaning**:
  - **aspect** : structure of the system (possible states)
  - **aspect** : behaviour of the system (possible interactions)
- The name of an association may reflect one aspect more than the other:
  - **static names**: contains, located-in, works-for, married, etc.
  - **dynamic names** (of action): sells, publishes, consults, etc.
- Static names are preferable, leaving dynamic names to be used as **names of operations**, invoked through the association by sending messages
- A same association enables the invocation of many operations



Generalisation and classification

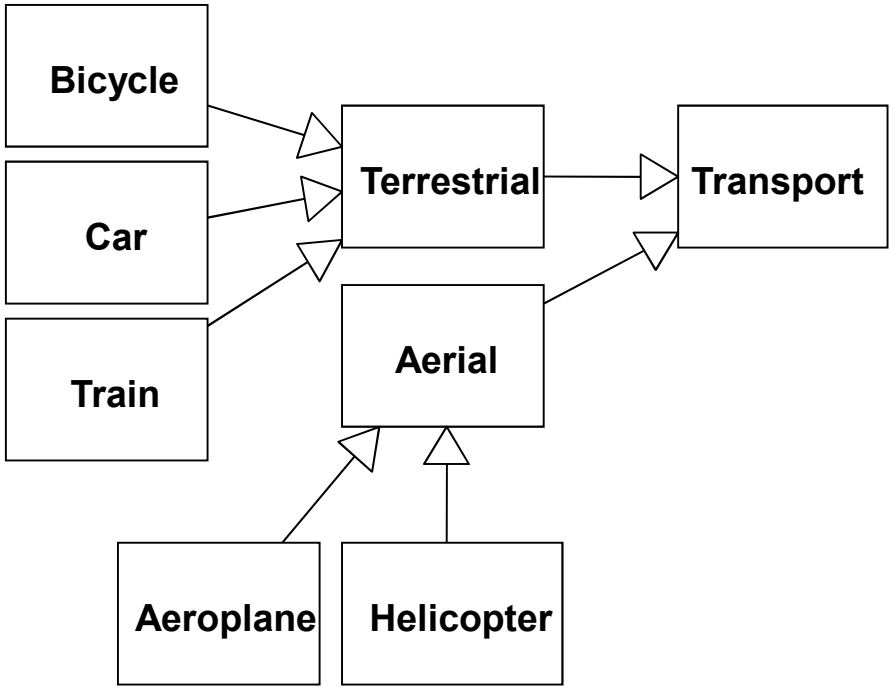
- Principle of substitution:
  - **extention**: all the objects of the subclass also are of the superclass
  - **intention**: the definition of the superclass is applicable to the subclass
- Generalisation: class-class
  - Cat **is a (sub-)class of** Mammal, Mammal **is a (sub-)class of** Animal
- Classification: object-class
  - Fluti **is a** Cat, Fluti **is a** Mammal, Fluti **is an** Animal



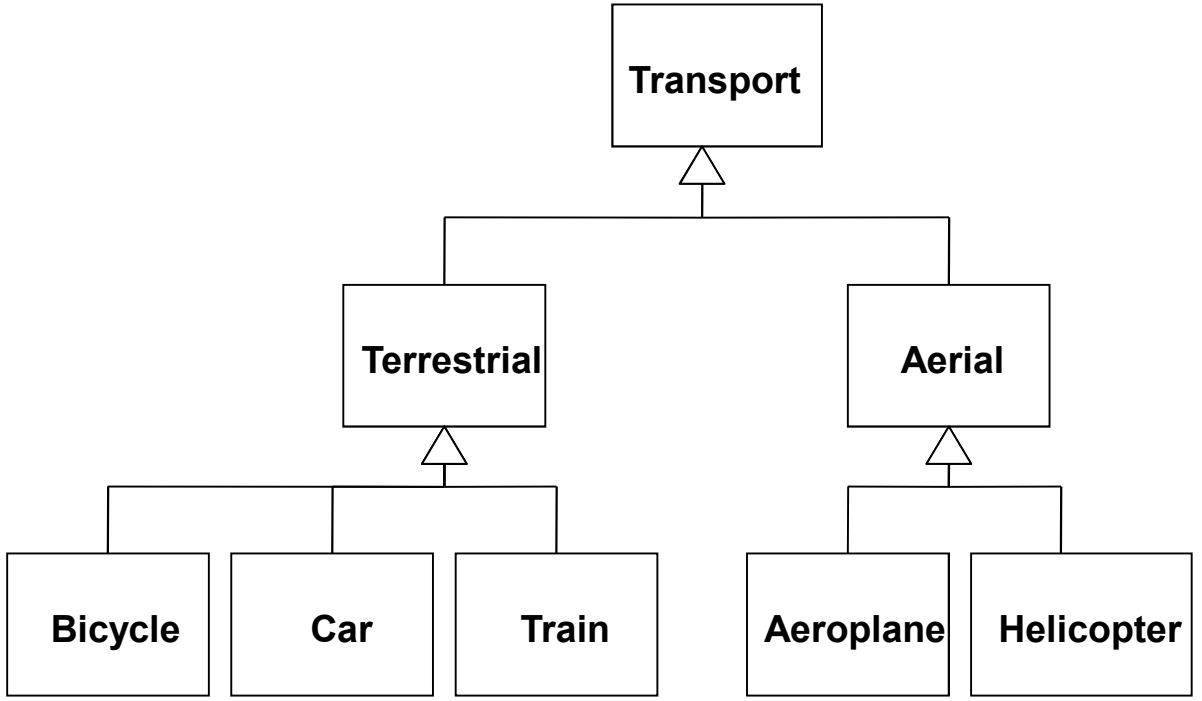
Generalisation and specialisation

- Two complementary points of view :
  - **to generalise** is to identify the **common properties** (attributes, associations, operations) of several classes and to represent them by a more general class, known as their **superclass**
    - augment the level of abstraction, reduce complexity, organize
  - **to specialise** is to capture **new specific properties** of a set of objects of a given class, which have not been identified in that class, and to represent them by a new class, know as its **subclass**
    - reuse of a concept while adding new properties (or variations) to it
- This relation is purely between classes:
  - it does not have instances, nor multiplicity
  - the subclass inherits **all** the properties of the superclass
  - the inherited properties of the superclass are not represented in the subclass (unless they are redefined operations)
  - every generalisation induces a **dependency** subclass → superclass

Class hierarchies



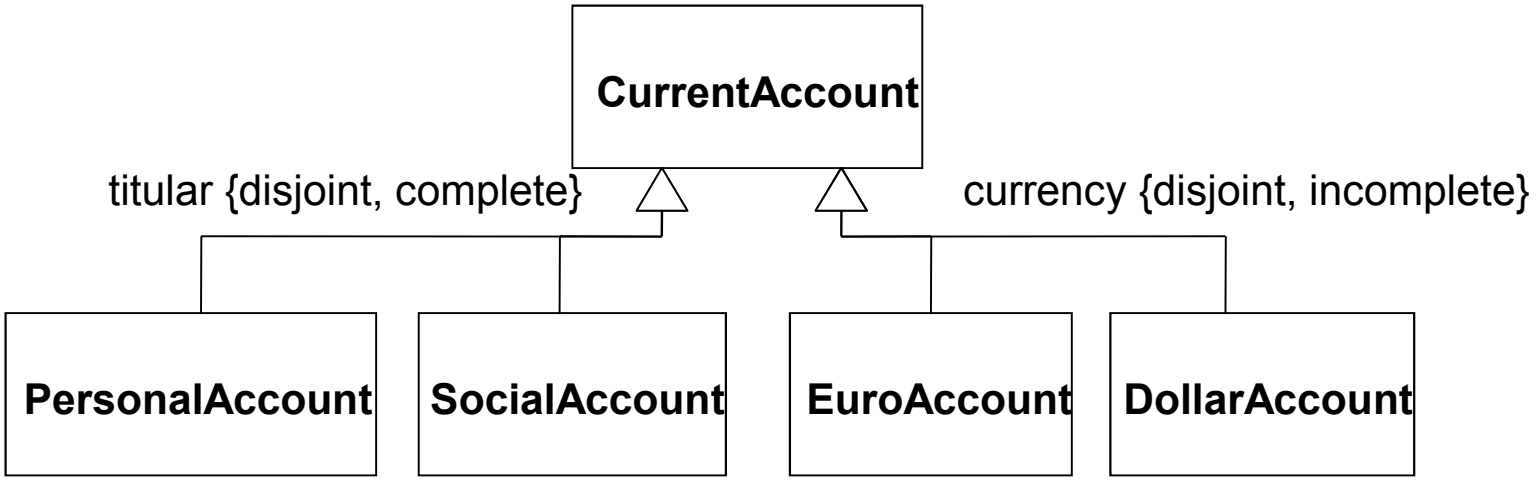
Alternative representations :
- binary relations
- tree structure



Generalisation:
- not reflexive
- transitive
- asymmetric

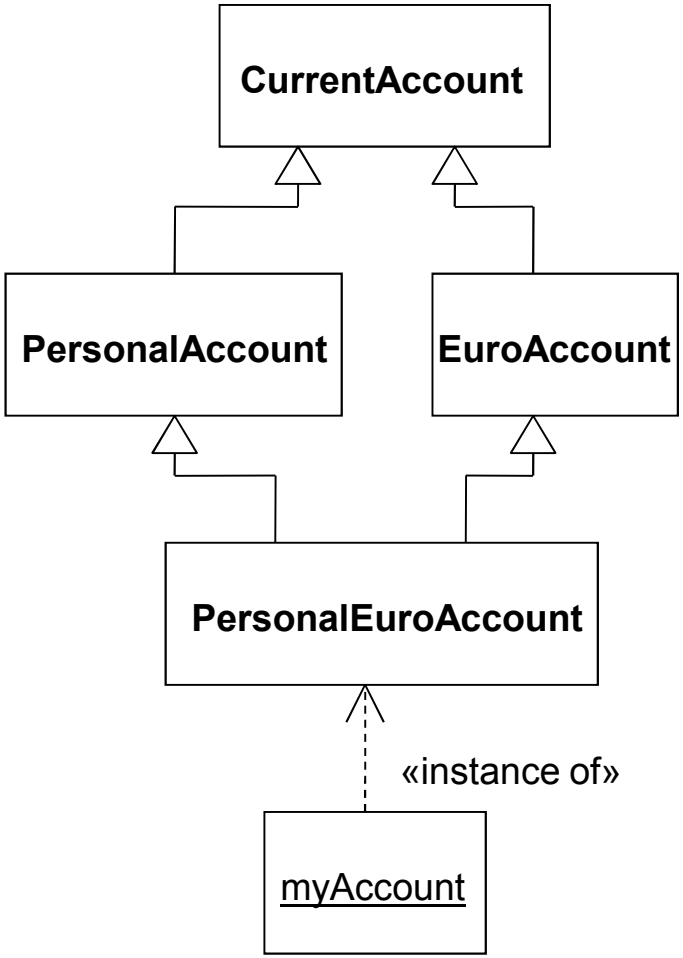
Dimensions of specialisation

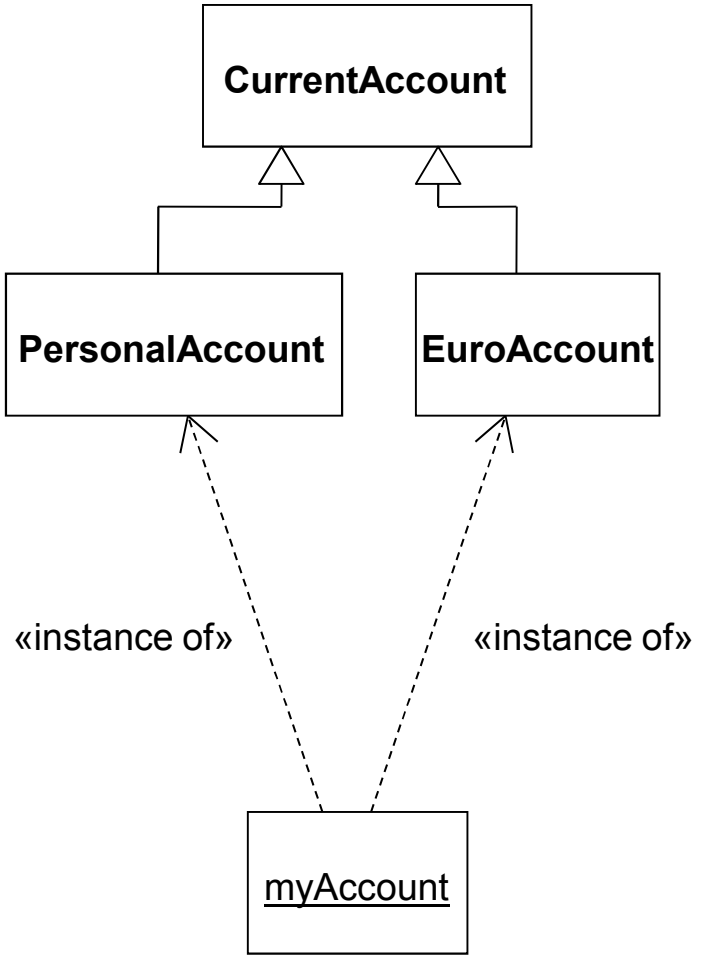
- A superclass can be specialised in different **groups of subclasses** according to independent criteria : **discriminators**



- Restrictions:
  - **overlapping/disjoint**: the subclasses can / cannot have instances in common
  - **incomplete/complete**: there are / are no other subclasses
- Values by default: disjoint, incomplete
- **Partition** (strictly speaking): disjoint, complete

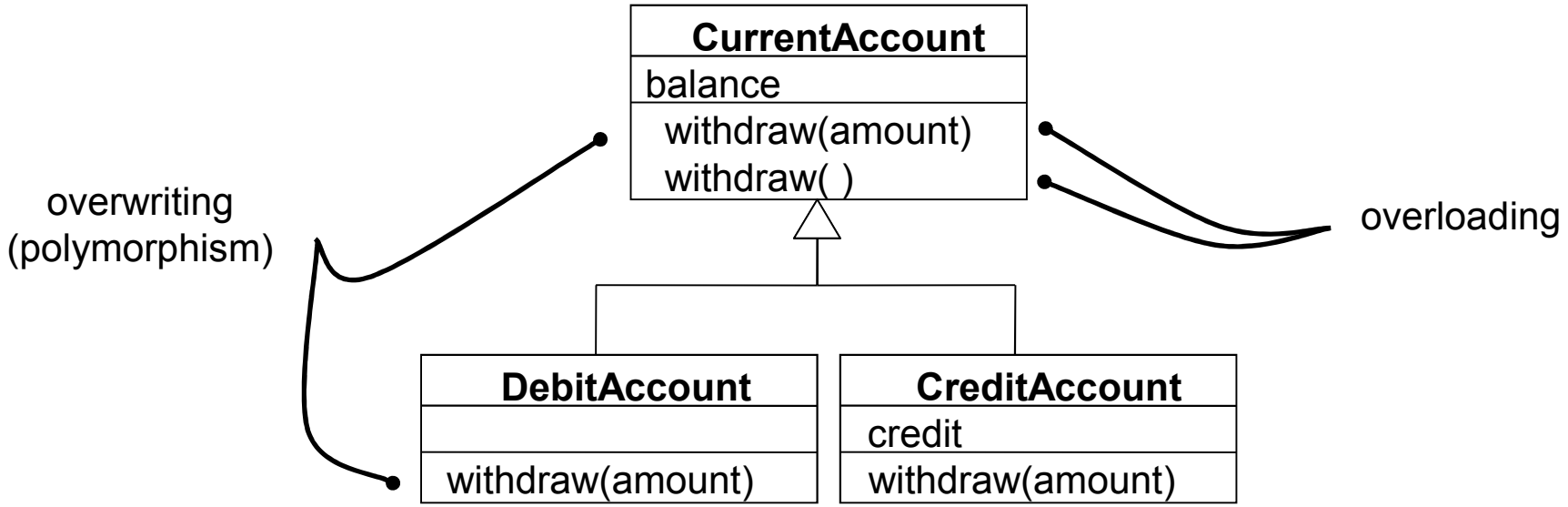
Multiple Generalisation vs. Multiple Classification





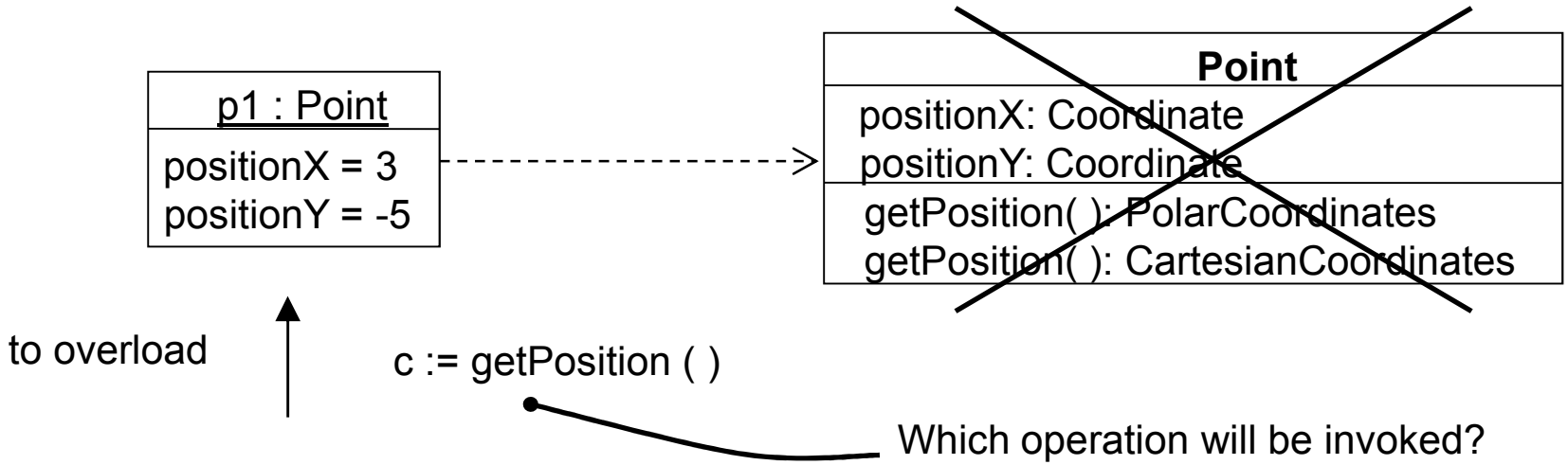
Polymorphism of operations

- Capacity of executing different methods in response to a same message
- A **polymorphic operation** is one that has multiple implementations/methods
- Do not confuse **overwriting a method** with operation **overloading**
  - to overwrite: to redefine in another class the **method** of a same operation
    - the method is selected at **execution time**
  - to overload: to reuse the **name** of an operation, but with different parameters
    - the operation is selected **at compilation time;** it is not polymorphic



Signature of operations

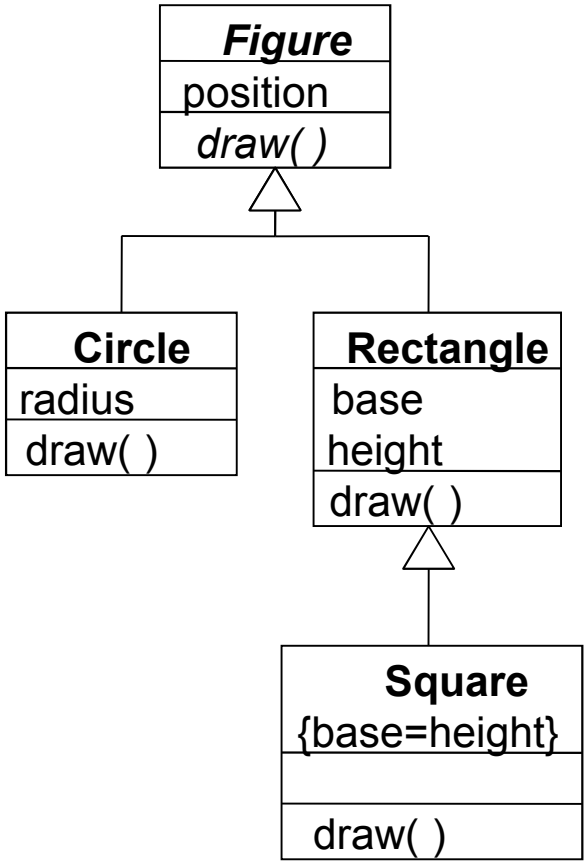
- A class cannot have two operations with the same **signature**, which consists of:
  - **name** of operation, number (order) and **type of the parameters**
- The **names of the parameters** are not part of the signature
- The **type of the return value** is not part of the signature, because it is not used to distinguish which operation shall be executed
  - ➔ It cannot be used for **overloading** operations (and neither for **overwriting)**



Abstract classes and operations

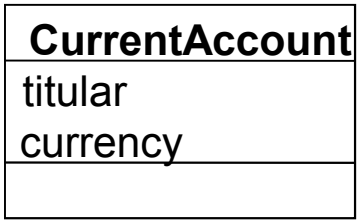
- **Abstract operation** : the signature is specified, but not the implementation
  - a class with one or several abstract operations is incomplete: it cannot have direct instances
  - abstract operations, just like the concrete ones, can be overwritten (polymorphic)
  - it is safer to overwrite an abstract operation than to overwrite a concrete operation (less risk of changing its meaning)

- **Abstract class**: is incomplete, cannot have direct instances
  - can have indirect instances through its concrete subclasses
  - a concrete class...
    - cannot have abstract operations
    - must provide implementations for all the abstract operations inherited

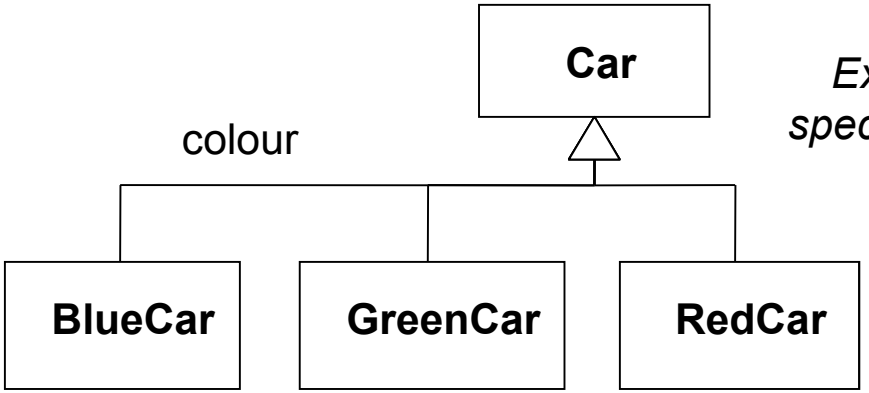


Subclass vs. Association (Attribute)

- How to model the properties of objects?
  General rule:
  - changeable property or large range of values: **attribute**
  - fixed property with enumerated values: **specialisation** (each property translates into a **criteria** of specialisation, each value in a **subclass**)
    - can also be modelled as an attribute with a fixed value



Alternative to the double specialisation



Excessive specialisation?

Class Inheritance versus object composition

- The two most common techniques for reusing functionality in object-oriented
- Class inheritance lets you define the implementation of one class in terms of another's.
  - Reuse by subclassing is often referred to as white-box reuse. The term "whitebox“ refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses.
- Object composition is an alternative to class inheritance.
  - New functionality is obtained by assembling or *composing* objects to get more complex functionality. This style of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as "black boxes.“
- Class Inheritance is defined at compile time while Object composition at run-time

Favor object composition over class inheritance.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Class Inheritance : advantages and disadvantages

- Class inheritance is defined statically at compile-time and is straightforward to use, since it's supported directly by the programming language.
- Class inheritance also makes it easier to modify the implementation being reused.
- When a subclass overrides some but not all operations, it can affect the operations it inherits as well, assuming they call the overridden operations.

Disadvantages

- You can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time.
- Parent classes often define at least part of their subclasses' physical representation.
  - The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change.
- Should any aspect of the inherited implementation not be appropriate for new problem domains, the parent class must be rewritten or replaced by something more appropriate

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Object Composition : advantages and disadvantages

- Composition requires objects to respect each others' interfaces, which in turn requires carefully designed interfaces that don't stop you from using one object with many others.

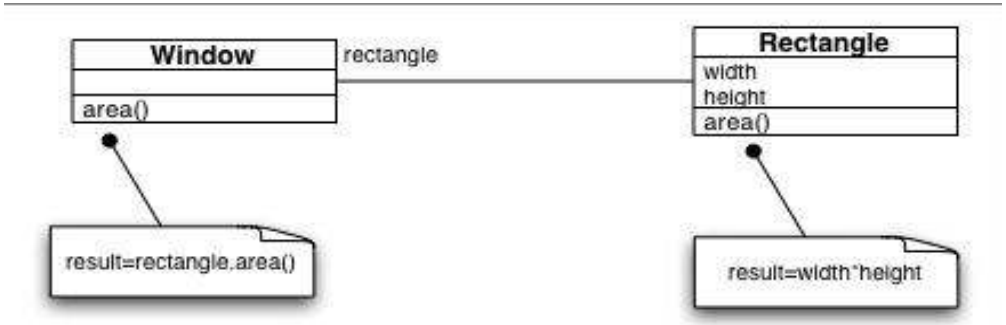
Advantages

- Because objects are accessed solely through their interfaces, we don't break encapsulation. Any object can be replaced at runtime by another as long as it has the same type. Moreover, because an object's implementation will be written in terms of object interfaces, there are substantially fewer implementation dependencies.
- Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task. Your classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters. On the other hand, a design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their interrelationships instead of being defined in one class.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance. In delegation, *two* objects are involved in handling a request: a receiving object delegates operations to its **delegate**.
- This is analogous to subclasses deferring requests to parent classes. But with inheritance, an inherited operation can always refer to the receiving object through the this member variable in C++ and self in Smalltalk. To achieve the same effect with delegation, the receiver passes itself to the delegate to let the delegated operation refer to the receiver.



E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Delegation : advantages and disadvantages

- The main advantage of delegation is that it makes it easy to compose behaviors at run-time and to change the way they're composed.
- Delegation has a disadvantage it shares with other techniques that make software more flexible through object composition: Dynamic, highly parameterized software is harder to understand than more static software.
- There are also run-time inefficiencies.
- Delegation is a good design choice only when it simplifies more than it complicates.
- It isn't easy to give rules that tell you exactly when to use delegation, because how effective it will be depends on the context and on how much experience you have with it.
- Delegation works best when it's used in highly stylized ways—that is, in standard patterns.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Components Based Design and Development

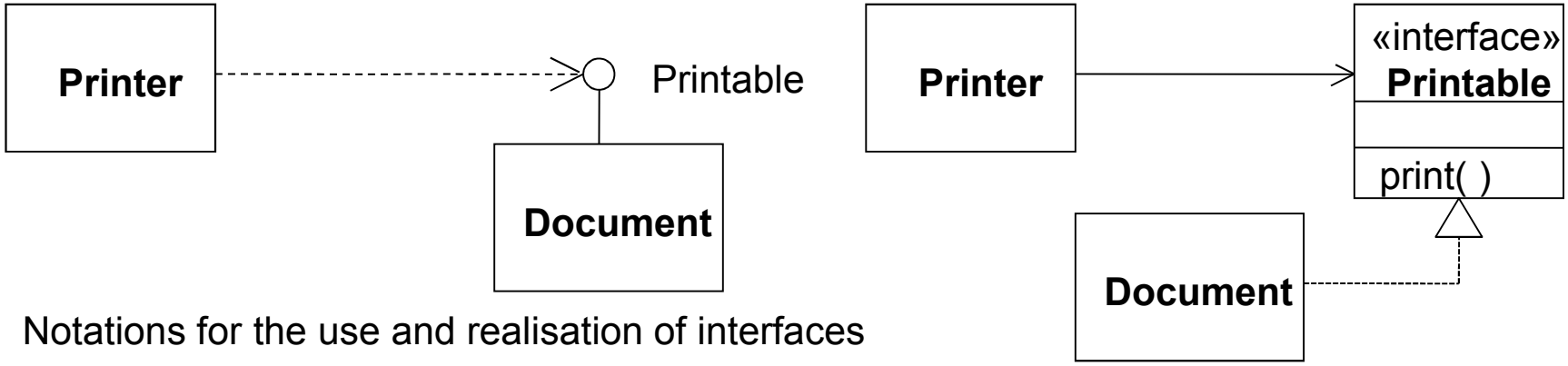
Computer Engineering Studies

Universidad Carlos III de Madrid

Unit 5.2: Interface Models

Interfaces

- Encapsulation: separation of interface and implementation in a class
  - a class can realize one or more interfaces
  - an interface can be realized by one or more classes
- Interface: **set of operations** that offer a coherent service
  - does not contain the implementation of the operations (**no methods**)
  - an interface cannot have **attributes** nor navigable **associations** (this restriction has been suppressed in UML 2.0)
  - analogue to an abstract class with only abstract operations, and without attributes nor associations



Notations for the use and realisation of interfaces

Class Versus Type

- A Type is a name used to denote a particular interface
  - We speak of an object as having the type "Window" if it accepts all requests for the operations defined in the interface named "Window."
- An object may have many types, and widely different objects can share a type.
- Part of an object's native interface may be characterized by one type, and other parts by other types.
- Two objects of the same type need only share parts of their native interfaces.
- Interfaces can contain other interfaces as subsets. We say that a type is a subtype of another if its interface contains the interface of its supertype.
- Often we speak of a subtype *inheriting* the interface of its supertype.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Interface vs Implementation

- Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces.
- There is no way to know anything about an object or to ask it to do anything without going through its interface.
- An object's interface says nothing about its implementation—different objects are free to implement requests differently.
- Two objects having completely different implementations can have identical interfaces.
- Dynamic binding means that issuing a request doesn't commit you to a particular implementation until runtime.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Interface vs Implementation II

- Consequently, you can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request.
- Dynamic binding lets you substitute objects that have identical interfaces for each other at run-time. This substitutability is known as polymorphism [..].
- It lets a client object make few assumptions about other objects beyond supporting a particular interface.
- Polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at run-time.
- An object's implementation is defined by its class.
  - The class specifies the object's internal data and representation and defines the operations the object can perform.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Interface vs Implementation III

- Consequently, you can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request.
- Subclasses can refine and redefine behaviors of their parent classes. More specifically, a class may override an operation defined by its parent class.
- Overriding gives subclasses a chance to handle requests instead of their parent classes.
  - Class inheritance lets you define classes simply by extending other classes, making it easy to define families of objects having related functionality.

"Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what not to put in the interface."

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Interface vs Implementation IV

"Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what not to put in the interface."

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Differences between Class and its Type

- An object's class defines how the object is implemented. The class defines the object's internal state and the implementation of its operations.
- An object's type only refers to its interface—the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

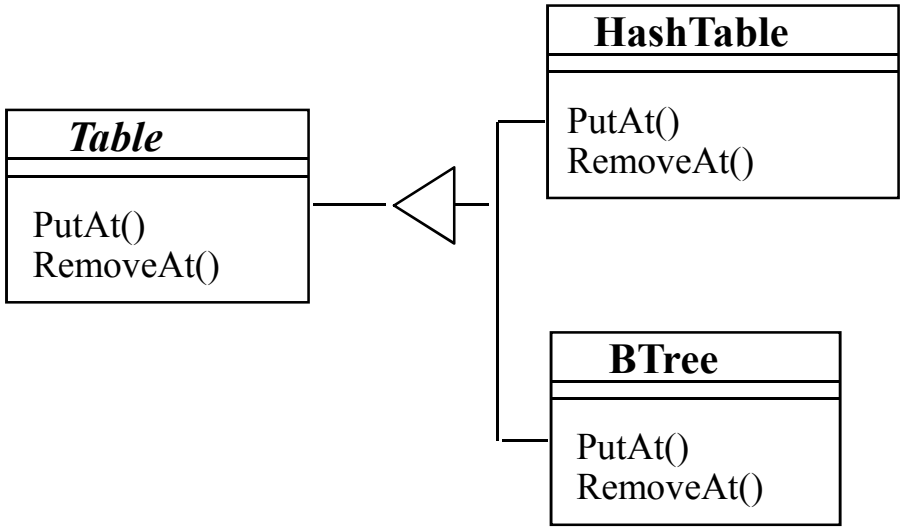
Class inheritance / Interface Inheritance (subtyping)

- Class inheritance defines an object's implementation in terms of another object's implementation.
  - In short, it's a mechanism for code and representation sharing.
- Interface inheritance (or subtyping) describes when an object can be used in place of another.
- The standard way to inherit an interface in C++ is to inherit publicly from a class that has (pure) virtual member functions.
  - Pure interface inheritance can be approximated in C++ by inheriting publicly from pure abstract classes. Pure implementation or class inheritance can be approximated with private inheritance.
- In Smalltalk, inheritance means just implementation inheritance.
  - You can assign instances of any class to a variable as long as those instances support the operation performed on the value of the variable.

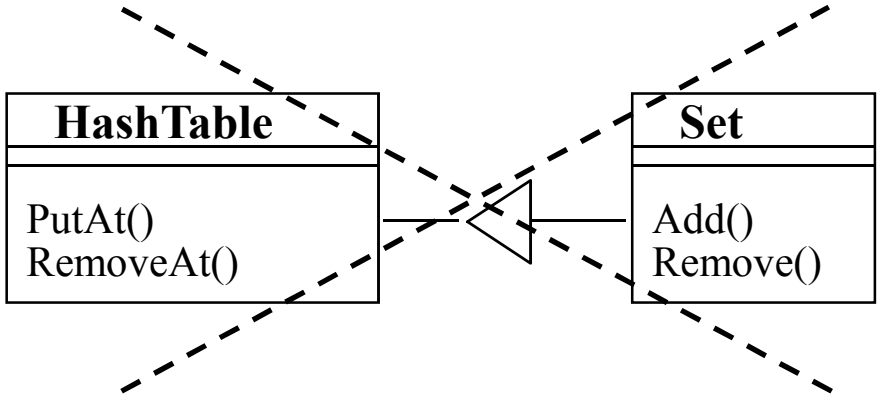
E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Class inheritance versus Interface Inheritance

- Interface inheritance (i.e., subtyping):
  - describes when different types of objects can be used in place of each other



- Implementation inheritance:
  - an object's implementation is defined in terms of the implementation of another.



Bernd Bruegge & Allen Dutoit - Object-Oriented Software Engineering: Conquering Complex and Changing Systems

Program to an Interface, not an Implementation

- Class inheritance is basically just a mechanism for extending an application's functionality by reusing functionality in parent classes.
- But inheritance's ability to define families of objects with *identical* interfaces (usually by inheriting from an abstract class) is also important.
  - Why? Because polymorphism depends on it.
- When inheritance is used carefully (some will say *properly*), all classes derived from an abstract class will share its interface.
  - This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class. *All* subclasses can then respond to the requests in the interface of this abstract class, making them all subtypes of the abstract class.

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Program to an Interface, not an Implementation: Benefits

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.

2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.

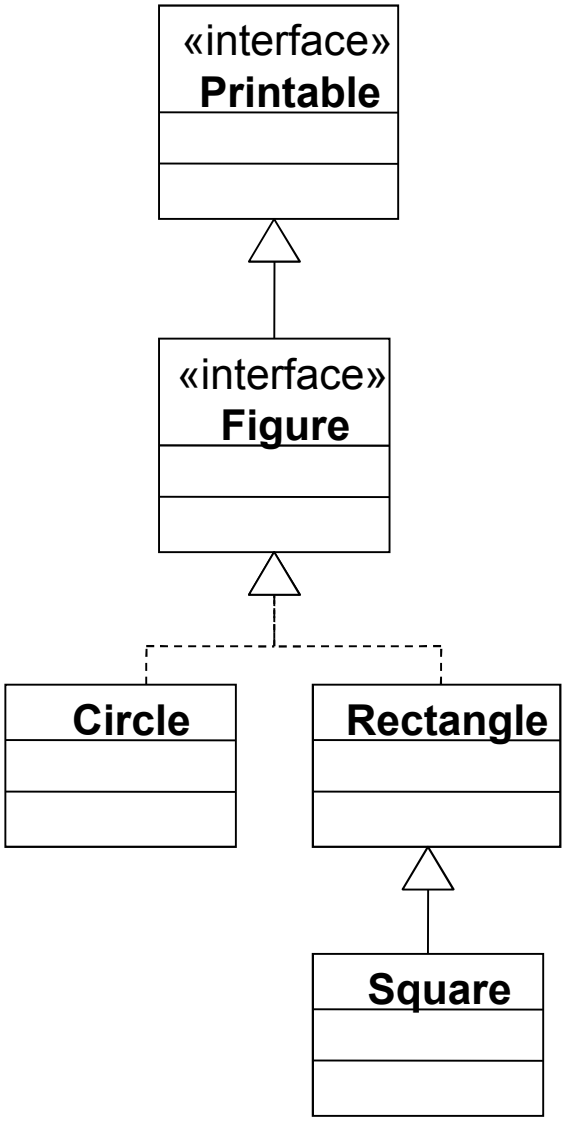
These are the fundamentals of the creational patterns

E.Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software

Generalisation vs. Realisation

- Realisation can be understood as a **"weak generalisation"**: it inherits the interface, but not the implementation:
  - reduces dependency
  - lowers reutilisation
  - alternative to multiple generalisation, but not supported by many programming languages
- Interfaces are generalisable elements
  - mixed hierarchies of interfaces and classes
- Design criterion: to commit only to the interface
  - declare the types of variables, or of parameters of operations, as interfaces, rather than as classes
  - any instance compatible with the interface can be used
- Example:

```
Figure f = new Square ( );
f.print
```



Interfaces

- An interface defines a contract
  - An interface is a type
  - Includes methods, properties, indexers, events
  - Any class or struct implementing an interface must support all parts of the contract
- Interfaces provide no implementation
  - When a class or struct implements an interface it must provide the implementation
- Interfaces provide polymorphism
  - Many classes and structs may implement a particular interface

Otro

Interfaces
Example

```
public interface IDelete {
    void Delete();
}
public class TextBox : IDelete {
    public void Delete() { ... }
}
public class Car : I
    public void Delete
}
```

```
TextBox tb = new TextBox();
IDelete iDel = tb;
iDel.Delete();

Car c = new Car();
iDel = c;
iDel.Delete();
```

Interfaces

Multiple Inheritance

- Classes and structs can inherit from multiple interfaces
- Interfaces can inherit from multiple interfaces

```
interface IControl {
    void Paint();
}
interface IListBox: IControl {
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {
}
```

Otro

Interfaces

Explicit Interface Members

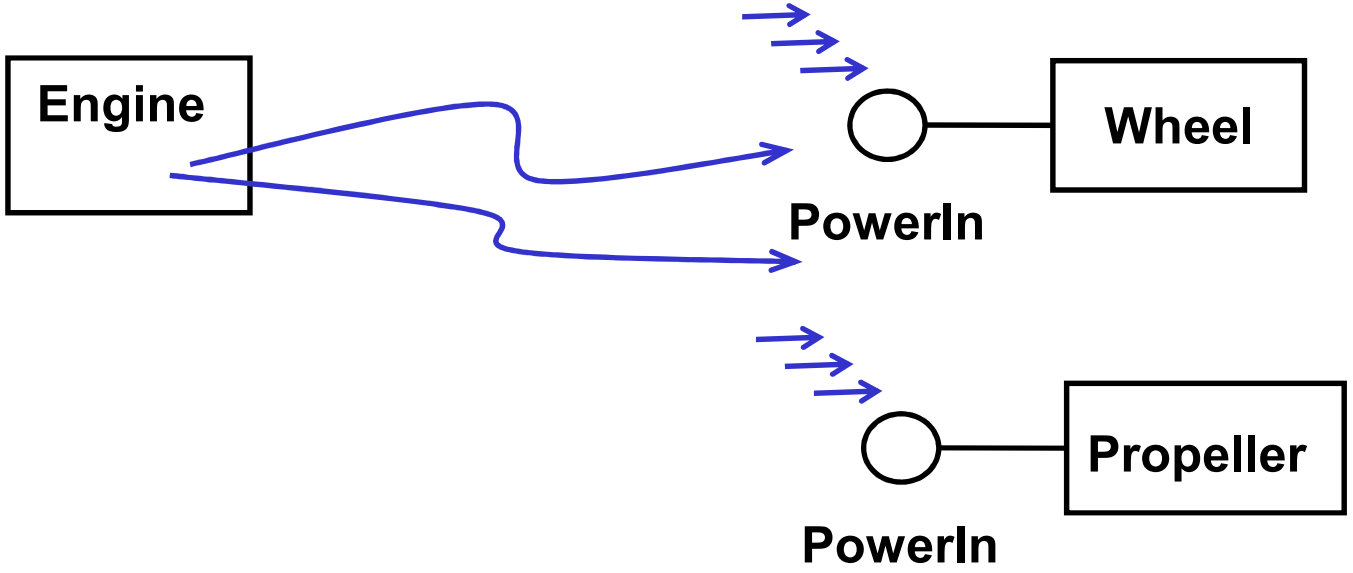
- If two interfaces have the same method name, you can explicitly specify interface + method name to disambiguate their implementations

```
interface IControl {
    void Delete();
}
interface IListBox: IControl {
    void Delete();
}
interface IComboBox: ITextBox, IListBox {
    void IControl.Delete();
    void IListBox.Delete();
}
```

Otro

Interfaces UML 1.x

- UML 1.x supports interfaces, but only in one direction:

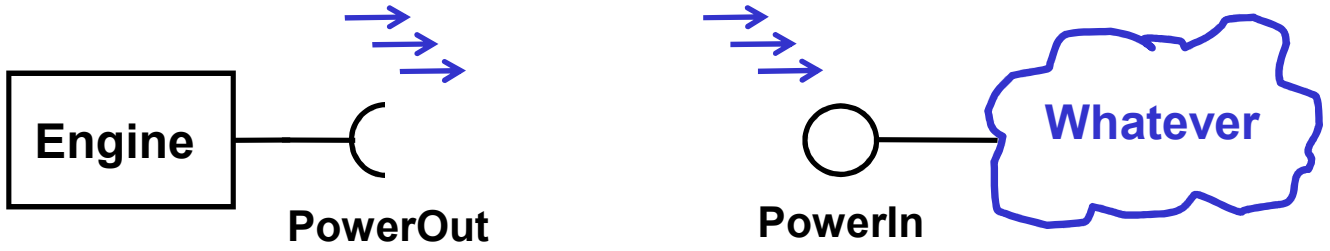


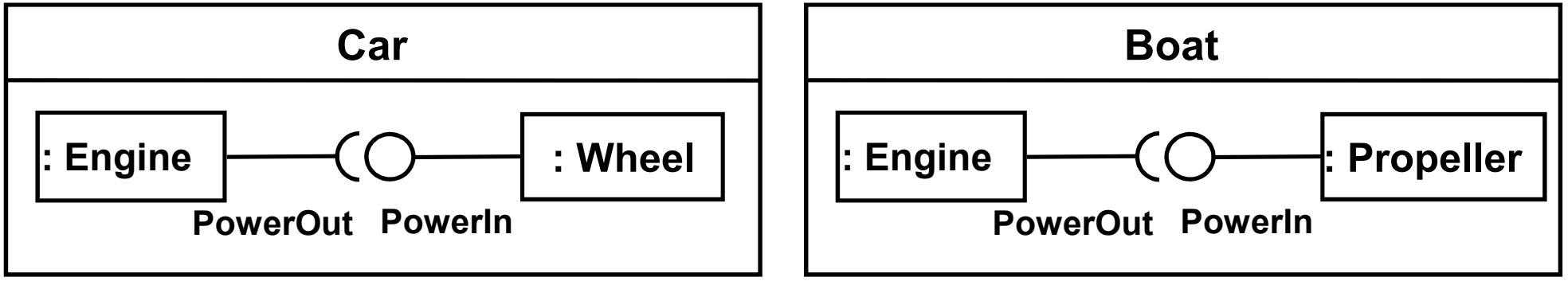
- **Interface usage buried in client methods.**

Ian Sommerville 2004 - Software Engineering, 7th edition. Chapter 8

Interfaces UML 2.0

- Bidirectional interfaces:

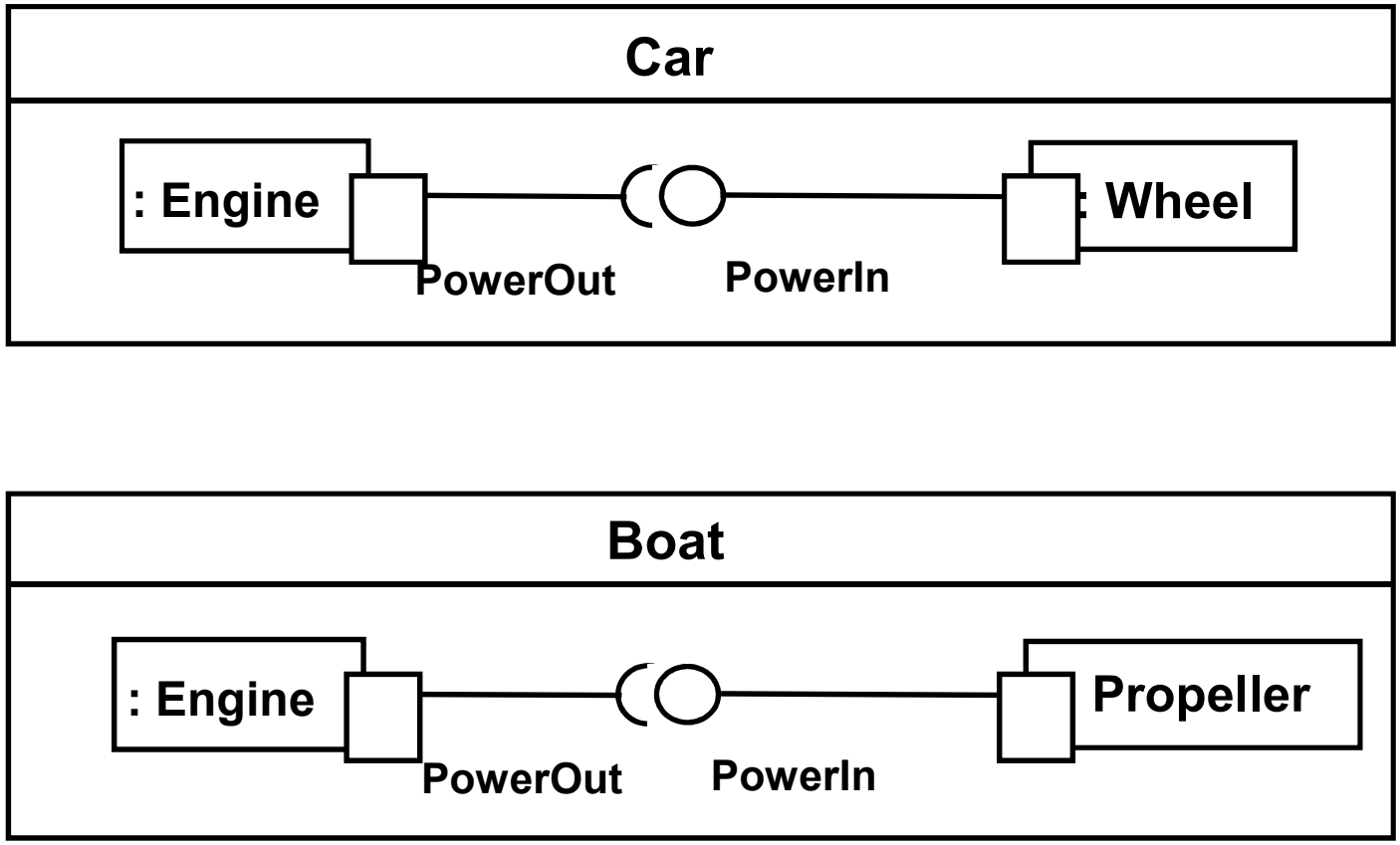




Ian Sommerville 2004 - Software Engineering, 7th edition. Chapter 8

Composition 2.0 (Ports)

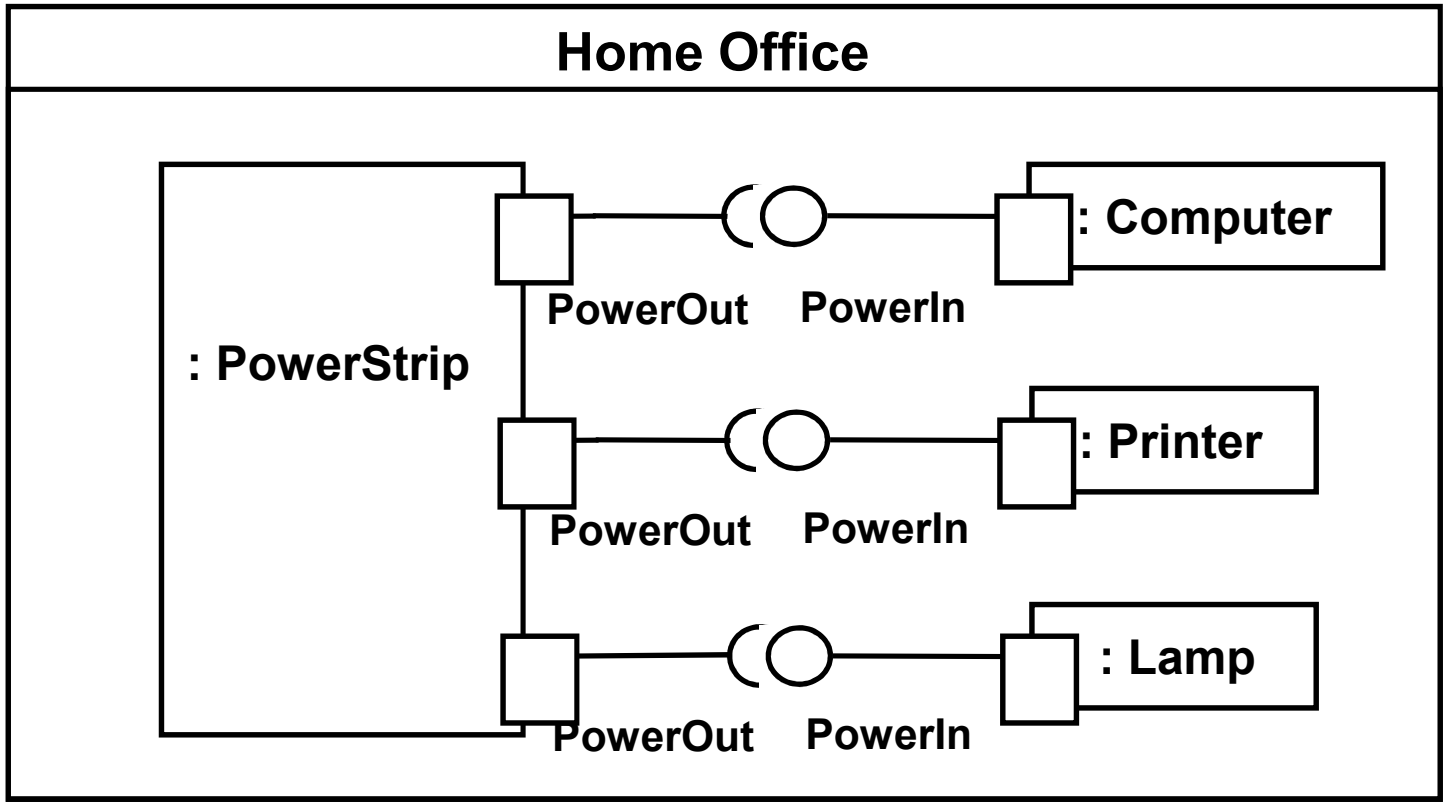
- Ports = public parts.



Ian Sommerville 2004 - Software Engineering, 7th edition. Chapter 8

Composition 2.0 (Ports)

- Multiple ports of the same type.



Ian Sommerville 2004 - Software Engineering, 7th edition. Chapter 8

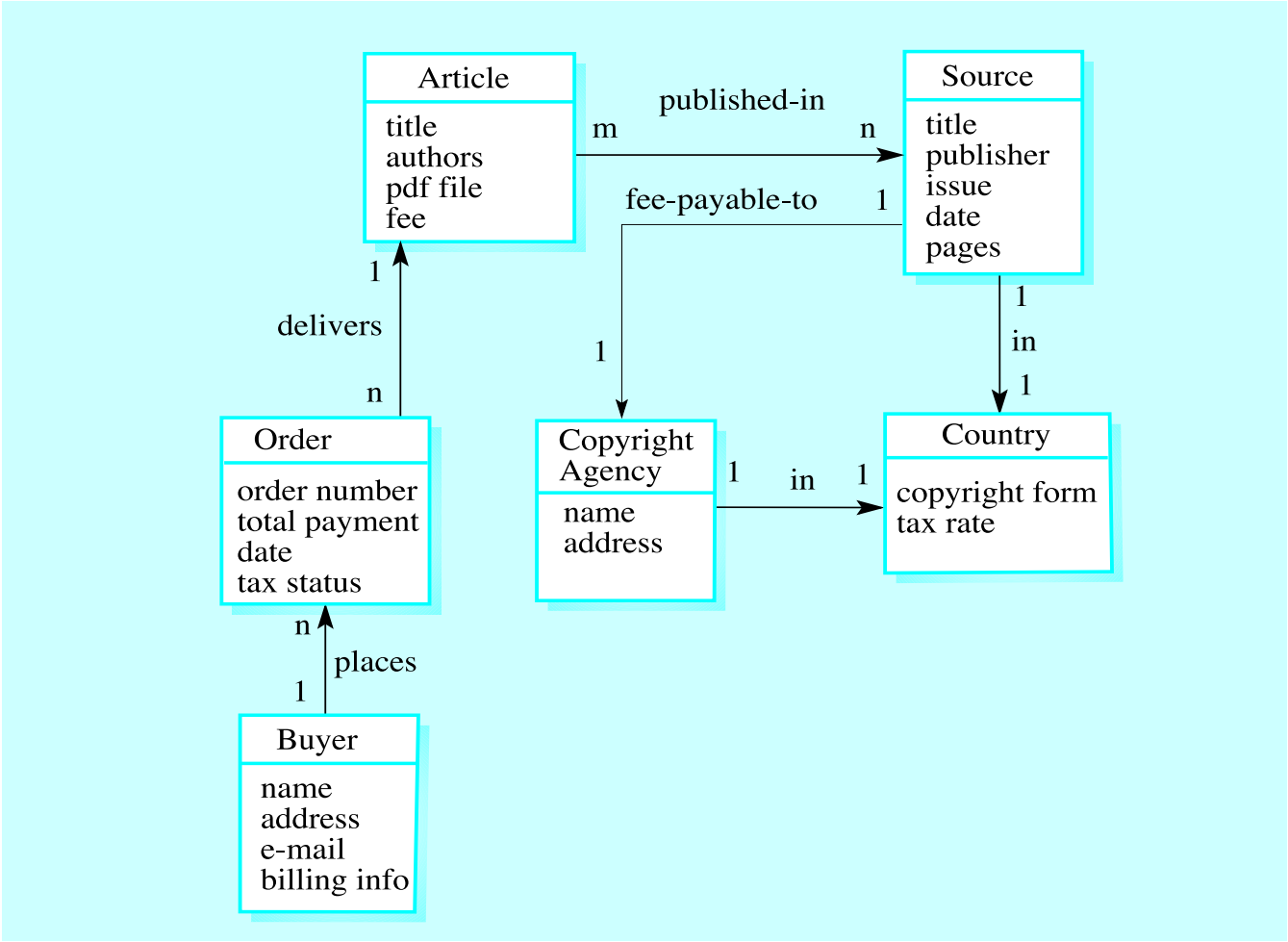
Semantic data models are NOT Object Models

DATA MODELS

- Used to describe the logical structure of data processed by the system.
- An entity-relation-attribute model sets out the entities in the system, the relationships between these entities and the entity attributes
- Widely used in database design. Can readily be implemented using relational databases.
- No specific notation provided in the UML but objects and associations can be used.

Ian Sommerville 2004 - Software Engineering, 7th edition. Chapter 8

Library semantic model



Ian Sommerville 2004 - Software Engineering, 7th edition. Chapter 8