Module 3

Domain Class Diagrams

Vasco Amaral
vma@fct.unl.pt

You know… classes
You have been working with them for 3 years…

Keep in mind all you know about them. It remains relevant.
Keep an open mind. Today you will see them from a different perspective.

Objects

(quick reminder of what you learned in Introduction to Programming, just in case...)

What is an object?

- A discrete entity with a well-defined boundary that **encapsulates state and behavior**; an **instance of a class**.
  - Objects combine data and function in a cohesive unit
  - Objects **hide data behind a layer of operations**
    - This is known as **encapsulation**, or data hiding
    - Encapsulation is good style, although not mandatory

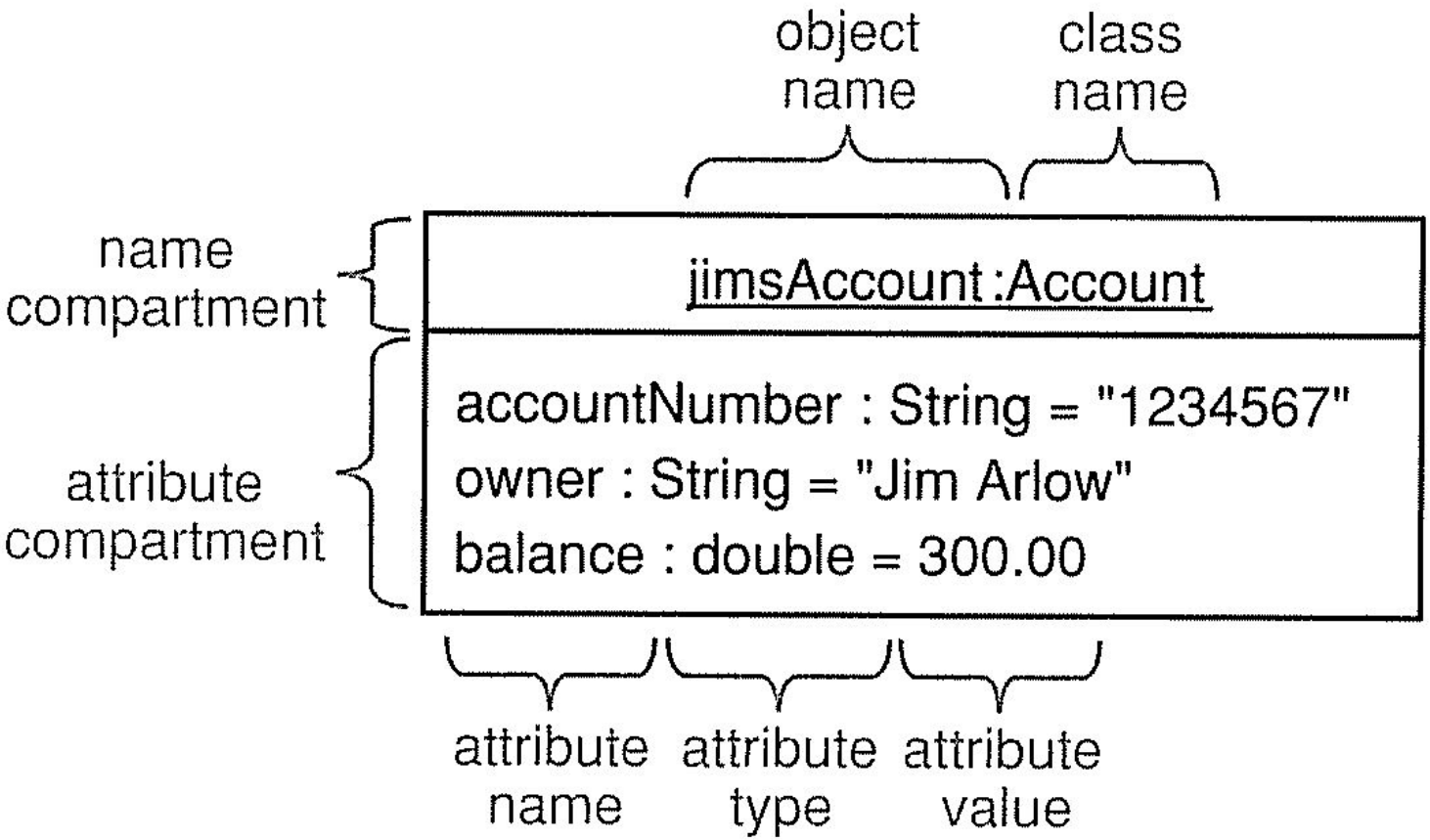
Object properties

- Identity
  - Every object is uniquely identifiable
- State
  - The current state of an object is determined by the values of its attributes and the relationships to other objects in a given point in time
- Behavior
  - The behavior of an object is characterized by its operations
    - Some operations modify the object's state
    - Other operations allow querying the object about its state

Objects collaborate to generate system behavior

- Objects form links among themselves and send messages back and forth through those links
- When an object receives a message, it checks its set of operations looking for an operation whose signature matches the one in the message
  - If a match exists, the object invokes the operation and, possibly, returns a result to the object which sent the message

Objects in UML



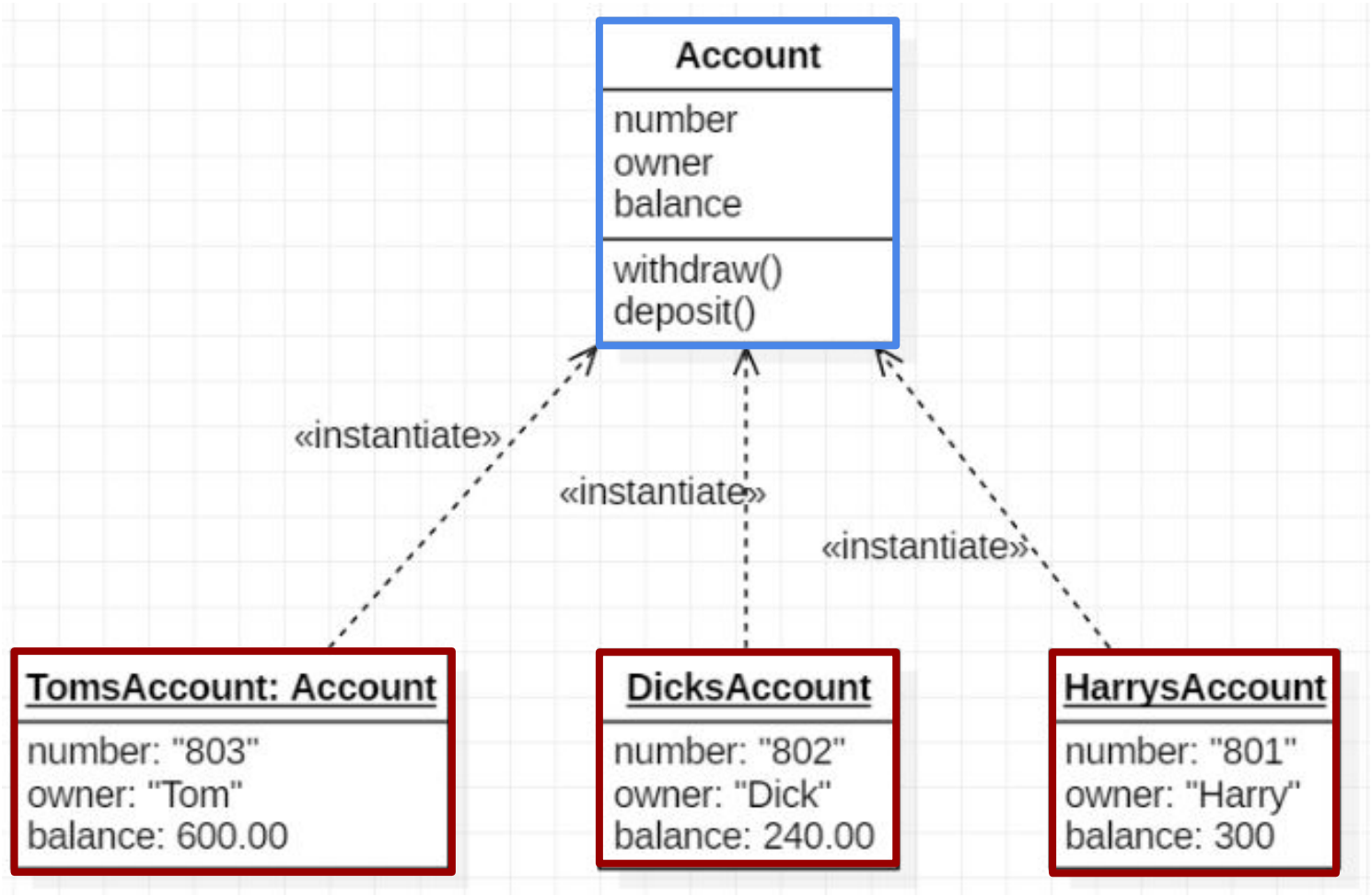
Classes

(quick reminder of what you learned in Introduction to Programming, just in case...)

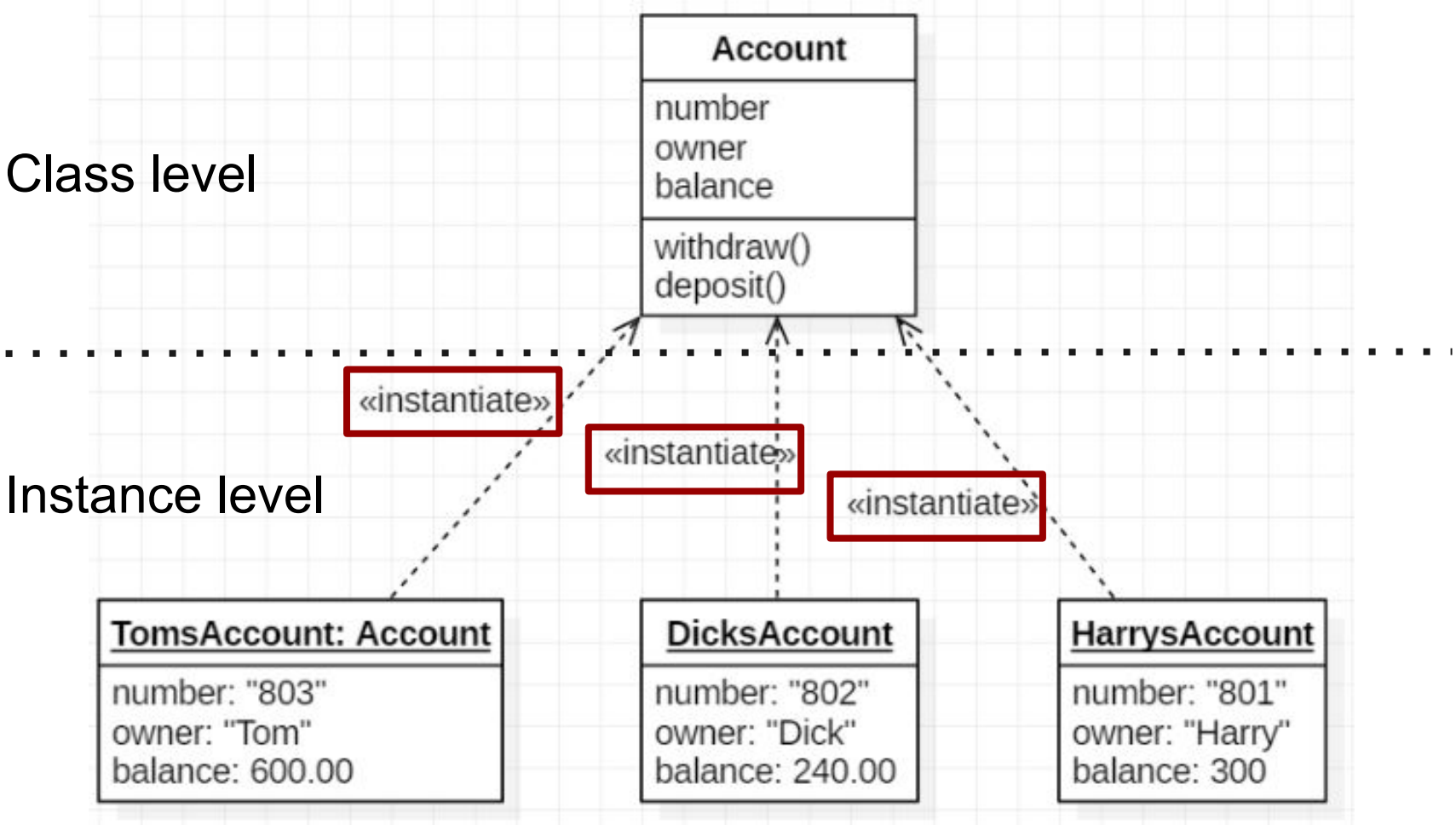
What is a class?

- The descriptor for a set of **objects that share the same attributes, operations, methods, relationships, and behavior**
- Every object is an instance of exactly one class
  - Objects of the same class share a common structure
    - The same set of operations
    - The same set of attributes (but different values)
    - The same set of relationships

Objects are instantiations of classes



The <<instantiate>> stereotype classifies the kind of association between objects and classes



Analysis classes

Analysis classes

- Represent a crisp abstraction of the problem domain
- Model important aspects of the problem domain, such as “customer”, or “product”
- Should map clear in a **clear and unambiguous way to real-world business concepts**
  - This includes having adequate names for the classes representing those concepts

Analysis classes

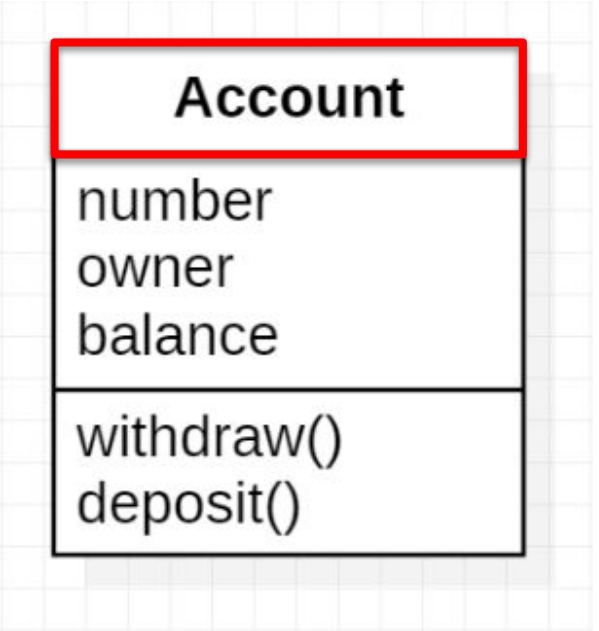
- Often, it is **up to the modeller to clarify** confused, or inappropriate business concepts into more adequate analysis classes
- The analysis model should **only contain analysis classes**
  - Classes arising from design considerations are NOT part of the analysis model
- Later, during design, these analysis classes will be refined into one or more design classes

Analysis classes form the Domain Class Diagram

- Domain class diagram contains the set of base classes of the problem (**classes of type entity**)
- This diagram will be later extended to contemplate other kinds of classes and dependencies between them
- **Types and arguments for operations come later**

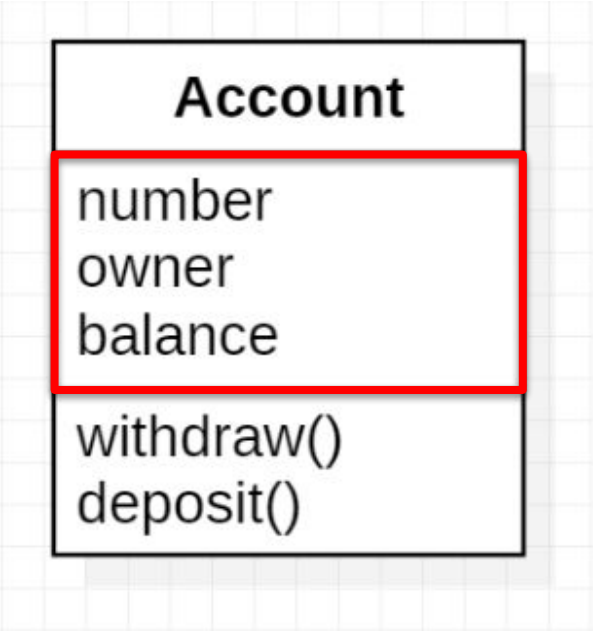
A class has a name

- The **name is a mandatory** feature of the class
- It **should be a noun** from the domain vocabulary
- Different classes with the **same name are only possible if they are defined in different packages**
- The name can be qualified by the package name (e.g. Bank::Account, where Bank is the package and Account the class)



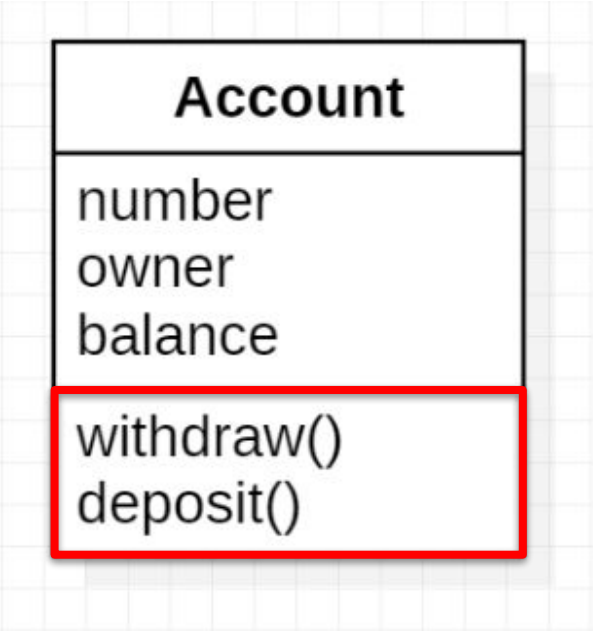
A class has attributes

- Only an important **subset of candidate attributes** should be modelled during analysis
- **Attribute names are mandatory**
  - Typically, use nouns that make sense in the particular domain
- Attribute types are optional, during analysis
  - Use them to clarify the domain model, when adequate



A class has operations

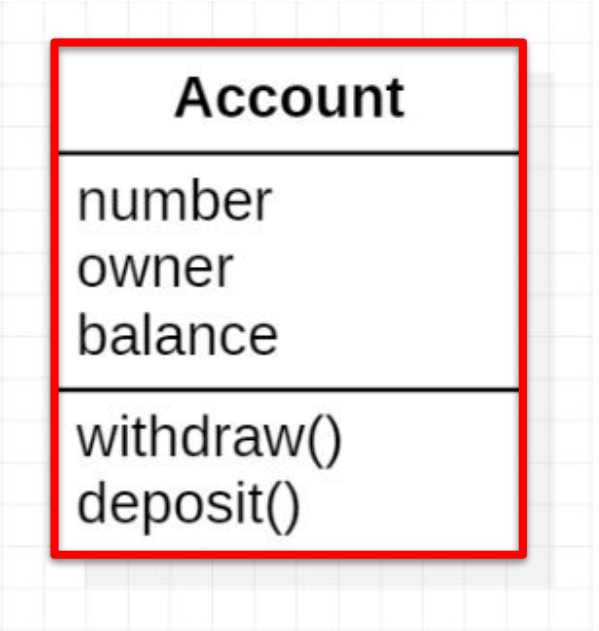
- An operation name is usually a **verb** to represent the class behavior
- Operations are **abstractions of something the object can do**
- Operations are shared by all instance objects of the class
- Only high-level operations are required, during analysis, to clarify the responsibilities of the class
- Parameters and return types are optional and should only be used if they help understanding the model



An analysis class often hides many details

And so do we. :-)
Just so you know, the following are hidden, for the time being:

- **Visibility** of attributes and operations
- **Stereotypes** - only shown if they help understanding the model
- **Tagged values** - only shown if they help understanding the model



We will get back to these later!

What makes a good analysis class?

- Its **name reflects its intent**
- It is a crisp abstraction that models one specific element of the problem domain
  - Must have a clear and obvious semantics
- It maps on to a clearly identifiable feature of the problem domain

What makes a good analysis class?

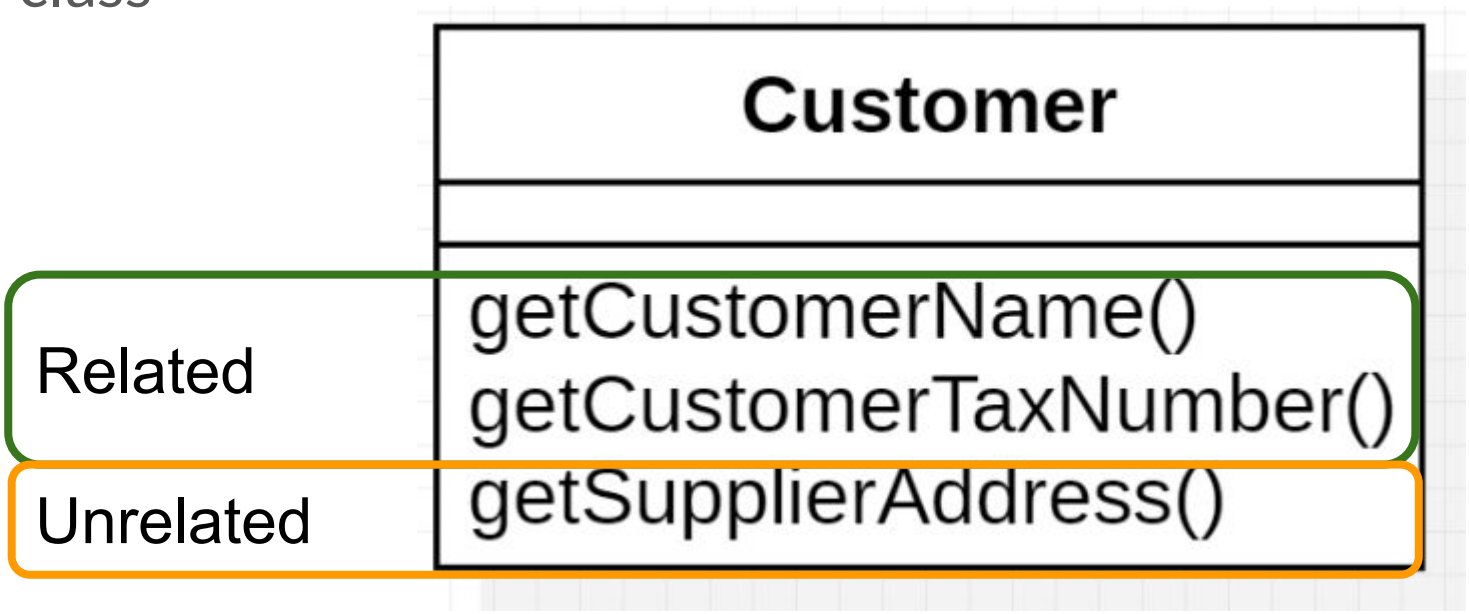
- It has a small, **well defined, set of responsibilities**
- Offers a clear separation between:
  - Specification and implementation
  - What is visible and hidden
- Has **high cohesion**
- Has **low coupling** to other classes

Responsibilities describe cohesive sets of operations

- A responsibility is a contract, or obligation, that the class has to its clients
- It is a service provided to other classes
- Each class should have a cohesive set of **responsibilities closely related to the intent of the class** (expressed by the class name)

Responsibilities describe cohesive sets of operations

- If you find a subset of responsibilities that break this cohesion and do not really match the intent of the class, you may have found a candidate for another class



Responsibilities describe cohesive sets of operations

- If you find a subset of **responsibilities that break** this cohesion and do not really match the intent of the class, you may have **found a candidate for another class**
- An even distribution of responsibilities among classes tends to lead to relatively low coupling among those classes
  - Localization of control, or too many responsibilities will increase coupling with the class where it occurs

Hints & Tips for creating well-formed analysis classes

Keep classes as simple as possible

- About **3 to 5 responsibilities per class**
- The class should be focused, with a small and manageable set of responsibilities

Classes should not be isolated

- Classes are supposed to collaborate with each other to provide benefits for to the system
- Isolated classes do not collaborate
- Each class should **collaborate with a small number of other classes**
- Classes may delegate part of their responsibilities to “helper” classes

Beware of too many too small classes

- Modelling is a balancing act
- **Too many, too small classes** (with just one or two responsibilities each) may **indicate that you are missing an opportunity to consolidate** some of those small classes into larger ones

Beware of too few too big classes

- Modelling is a balancing act
- **Too few, too big classes** (with over 5 responsibilities each) may indicate that you are creating classes which are not cohesive
- You may find opportunities to **decompose** some of these **into more cohesive classes**, with a more manageable set of responsibilities each

Beware of “functoids”

- A functoid is a normal procedural function disguised as a class
- These are relatively common when analysts originally trained with the technique of top-down functional decomposition start using OO for the first time
  - This difficulty in changing paradigms phenomenon is known as the *paradigm shift problem*

Beware of God classes

- These classes seem to do everything
  - Classes with “system”, or “controller”, in their names are often candidates
- Look up for cohesive subsets of responsibilities within these classes and factor them out accordingly into smaller classes
  - These classes should then collaborate to implement the behavior originally offered by the God class

Beware of deep inheritance trees

- In a good inheritance hierarchy, each level should have a well-defined purpose
- A common mistake is to use inheritance for functional decomposition where each level has a single responsibility
- In analysis, inheritance should be used when there is a clear and obvious hierarchy derived directly from the problem domain
- Business classes tend to form broader, rather than deeper, hierarchies

So, how to identify classes?

- Noun/verb analysis
- CRC analysis
- RUP technique

Noun/verb analysis

Finding classes through noun/verb analysis

- Analyse a textual specification of the requirements
- Nouns and noun phrases help to identify classes or attributes
- Verbs and verb phrases help to identify responsibilities or operations of a class
- Beware of synonyms and homonyms, as these can create confusion

Finding classes through noun/verb analysis

- Requires domain knowledge
- This technique is
  - Good for identifying classes
  - Bad for identifying their requirements (characteristics and relations)

Noun/verb analysis procedure

1. **Collect relevant documentation**
   - requirements (text/model), use case model, project glossary, …
2. Highlight nouns/noun phrases (candidate classes and attributes)
3. Highlight verbs/verb phrases (candidate responsibilities)
4. Clarify terms you are not familiar with, with the help of domain experts
5. Use a project glossary to collect these terms, identify synonyms and homonyms - candidate classes, attributes and responsibilities
6. Tentatively allocate attributes and responsibilities to classes
7. In this process, try to identify relationships between classes (use cases are a good starting point for these)

Identify the candidate classes and attributes (nouns and noun phrases)

Bank customers can debit and credit amounts in their bank accounts, or ask for their current balance. These operations may be performed in ATM machines, or in the bank counter. The transactions on a bank account are performed via a bank check, or using the ARM machines with a card. There are two kinds of bank accounts: current account and savings account. A savings account pays interest and cannot be accessed via the ATM machines.

Identify the candidate classes and attributes (nouns and noun phrases)

Bank customers can debit and credit amounts in their bank accounts, or ask for their current balance. These operations may be performed in ATM machines, or in the bank counter. The transactions on a bank account are performed via a bank check, or using the ATM machines with a card. There are two kinds of bank accounts: current account and savings account. A savings account pays interest and cannot be accessed via the ATM machines.

Identify the candidate operations (verbs and verb phrases)

Bank customers can debit and credit amounts in their bank accounts, or ask for their current balance. These operations may be performed in ATM machines, or in the bank counter. The transactions on a bank account are performed via a bank check, or using the ATM machines with a card. There are two kinds of bank accounts: current account and savings account. A savings account pays interest and cannot be accessed via the ATM machines.

So, which of these are classes, attributes and responsibilities?

CRC analysis

Finding classes with CRC analysis

- CRC stands for Class, Responsibilities and Collaborators
- CRC is a brainstorming technique in which you capture on sticky notes the important things in the problem domain.
- Commonly done with sticky notes (post-its)
- Record in each note the class, its responsibilities and collaborators
- Sticky notes in a whiteboard and draw lines between collaborating classes, to identify candidate relationships
- Typically, this is used in conjunction with noun / verb analysis of use cases, requirements, glossary, etc.

Class Name: Bank Account	
Responsibilities: Maintain balance	Collaborators: Bank

CRC Analysis procedure - Brainstorm

Phase 1: Brainstorm - gather information
Participants: OO analysts, stakeholders, domain experts, and a facilitator

CRC Analysis procedure - Brainstorm

Phase 1: Brainstorm - gather information

Participants: OO analysts, stakeholders, domain experts, and a facilitator

1. Explain participants this is a true brainstorm
    a. All ideas are accepted as good ideas
    b. Ideas are recorded, but not debated - never argue about something, just write it down and move on

CRC Analysis procedure - Brainstorm

Phase 1: Brainstorm - gather information

Participants: OO analysts, stakeholders, domain experts, and a facilitator

1. Explain participants this is a true brainstorm
   a. All ideas are accepted as good ideas
   b. Ideas are recorded, but not debated - never argue about something, just write it down and move on
2. Ask team members to name the “things” that operate in their business domain - for example, customer, or product
   a. Write each “thing” on a sticky note: it is a candidate class or attribute of a class
   b. Stick the note on a wall, or whiteboard

CRC Analysis procedure - Brainstorm

Phase 1: Brainstorm - gather information

Participants: OO analysts, stakeholders, domain experts, and a facilitator

1. Explain participants this is a true brainstorm
   a. All ideas are accepted as good ideas
   b. Ideas are recorded, but not debated - never argue about something, just write it down and move on
2. Ask team members to name the “things” that operate in their business domain - for example, customer, or product
   a. Write each “thing” on a sticky note: it is a candidate class or attribute of a class
   b. Stick the note on a wall, or whiteboard
3. Ask team members to state responsibilities those things might have and record those in the corresponding compartment in the sticky note

CRC Analysis procedure - Brainstorm

Phase 1: Brainstorm - gather information

Participants: OO analysts, stakeholders, domain experts, and a facilitator

1. Explain participants this is a true brainstorm
   a. All ideas are accepted as good ideas
   b. Ideas are recorded, but not debated - never argue about something, just write it down and move on
2. Ask team members to name the “things” that operate in their business domain - for example, customer, or product
   a. Write each “thing” on a sticky note: it is a candidate class or attribute of a class
   b. Stick the note on a wall, or whiteboard
3. Ask team members to state responsibilities those things might have and record those in the corresponding compartment in the sticky note
4. Ask team members to identify classes that might work together; rearrange notes on the whiteboard to reflect this organization and draw lines between cooperating classes, or record collaborators in the corresponding compartment

CRC Analysis procedure - Analyze information

Phase 2: Analyze information

Participants: OO analysts, domain experts

CRC Analysis procedure - Analyze information

Phase 2: Analyze information

Participants: OO analysts, domain experts

- Decide which sticky notes should be classes (those which represent key business concepts)

CRC Analysis procedure - Analyze information

Phase 2: Analyze information
Participants: OO analysts, domain experts

- Decide which sticky notes should be classes (those which represent key business concepts)
- Other notes may become classes, or attributes
  - If a note seems to be logically a part of another note, it is probably representing an attribute
  - If a note does not seem particularly important, or has little interesting behavior, maybe it can be an attribute for another class

CRC Analysis procedure - Analyze information

Phase 2: Analyze information
Participants: OO analysts, domain experts

- Decide which sticky notes should be classes (those which represent key business concepts)
- Other notes may become classes, or attributes
  - If a note seems to be logically a part of another note, it is probably representing an attribute
  - If a note does not seem particularly important, or has little interesting behavior, maybe it can be an attribute for another class
- If you really do not know what to do with a note, make it a class

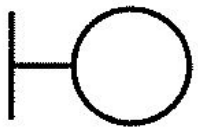
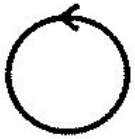
CRC Analysis procedure - Analyze information

Phase 2: Analyze information
Participants: OO analysts, domain experts

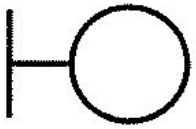
- Decide which sticky notes should be classes (those which represent key business concepts)
- Other notes may become classes, or attributes
  - If a note seems to be logically a part of another note, it is probably representing an attribute
  - If a note does not seem particularly important, or has little interesting behavior, maybe it can be an attribute for another class
- If you really do not know what to do with a note, make it a class
- Make a best guess and get this process to closure - this is just a first cut model, you will come back to it to refine it, later

RUP stereotypes

Consider three distinct types of analysis classes

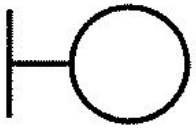
Stereotype	Icon	Semantics
<<boundary>>		A class that mediates interaction between the system and its environment
<<control>>		A class that encapsulates use-case-specific behavior
<<entity>>		A class that is used to model persistent information about something

Identifying <<boundary>> classes



- These classes are in the boundary of your system and **communicate with external actors** that you identified through Use Case Analysis
- Communications between actors and your system must be enabled by some <<boundary>> class instance
- There are **three types of <<boundary>> classes**
  - **User interface classes -** classes that interface between the system and humans
  - **System interface classes -** classes that interface with other systems
  - **Device interface classes -** classes that interface with other devices - e.g. sensors

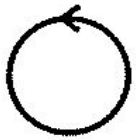
Identifying <<boundary>> classes



- When a boundary class services more than one actor, these actors should be of the same kind (a human, a system, or a device)
  - If they are of different kinds, there is probably something wrong
- Keep at a high abstraction level
  - You are concerned with identifying the class, not with its specific details
  - Do not model interface details here - a dummy (user/system/device) interface class will do just fine

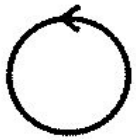
You will fill in those details during design, not now!

Identifying <<control>> classes



- <<control>> classes coordinate system behavior during one or more use cases
- Consider the behavior of the system as described by the use cases, and then work out how these responsibilities can be partitioned among several <<control>> classes
  - Simple behavior may be distributed among <<boundary>> or <<entity>> classes
  - More complex behavior is better localized in a <<control>> class

Don't be a <<control>> freak! :-)



- **<<control>>** classes should arise naturally from the problem domain
- Do not "artificially" create a **<<control>>** class for each use case
  - You are NOT trying to analyse this system through functional decomposition
- **<<control>>** classes tend to cross-cut several use cases
- Sometimes, a single use case is better modelled with several **<<control>>** classes
- If a **<<control>>** class is too complex, and if you find it to be not as cohesive as one would like, this may be a good hint to break it into a set of cohesive **<<control>>** classes

Identifying <<entity>> classes

- **<<entity>>** classes model something with simple behavior that mostly consists of getting and setting values. They:
  - Crosscut several use cases
  - Manipulated by <<control>> classes
  - Provide information to and accept information from boundary classes
  - Represent key things managed by the system
  - Are often persistent
- **<<entity>>** classes express the logical data structure of the system
  - They are closely related to entities, or tables, in a data model, when there is a data model

Finding classes from other sources

- Physical objects, such as aircraft, or person, may indicate an object
- Paperwork (invoices, orders, …) can be a good source as well
  - Be careful not to replicate in the system excessive paperwork that the system is supposed to simplify
- Known interfaces to the outside world (e.g. screens, keyboards) are candidate classes, particularly for embedded systems
- Conceptual entities are crucial, even if they do not manifest as physical objects
  - Look up for cohesive abstractions

Create your first-cut model

01	Compare all sources of classes	• Noun/verb analysis • CRC cards • RUP • Other sources
02	Consolidate classes, attributes and responsibilities into an analysis tool	• Project glossary for synonyms and homonyms • Look up for inconsistencies from the different sources and solve them
03	Collaborators (the lines linking the sticky notes) represent associations among classes	• We will look into these in a moment
04	Improve names to adhere to any relevant guidelines that may apply	• If the organization has a particular set of guidelines, use it • Otherwise, use a generic one

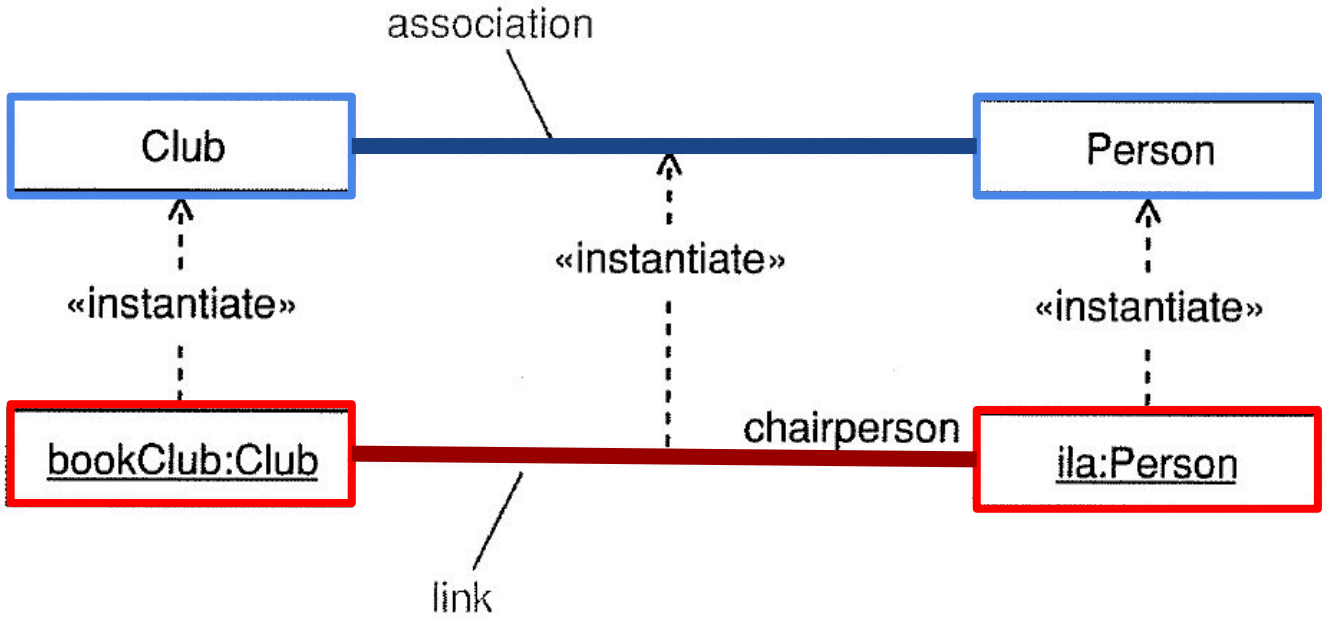
Relationships

What is a relationship?

- UML relationships connect model elements
- You have seen a few examples so far
  - Actors to Actors
    - Generalization
  - Actors to Use Cases
    - Association
  - Use Cases and Use Cases
    - Generalization, <<include>>, <<extend>>
  - Activities
    - Control flow, data flow

Relationships among classes and objects

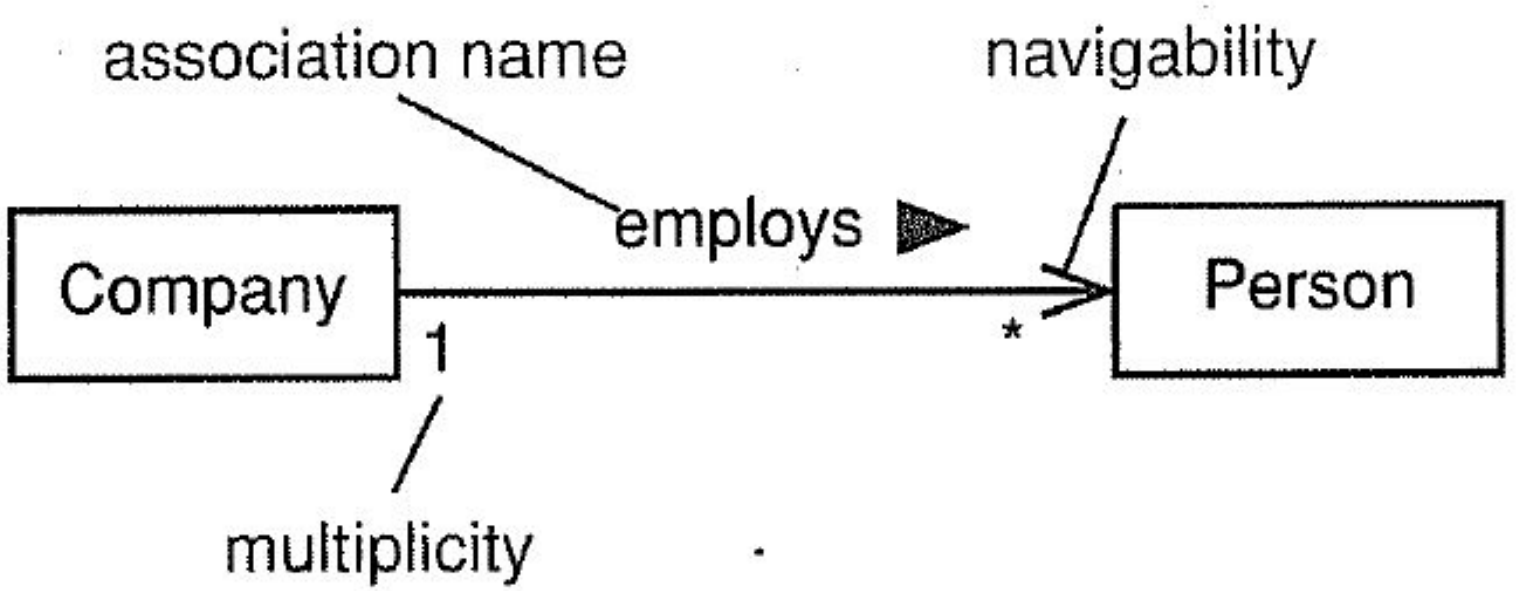
- An **association** is a relationship between **classes**
- A **link** is a relationship between **objects**
- A **link** is an instantiation of an **association**
- Objects instantiate classes just as links instantiate associations



Association syntax

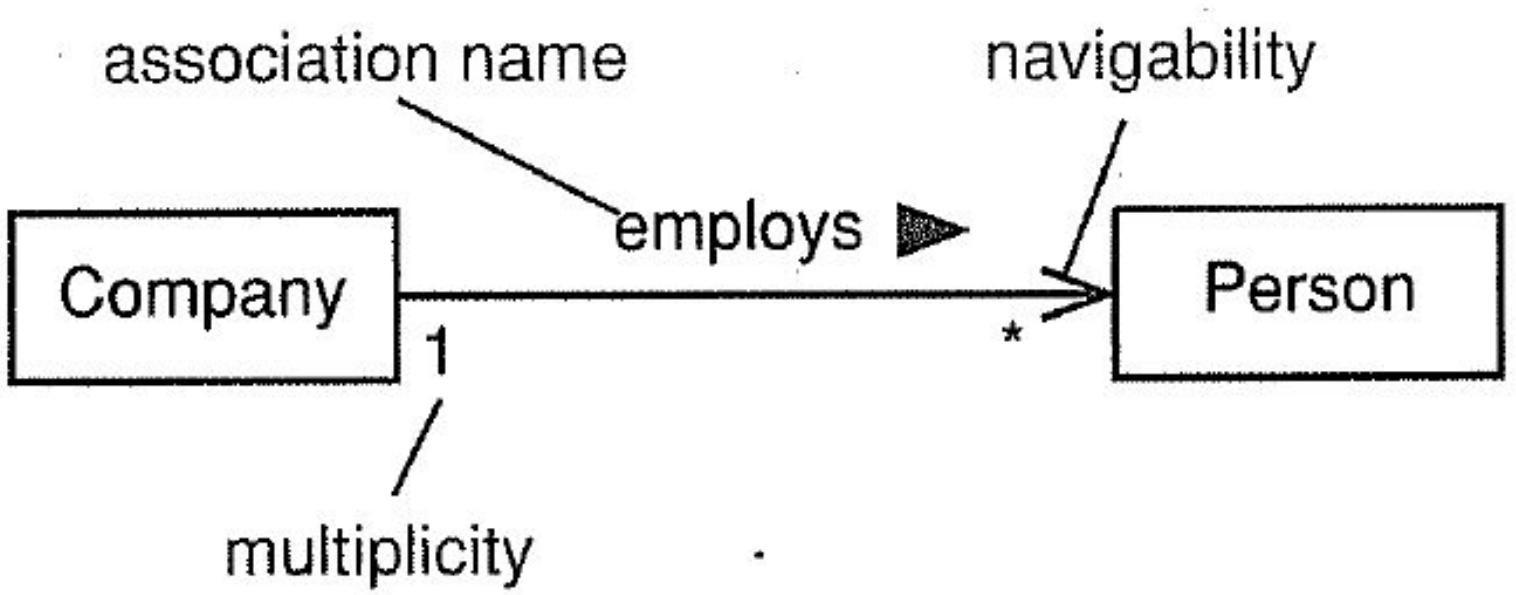
- Association name
  - Verb phrase, indicating an action of the source object to the target object (i.e. the semantics of the association)
    - May be annotated with a small black arrowhead to denote direction
- Role name
  - Noun phrase indicating the role of the objects linked by instances of the association
- Multiplicity
  - Constrains the number of objects of a class participating in a relationship at any point in time
- Navigability
  - If unidirectional, associations should include an arrow

"A company employs many persons"
"A person is employed by exactly one company"



Multiplicity constrains the number of objects of a class participating in a relationship at any point in time

- Person objects can only be employed by one company **at any given time**
- Person objects must **always** be employed by one company
- **Over time**, a Person object might be employed by different companies

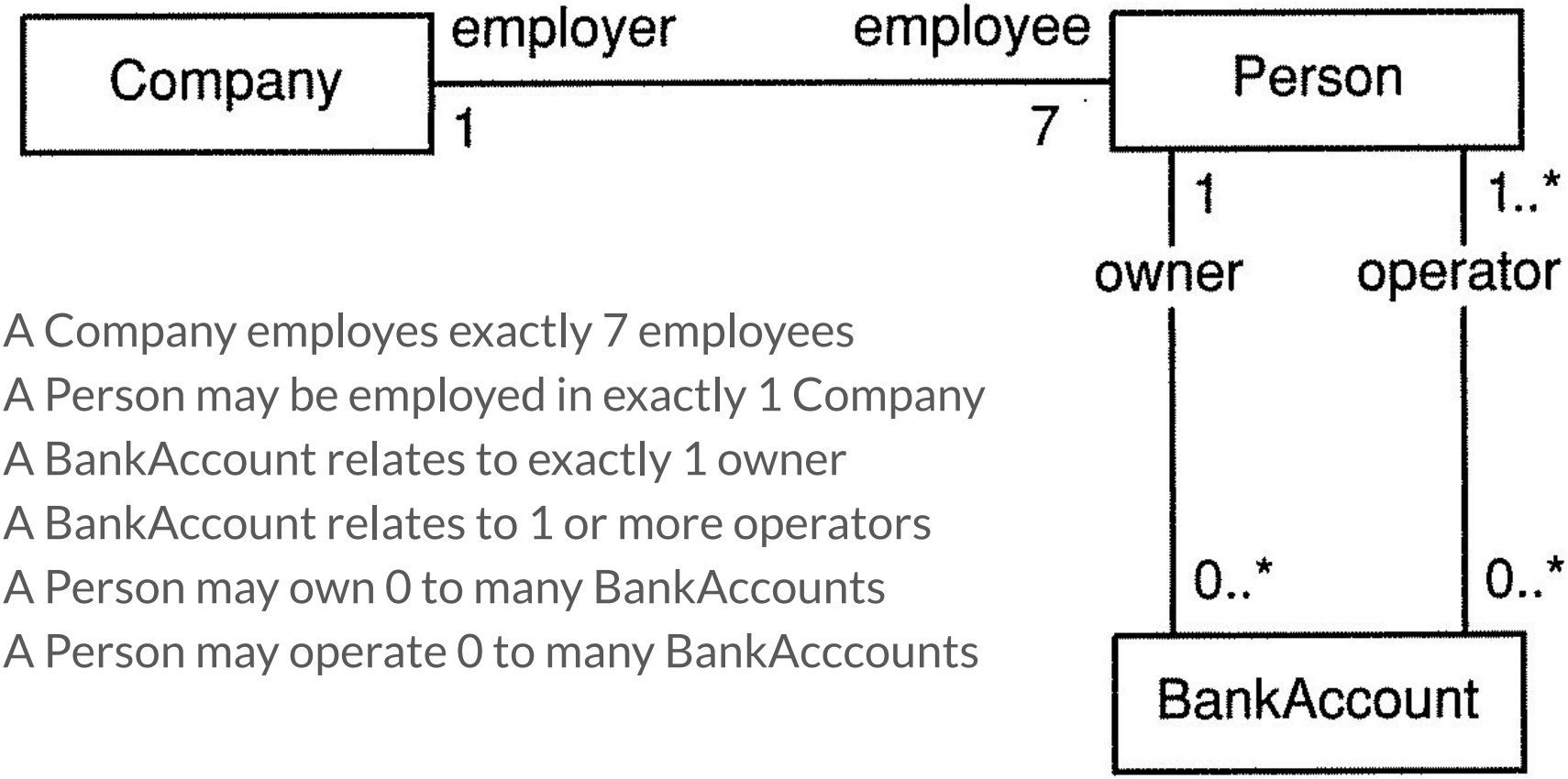


What kinds of multiplicities can we define?

These multiplicity adornments should be placed next to the corresponding classes. When reading a model, you can use these adornments to correctly express the model intention.

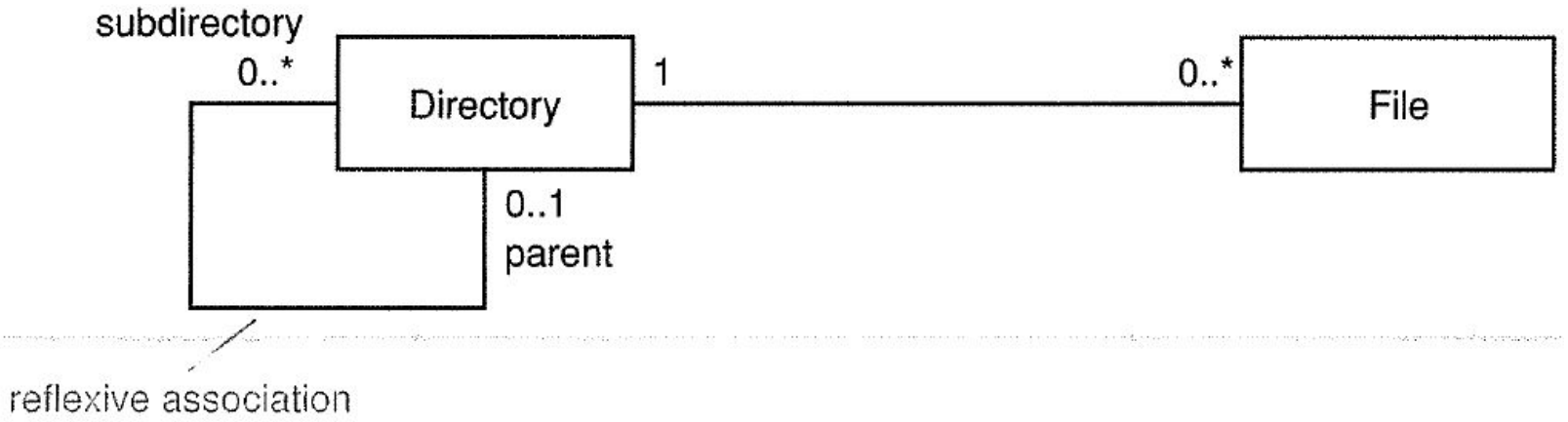
Adornment	Semantics
0..1	0 or 1
0..*	0 or more
*	0 or more
1	Exactly 1
1..*	1 or more
1..4	1 to 4
1..3, 7..10, 22	1 to 3, 7 to 10, 22

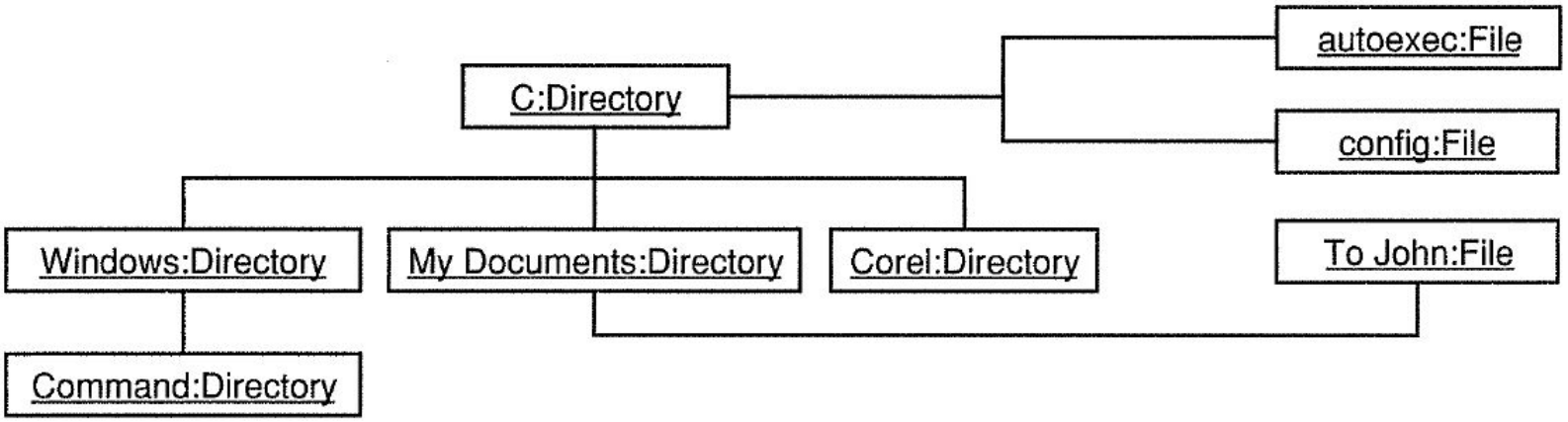
Make sure you read associations and multiplicities precisely



A Company employes exactly 7 employees
A Person may be employed in exactly 1 Company
A BankAccount relates to exactly 1 owner
A BankAccount relates to 1 or more operators
A Person may own 0 to many BankAccounts
A Person may operate 0 to many BankAcccounts

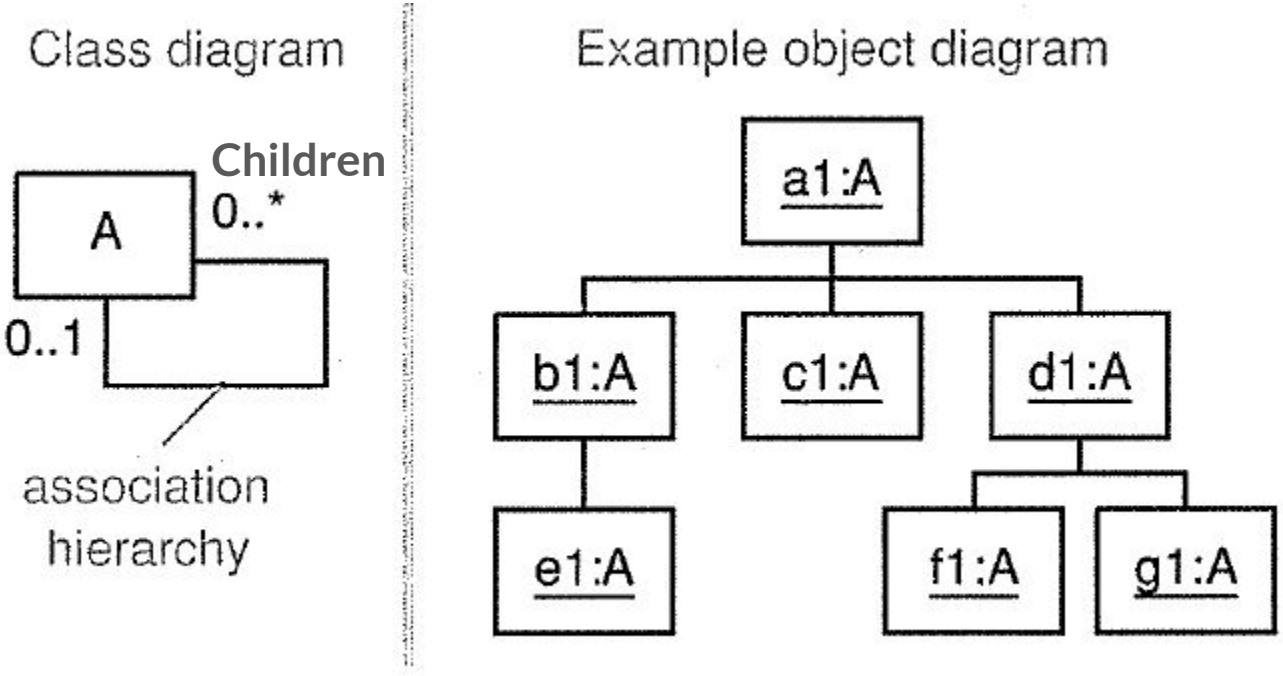
Reflexive associations are associations to the same class



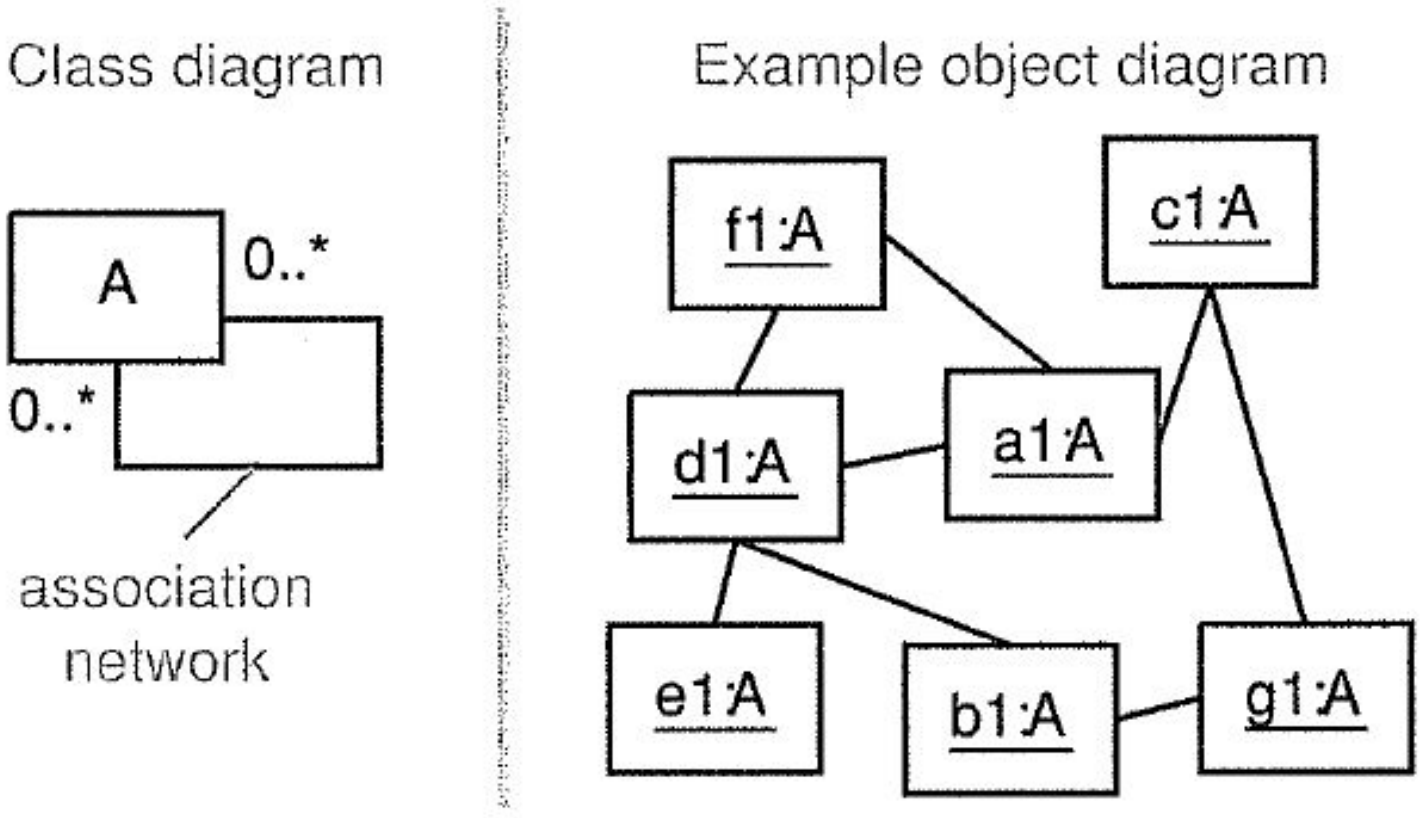


In a hierarchy, an object may have 0 or 1 objects directly above it (parent). The object may have 0 or more children.

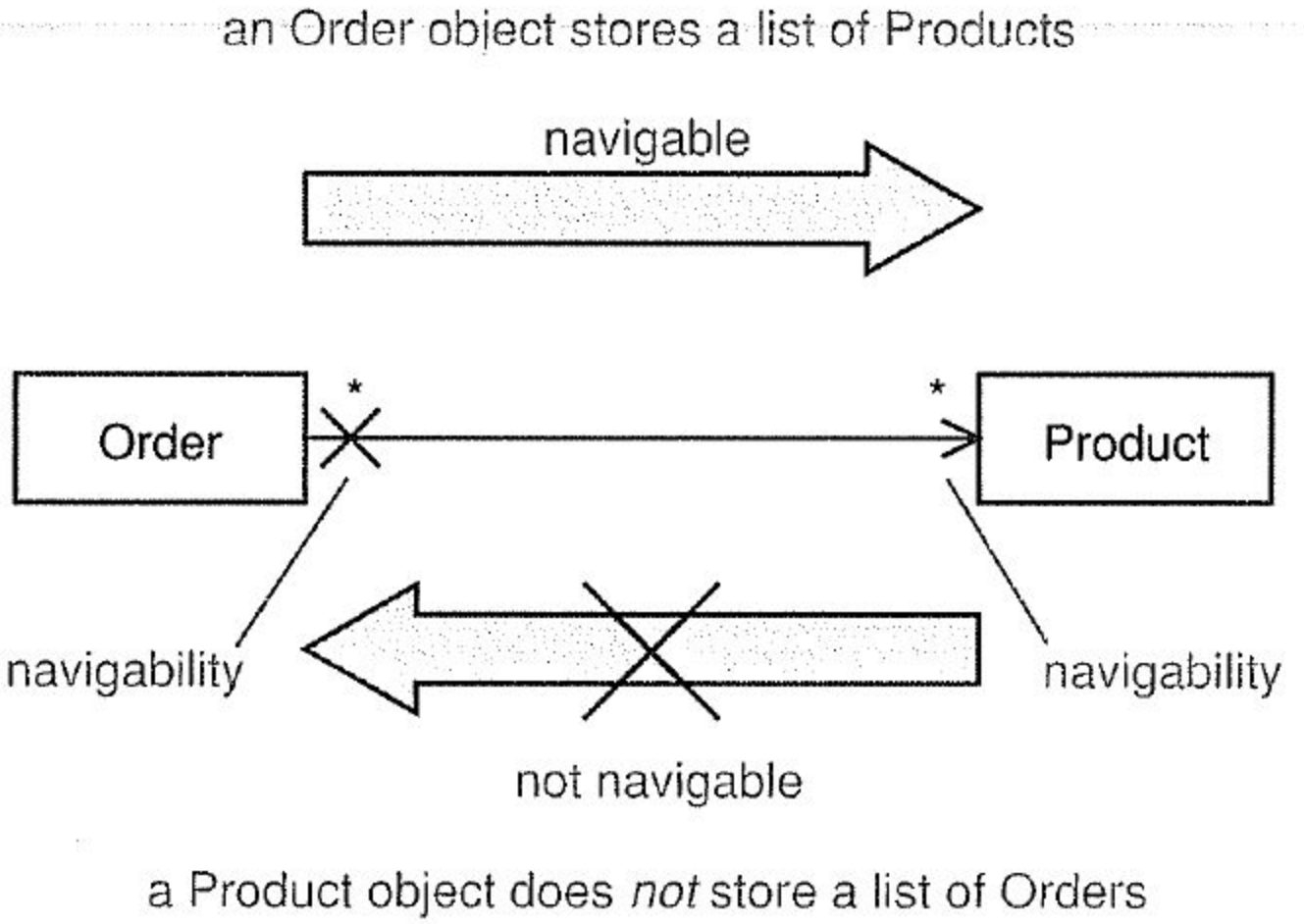
Parent



In a network, each node may have 0 to many objects directly connected to it



Navigability indicates that objects in the source class "know about" objects in the target class

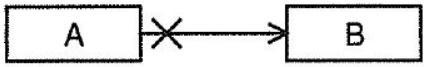
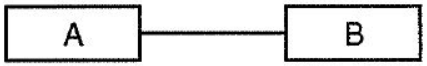


UML specifies 3 alternative idioms for navigability

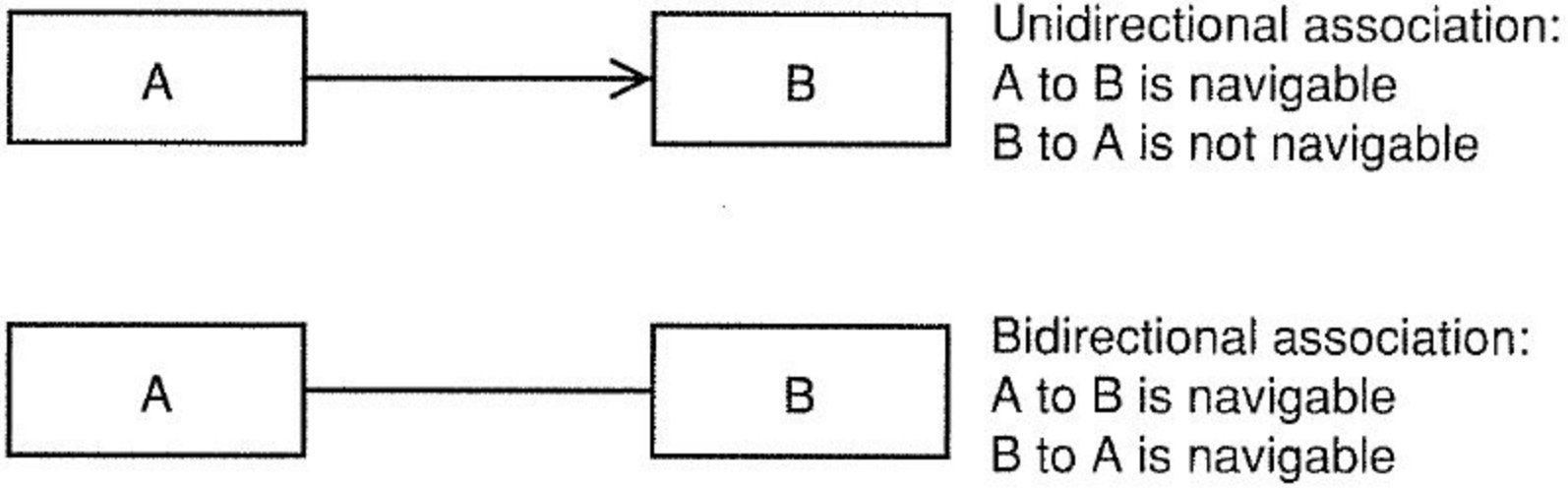
- **Strict UML 2 navigability** - all crosses and arrows are explicitly used
  - This is the most precise alternative
- **No navigability** - no crosses or arrows are used
  - This is the most ambiguous alternative
- **Standard practice** - only arrows are defined
  - This is a compromise idiom, and the one most commonly used, in practice

Comparing the alternative idioms

Adopted in this course

UML 2 navigability idioms			
UML 2 syntax	Idiom 1: Strict UML 2 navigability	Idiom 2: No navigability	Idiom3: Standard practice
	A to B is navigable B to A is navigable		
	A to B is navigable B to A is not navigable		
	A to B is navigable B to A is undefined		A to B is navigable B to A is not navigable
	A to B is undefined B to A is undefined	A to B is undefined B to A is undefined	A to B is navigable B to A is navigable
	A to B is not navigable B to A is not navigable		

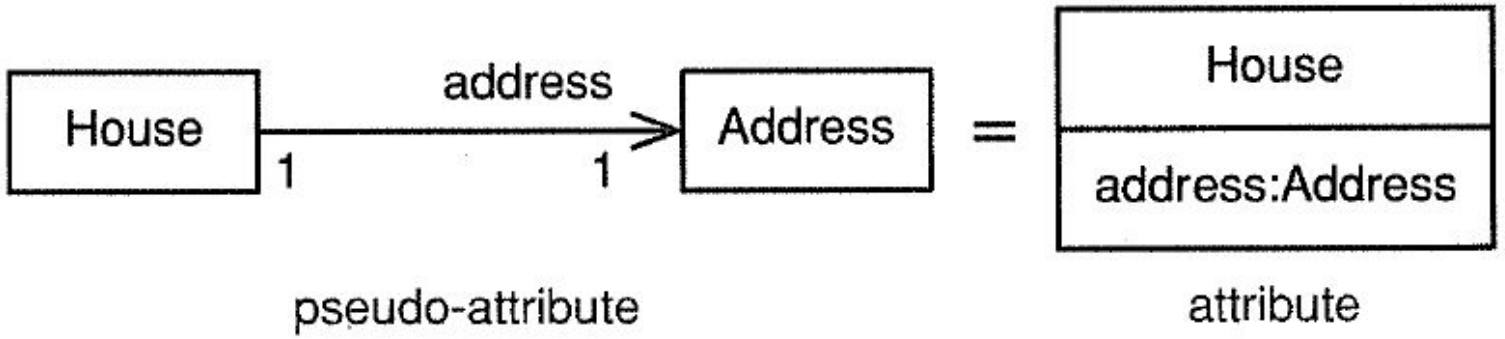
Adopted in this course



Caveats:

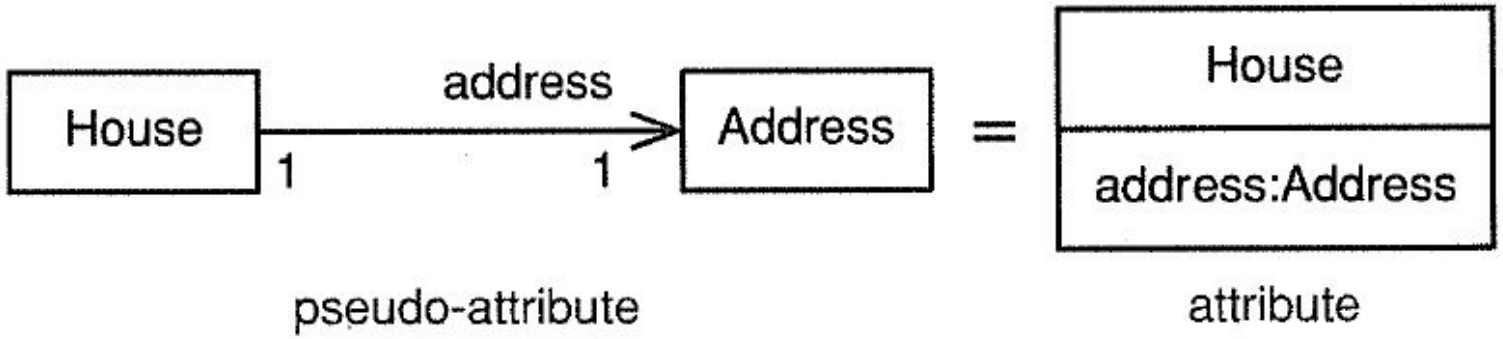
- It is impossible to know if navigability is fully defined, or not, with this idiom
- It changes the semantics of a unidirectional relationship from *Undefined->Navigable* to *NotNavigable->Navigable*
- It is impossible to show both-ways non-navigable associations (but these are not used in practice)

Class associations and class attributes



- The object from the source class has a reference to an object of the target class
- If the association has a role in the target class, that role corresponds to a pseudo-attribute in the source class, which can then be transformed into an attribute
- In this example, the class House has a pseudo-attribute called address, of type Address

This association can be transformed into code!

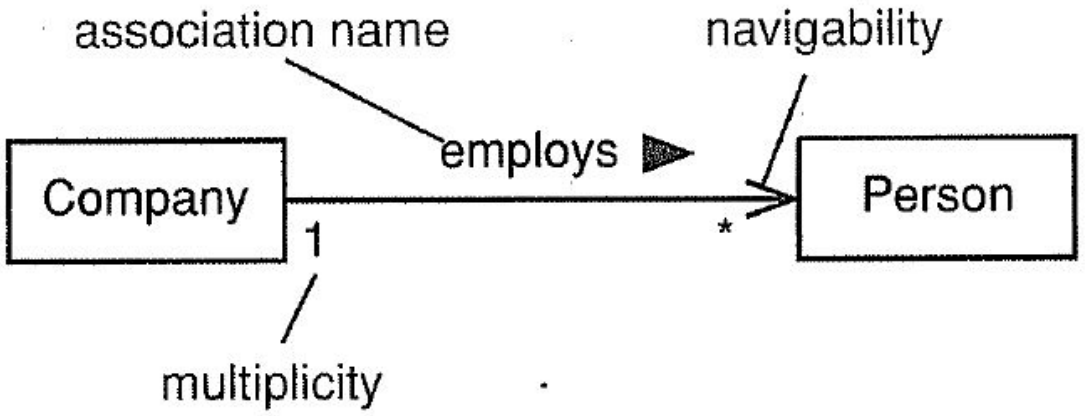



```
public class House {
   Address address;
}

public class Address {
}
```

WARNING:
Although we have just shown you code generated from an analysis class, you should only perform code generation from design class models!

With other cardinalities, you may need a collection of some kind (e.g. Collection, or Array)




```
public class Company {
   Collection<Person> person;
}

public class Person {
}
```

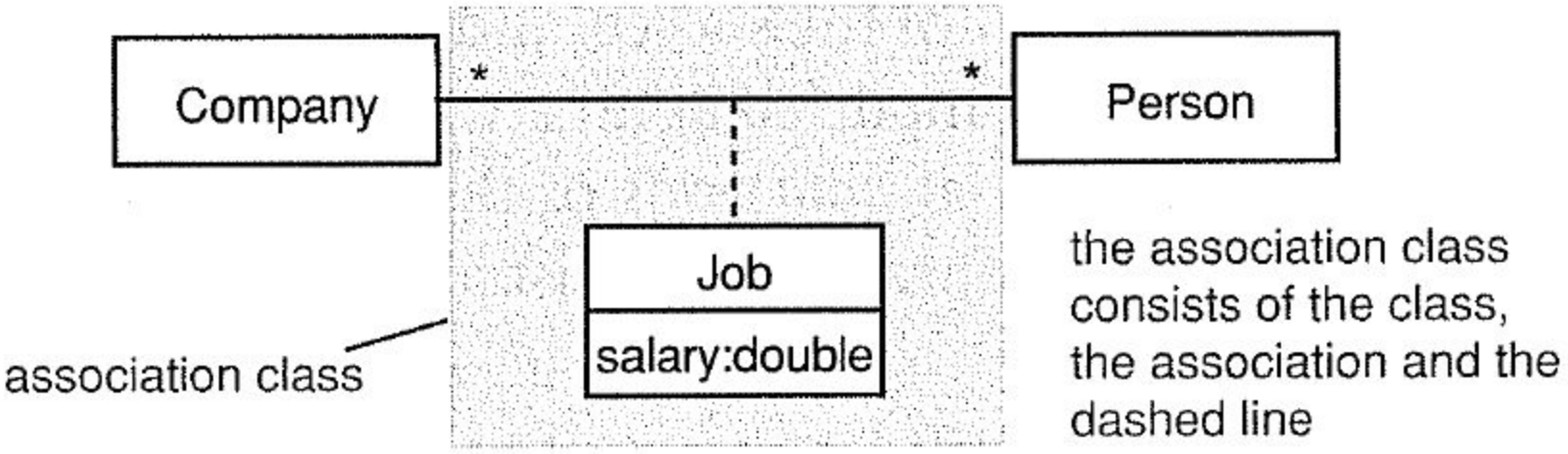
```
public class Company {
   Person[] person;
}

public class Person {
}
```

Why do we need association classes?

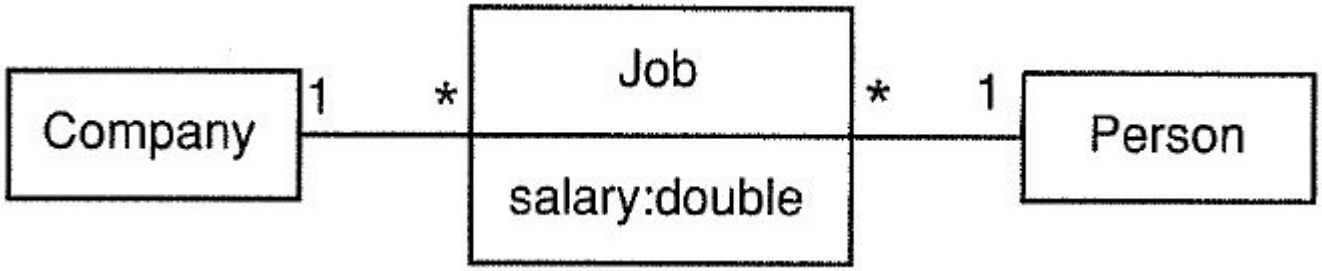
- In many to many associations, what can you do if you need to assign a value (e.g. the salary of the Person to the association?)
  - The Person may work for more than one company
  - If that is so, we cannot store information about the salary in any of the classes
    - If we store the salary in the company, which salary corresponds to which person?
    - If we store the salary in the person and the person works with more than one company, which salary corresponds to that particular

The salary property is best represented as an association class



- There can only be one link between two instances (one of Company, one of Person) at any given point in time
- The salary attribute is stored in an association class called Job
- The association class has, in practice a link to a Company and another to Person
- What if you need to represent persons who work in different companies at the same time?

The association class can be reified (i.e. Job can be transformed into a real class)




```
public class Company {
   Collection<Job> company;
}
```

```
public class Company {
   Collection<Job> company;
}
```

```
public class Job {
   Company company;
   Person person;
   double salary;
}
```

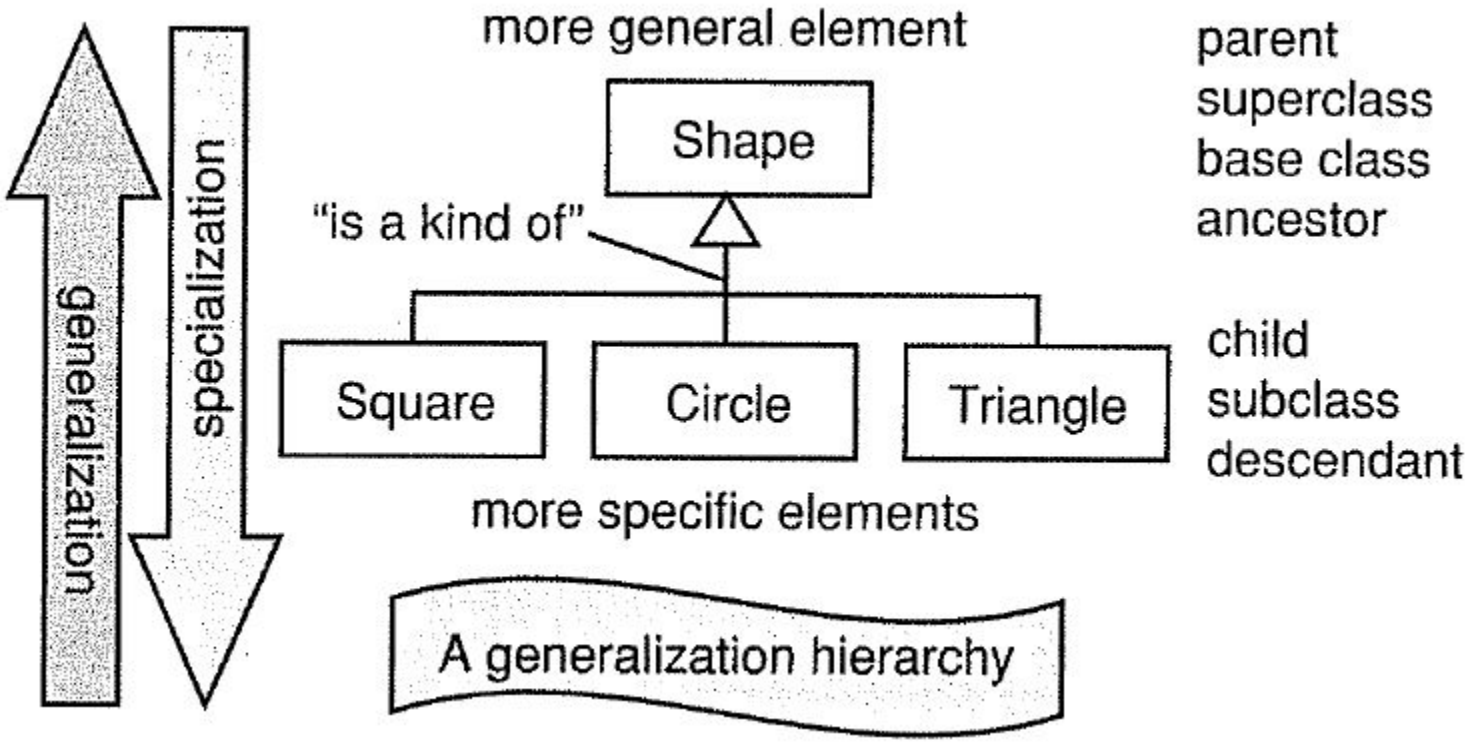
More than one link may exist between a company and a person at any given point in time!

Inheritance and polymorphism

Generalization is a relationship between a more general and a more specific class

- The more specific class is entirely consistent with the more general class
- The two classes obey the Liskov substitution principle
  - We can use the more specific class anywhere the more general class is expected without breaking the system
  - This is a much stronger dependency type than dependency through class associations
  - This mechanism leads to the highest level of coupling between classes

We create a generalization hierarchy by generalizing from more specific classes and specializing from more general classes

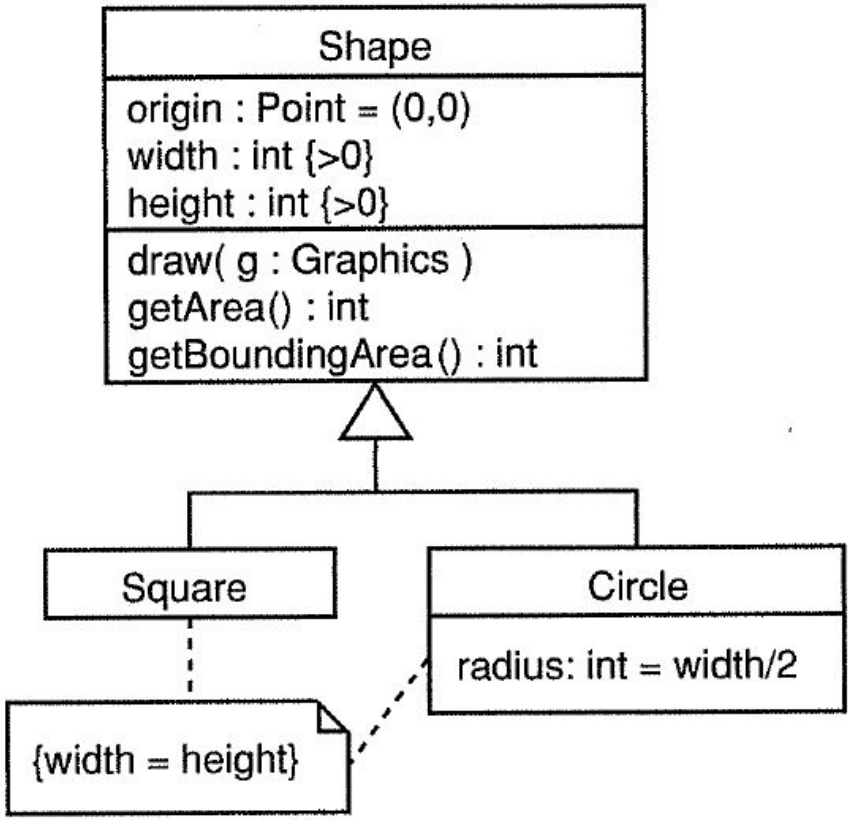


Generalization and inheritance

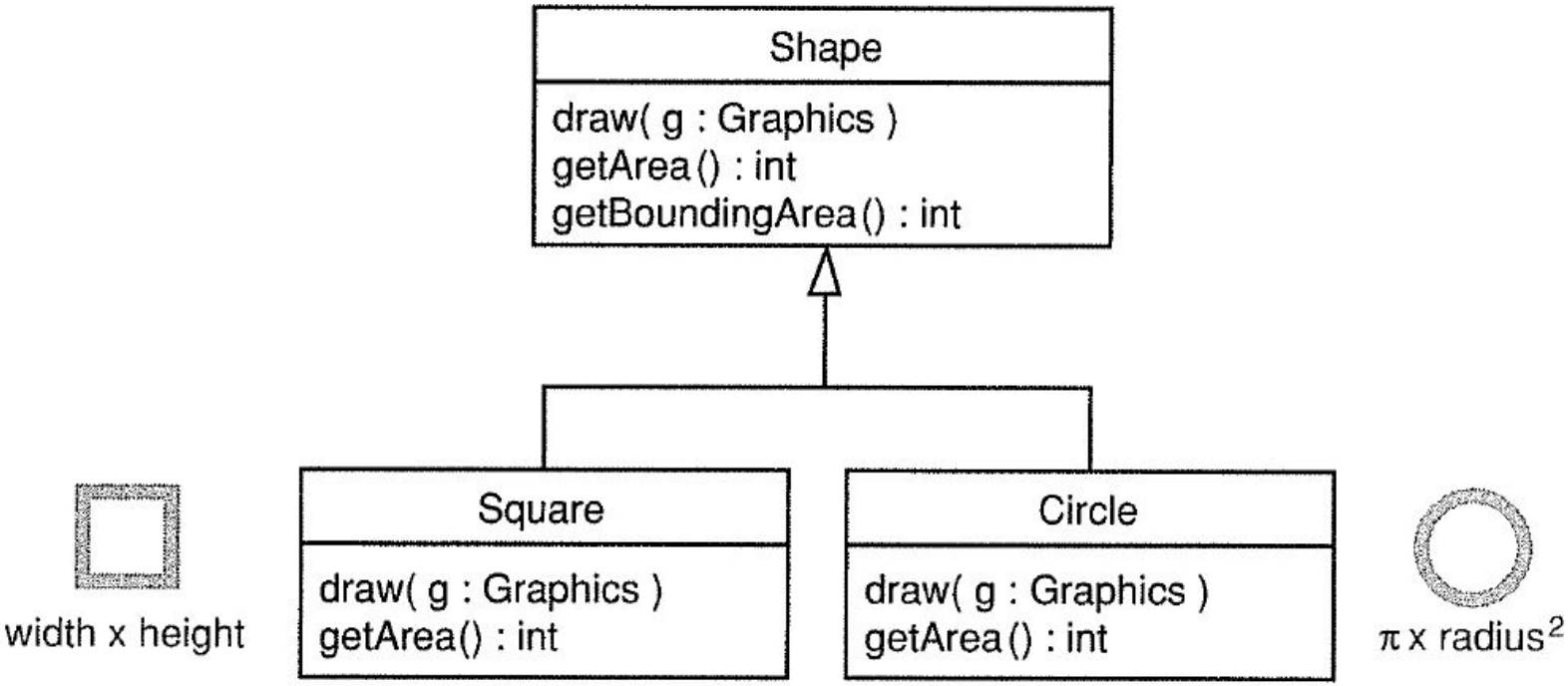
- Classes can be organized hierarchically where the superclass is the generalization of one or more classes (subclasses)
- A subclass inherits the attributes, operations, relationships and dependencies of the superclass
- The subclass can add new features and override other features
- Generalization in UML is implemented as inheritance in OOP

Overriding, and why we might need it...

- `Square` and `Circle` are types of `Shape`
- They inherit features, relationships and constraints from shape
- Inherited elements are not visible in the diagram (they are implicit!)
- However, **features like the `draw()` and `getArea()` operations cannot be generally defined for `Shape`**

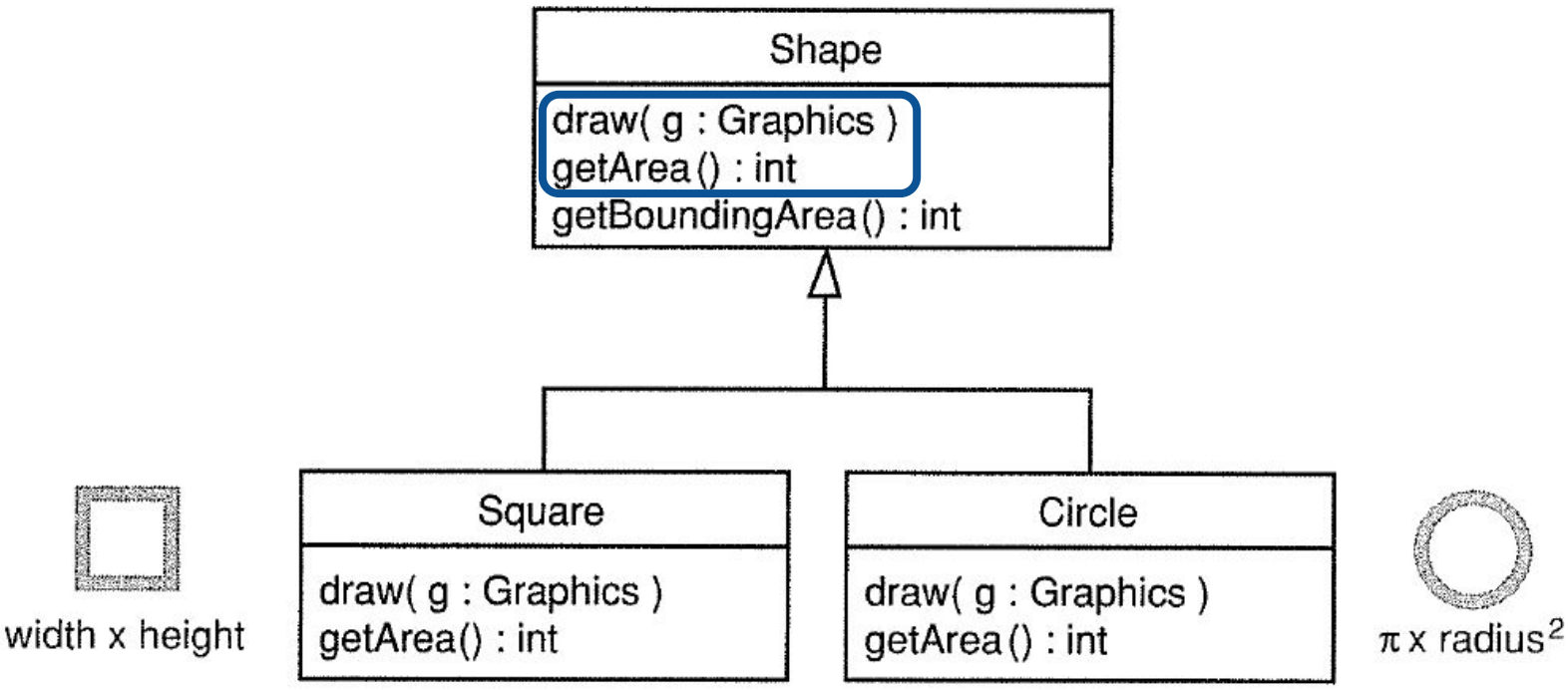


Subclasses override inherited operations by providing a new operation with the same signature



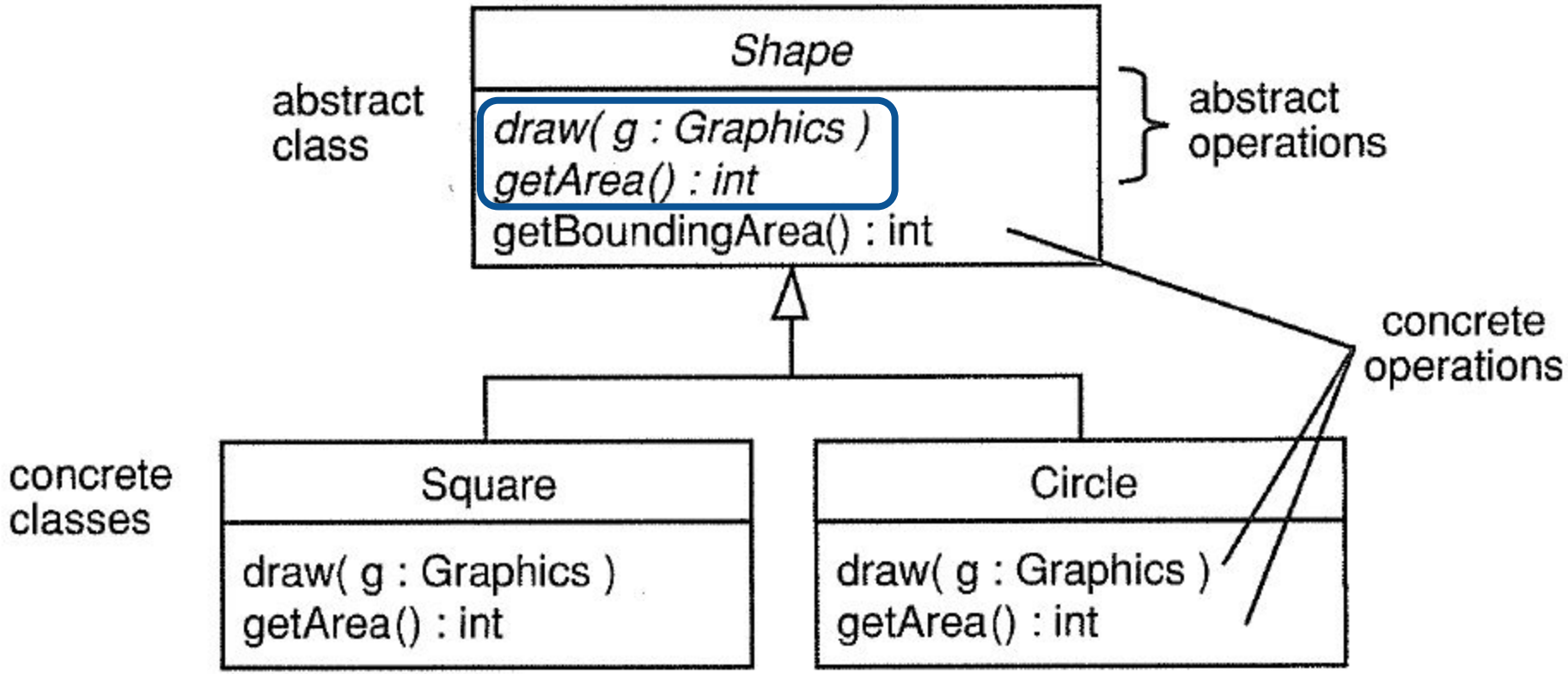
Warning: programming languages such as C++ and Java do not include the return type in the operation signature. If you redefine the return type to a non-conforming type, you get a compiler or interpreter error.

Actually, do we really want to create Shape objects? Not really. How would we implement some of those methods?

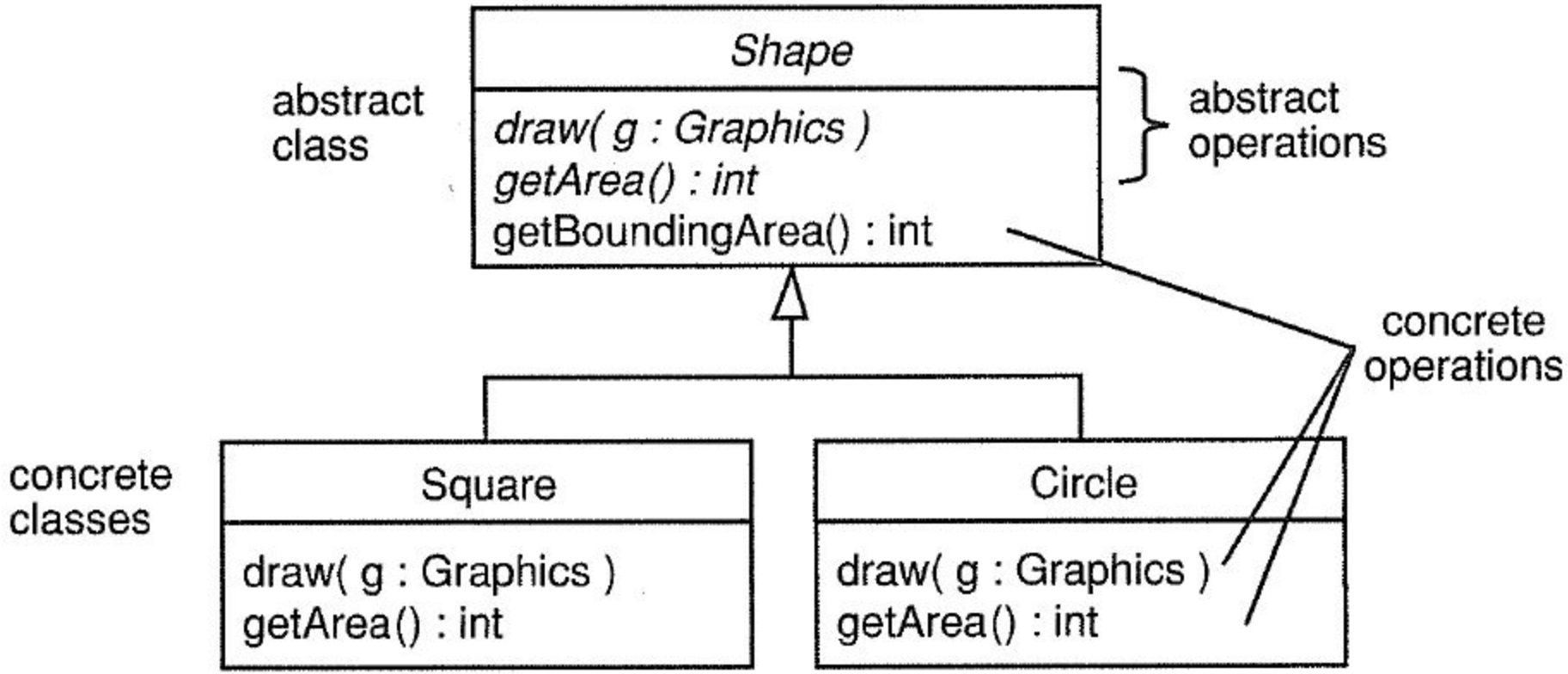


When a superclass was created to factorize common properties and behaviors of a set of classes, but contains one or more operations that we know we will need to redefine anyway, we should use an abstract class!

Abstract operations do not have an implementation: it is deferred to concrete operations in the sub-classes

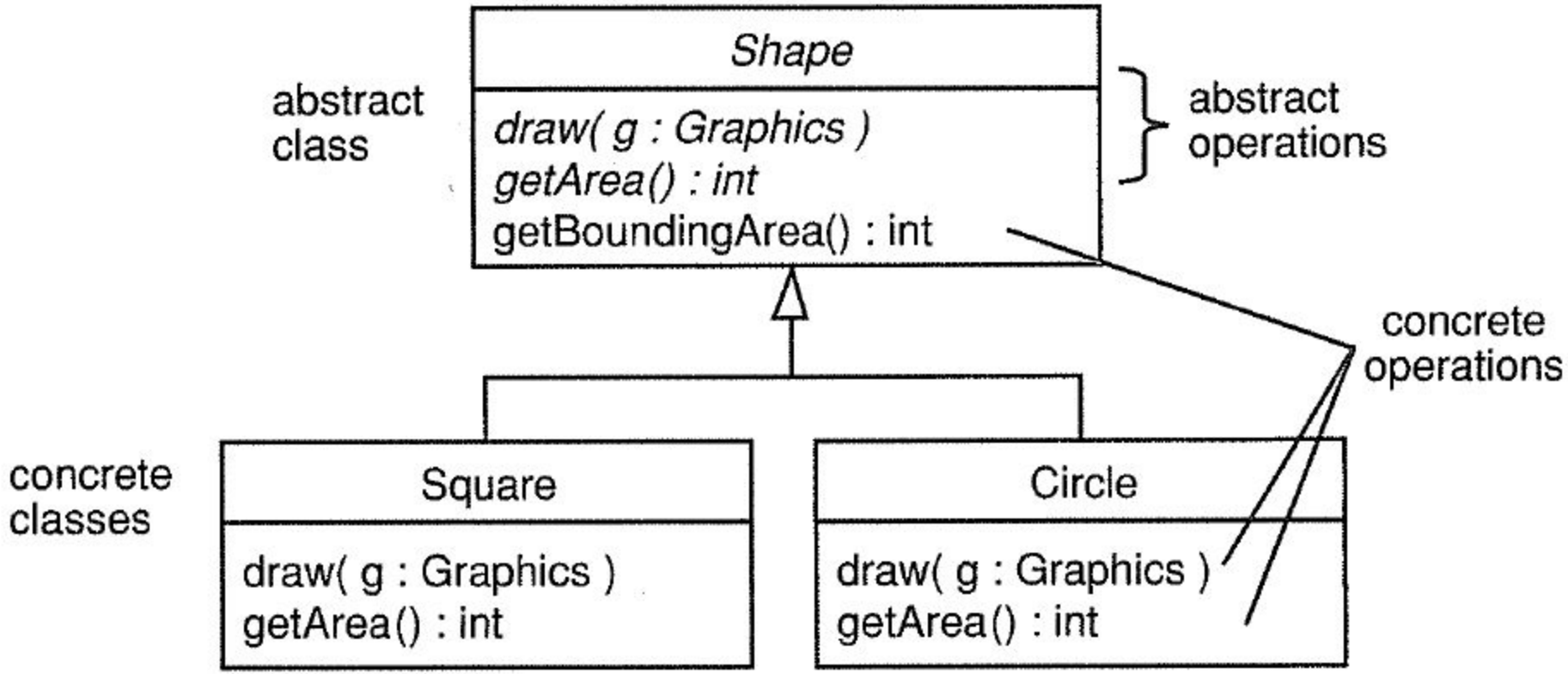


Abstract classes include one or more abstract operations and cannot be instantiated

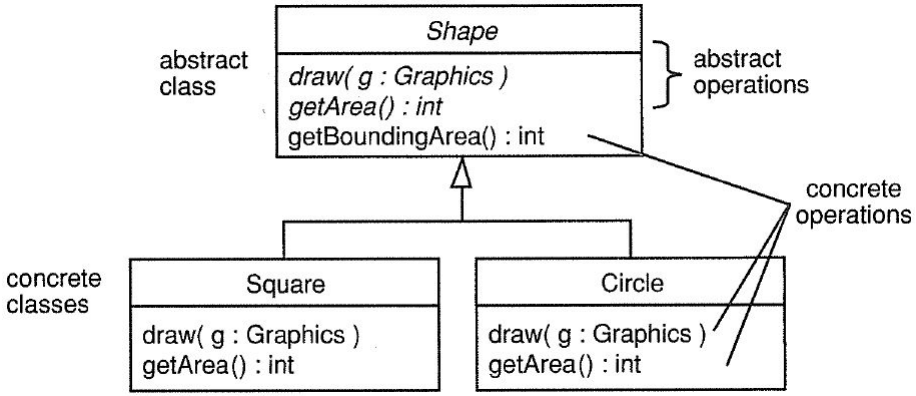


An abstract class is denoted by one of the following:
their name in ***italic***, the stereotype **<<abstract>>**, or the tagged value **{abstract}**

getBoundingArea() is concrete because it is always computed the same way (width * height)

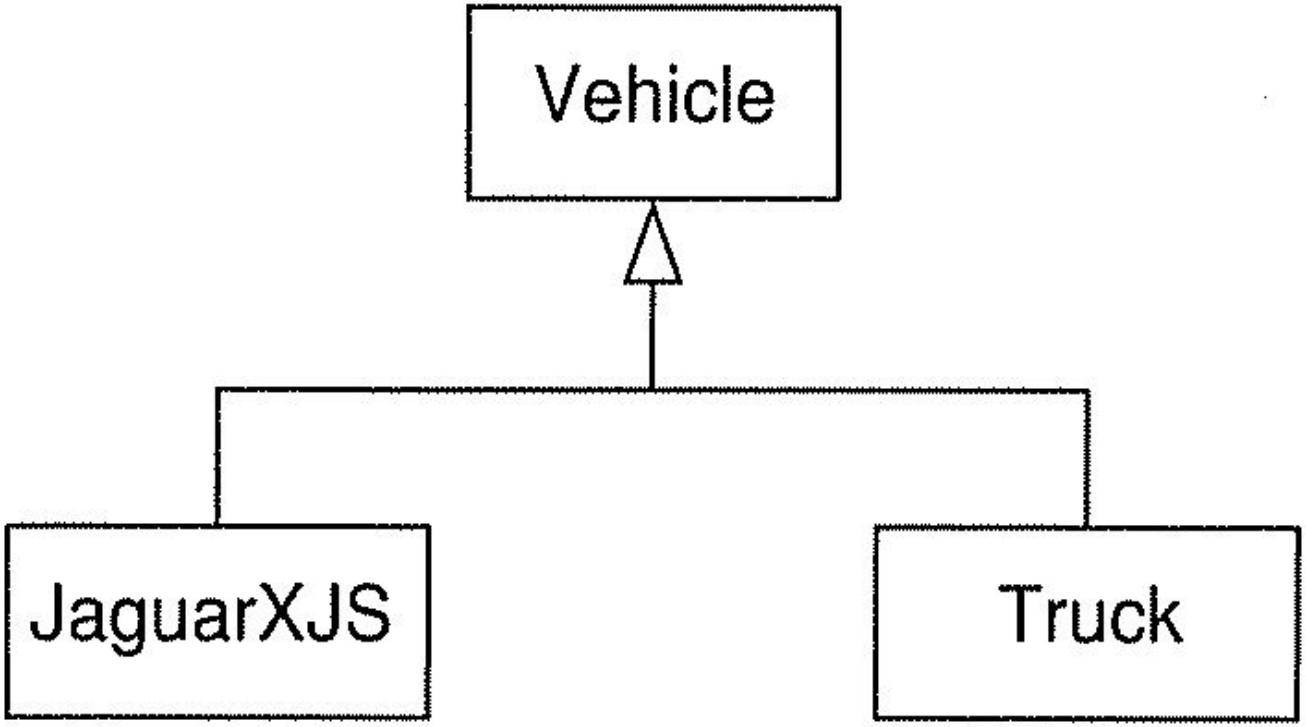


Advantages of using abstract classes and operations



- The set of abstract operations in the abstract class must be implemented in the concrete sub-classes
- You can write code to manipulate Shapes and then substitute Square, or Circle, as appropriate
- Code written for the abstract class should work correctly for all concrete classes - **remember the substitution principle**!

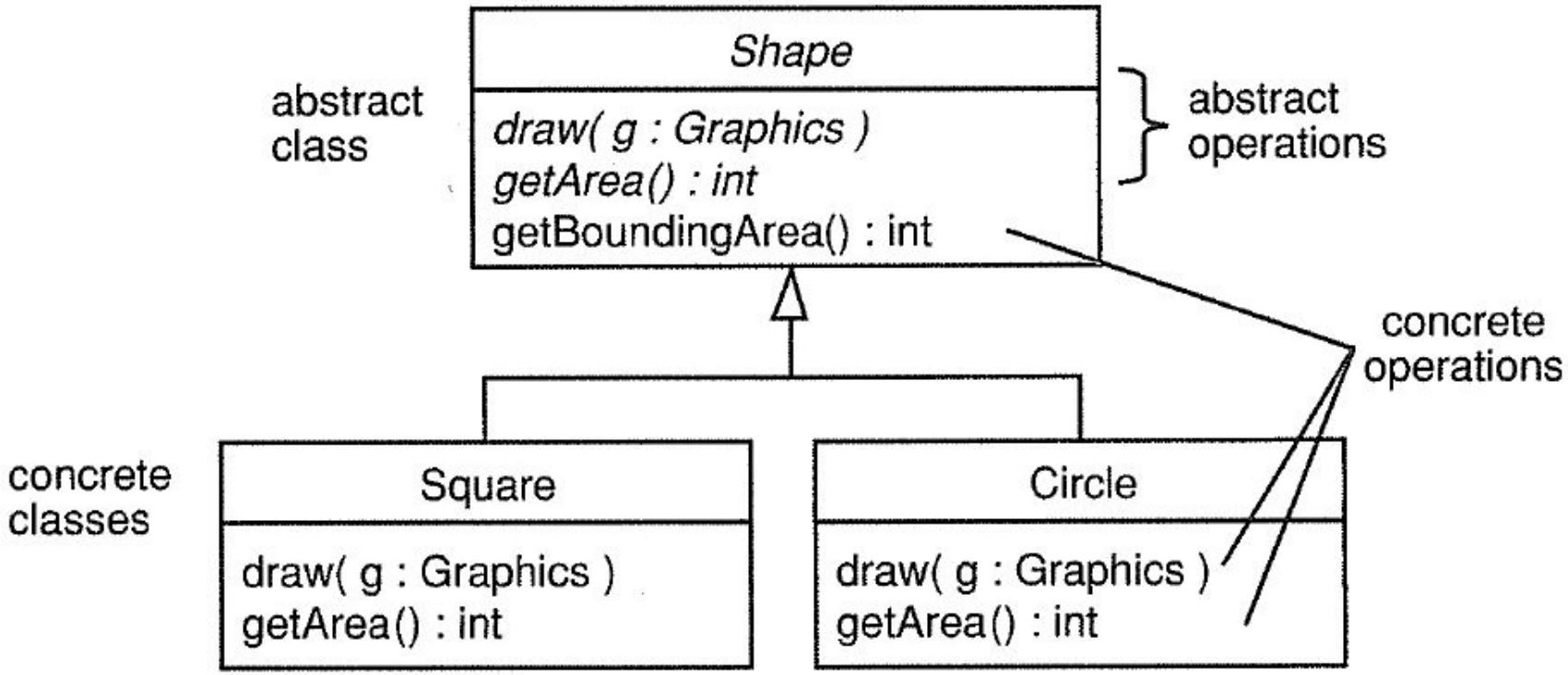
Keep classes at the same hierarchical level at the same level of abstraction



In this case, Jaguar XJS is a particular model from a given car brand, while Truck is a vehicle defined at a different level of abstraction. This is bad practice.

Polymorphism

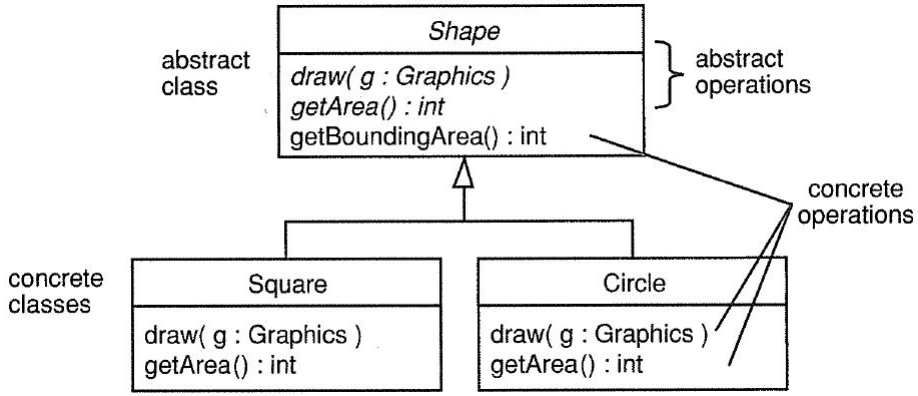
A polymorphic operation is an operation with several implementations



Objects from different sub-classes will receive exactly the same message, but react differently to it, according to their own implementation of the called operation

A polymorphic operation is an operation with several implementations

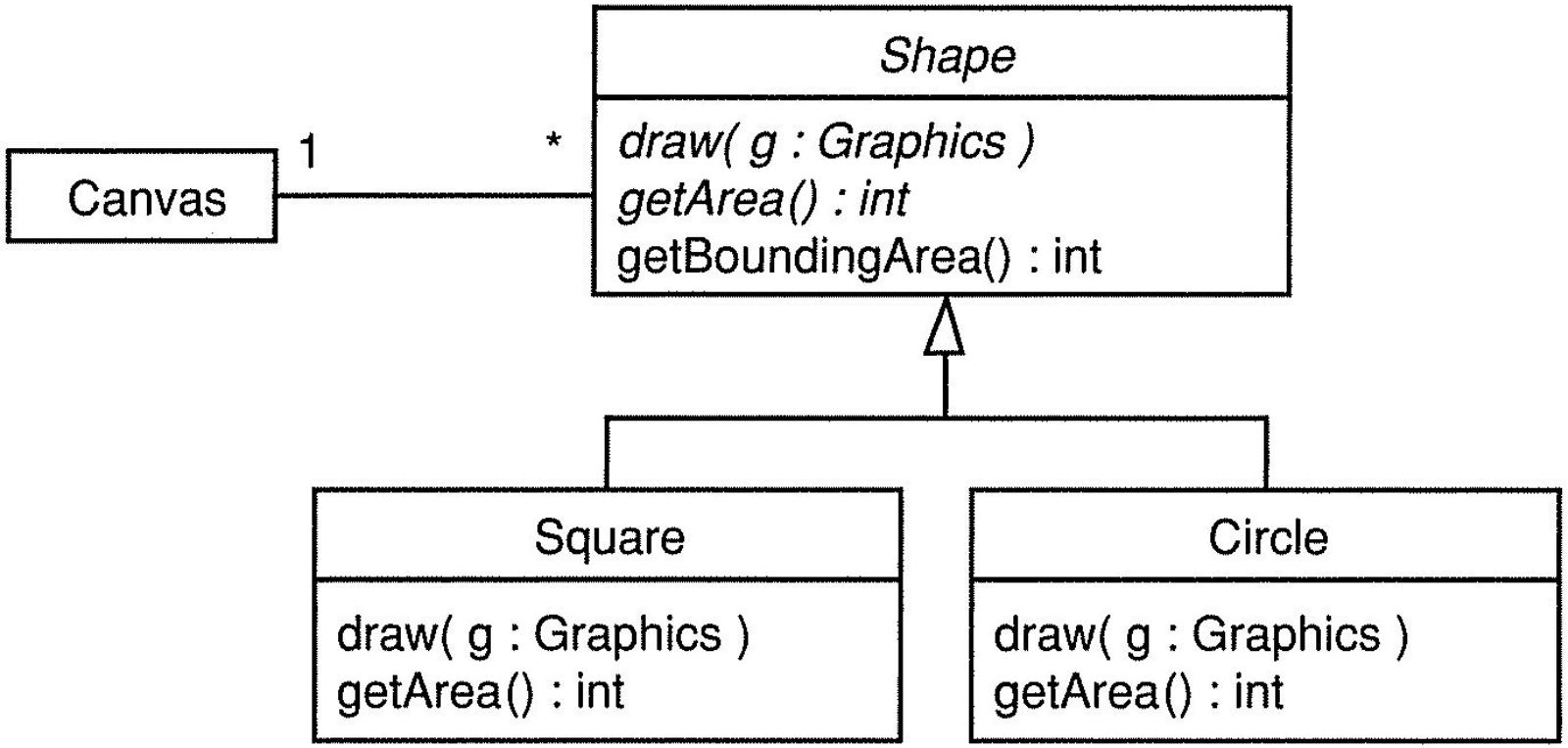
- Square and Circle specialize Shape
- They must provide implementations for the abstract operations draw() and getArea()



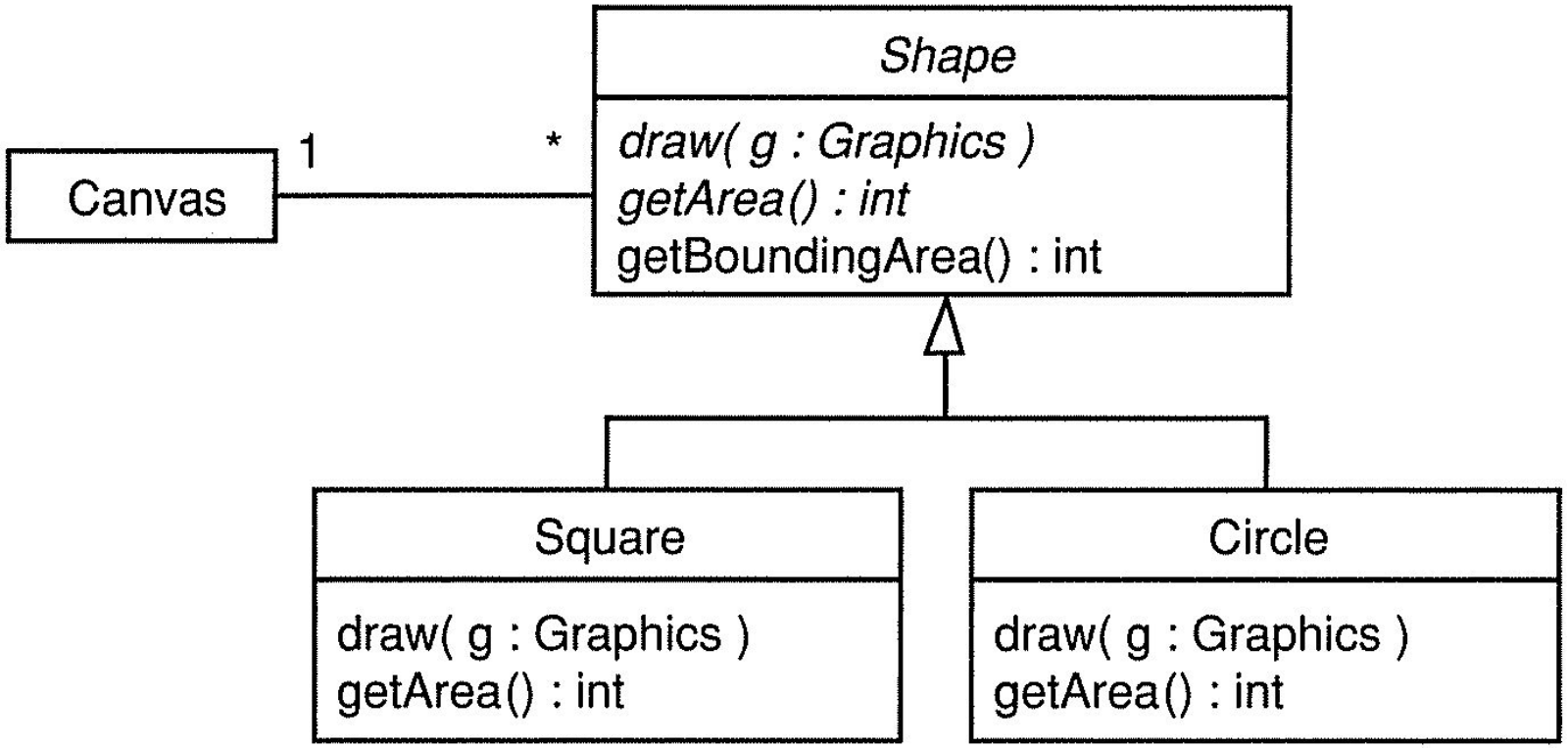
Concrete classes must provide concrete implementations to ALL the abstract operations inherited from their abstract super-classes
Objects of different sub-classes will implement the inherited abstract operation differently - hence the term polymorphism

Objects from different sub-classes will receive exactly the same message, but react differently to it, according to their own implementation of the called operation

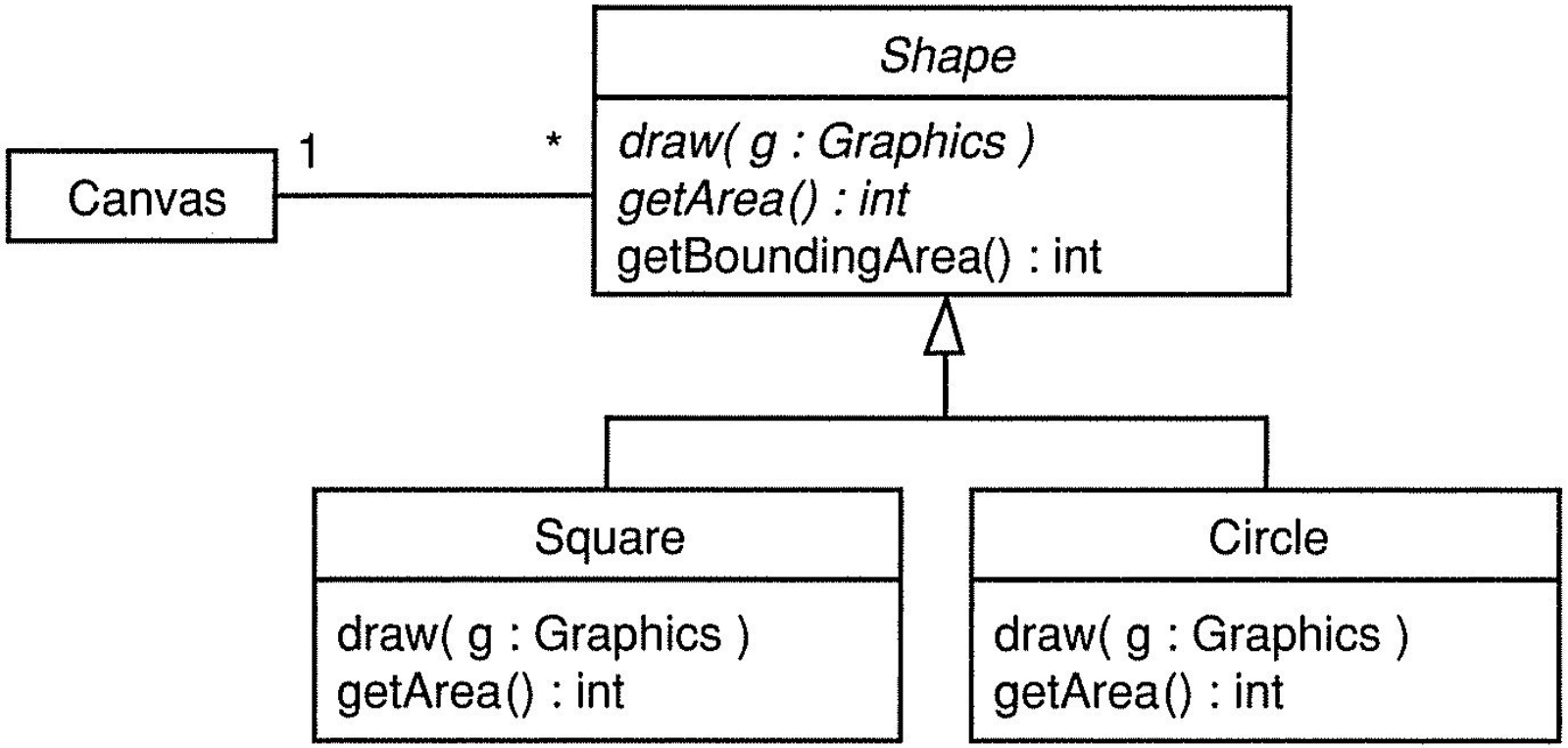
Each element of the Shapes collection in Canvas will be of a particular sub-class



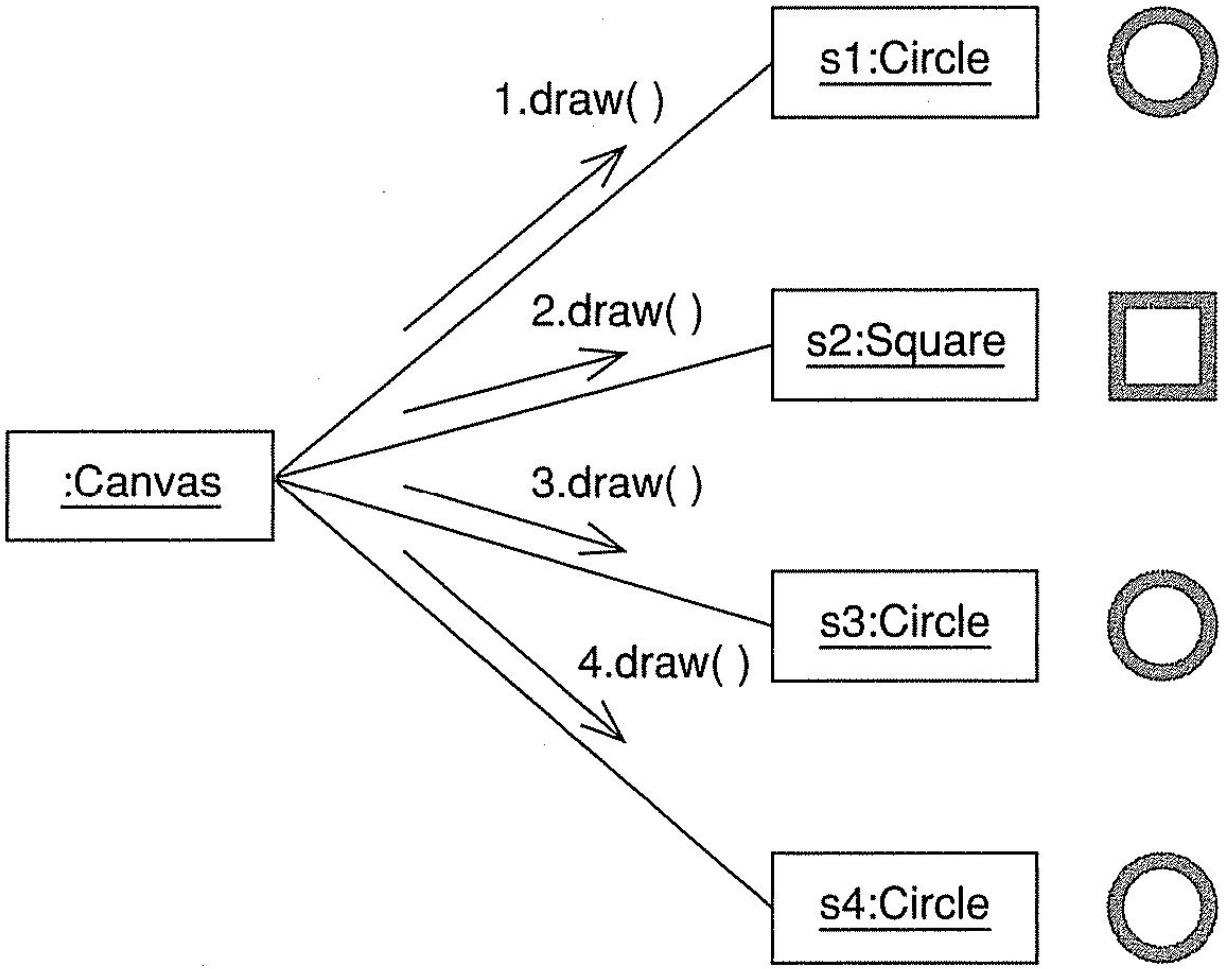
When calling a draw() or a getArea() operation of Shape from Canvas, we do not know which kind of Shape will answer



The actual class to answer the call will be determined via a mechanism named dynamic binding



Each collection member will use its own implementation of the draw() operation



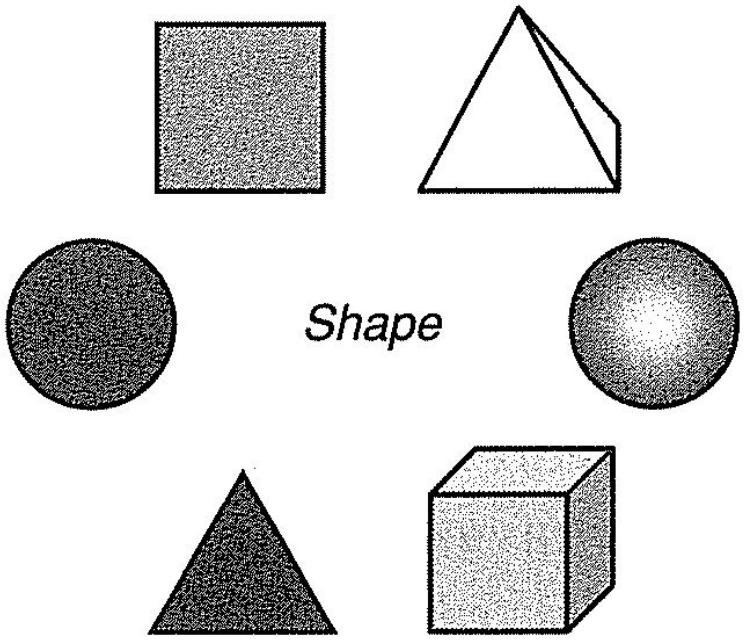
Can you override concrete operations?

YES!

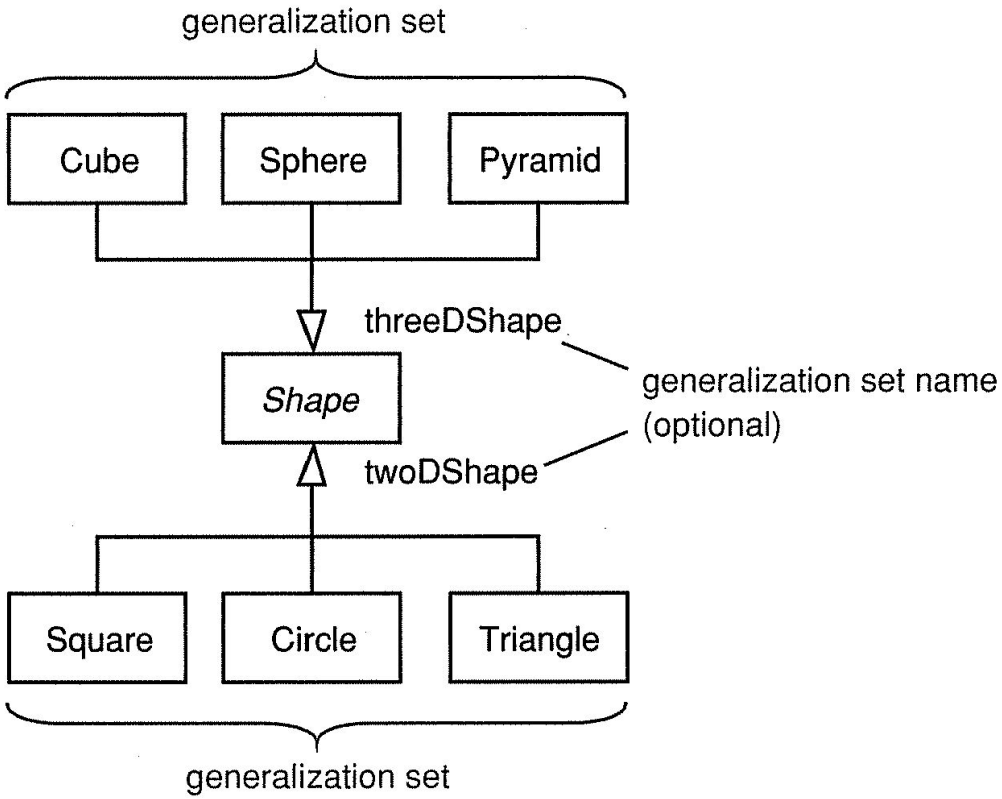
But please don’t. It is really bad style!

Generalization sets partition subclasses according to a specific rule

How would you partition these shapes?



You can use generalization sets to segregate the two sets (2d and 3d shapes)



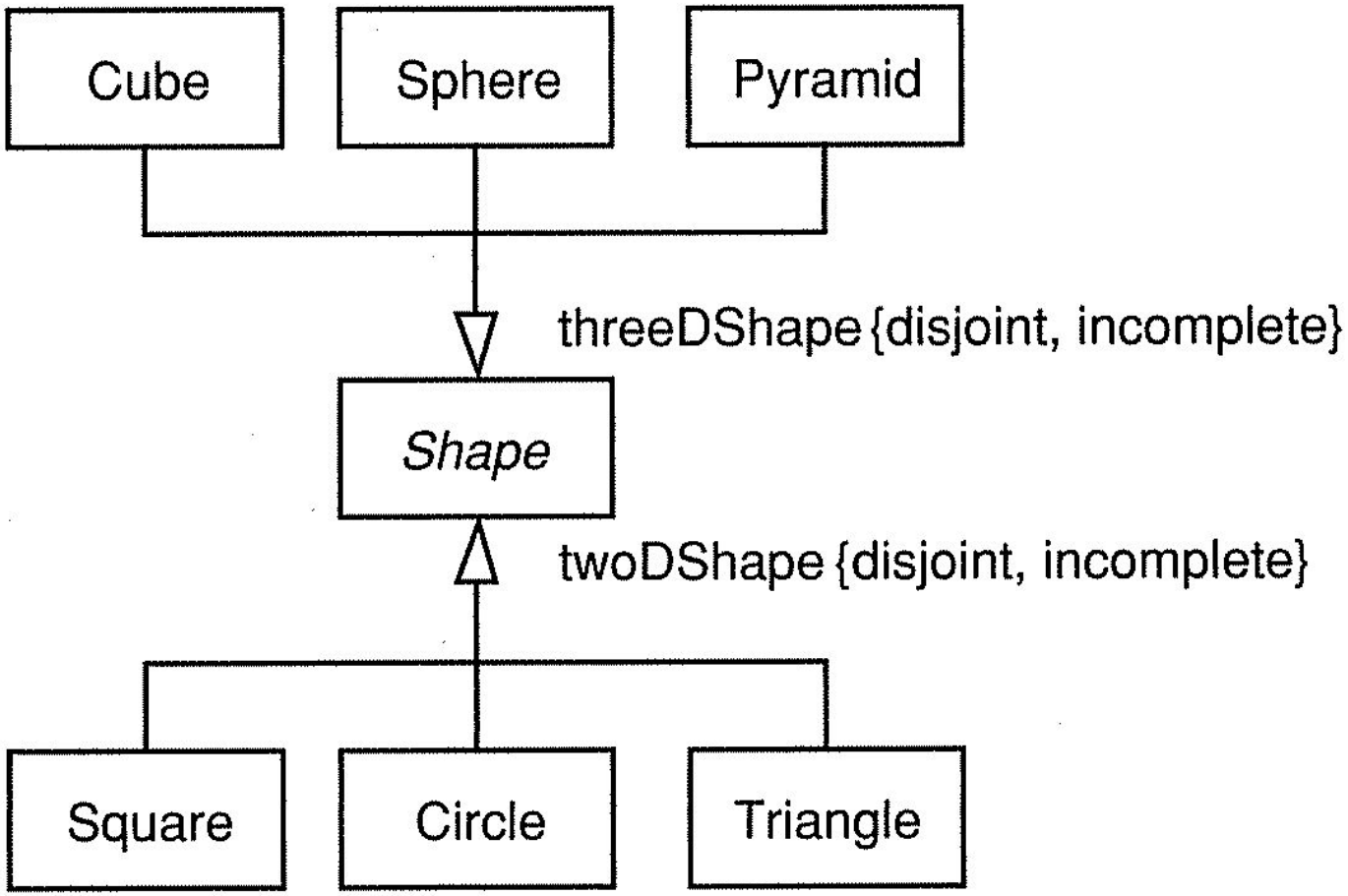
Generalization sets can be constrained

- {complete}
  - subclasses cover all possibilities
- {incomplete}
  - there may be other subclasses not modelled yet
- {disjoint}
  - an object can be an instance of one and only one of the members of the generalization set (this is the most common)
- {overlapping}
  - an object can be an instance of more than one of the members of the generalization set (requires multiple inheritance, or multiple classification)

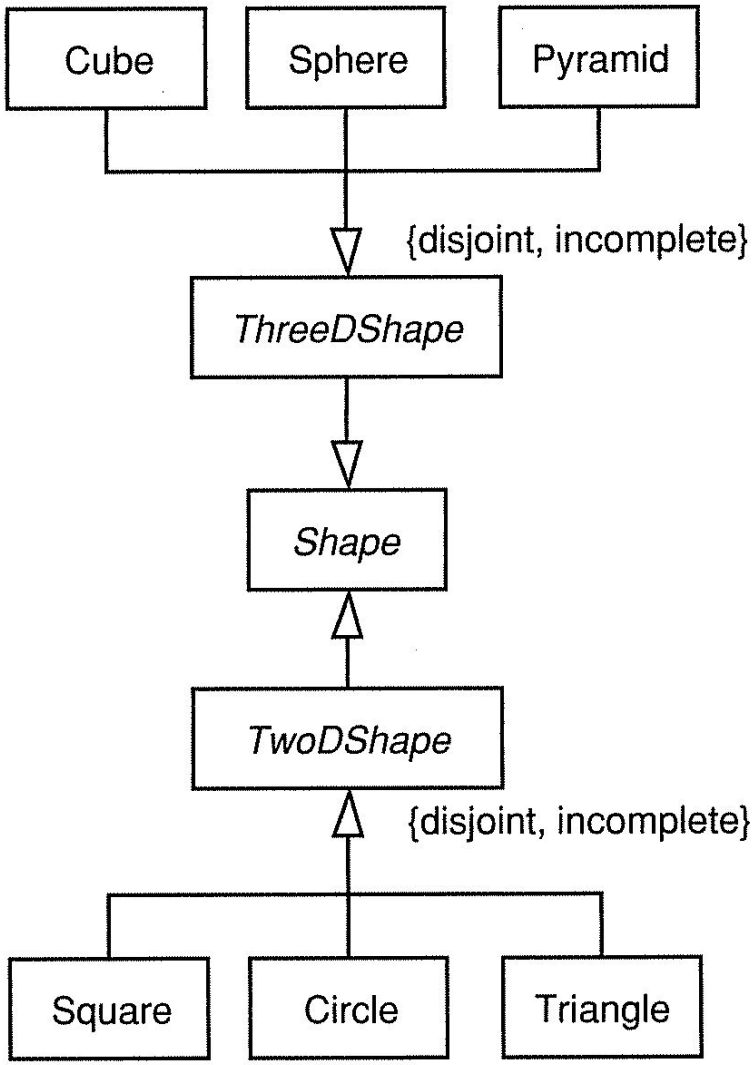
Generalization sets can be combined

Constraint	The set is complete	Members of the set may have instances in common
{incomplete, disjoint} – the default	N	N
{complete, disjoint}	Y	N
{incomplete, overlapping}	N	Y
{complete, overlapping}	Y	Y

Applying generalization sets to shapes



Resolve generalization sets into new classes in the inheritance hierarchy

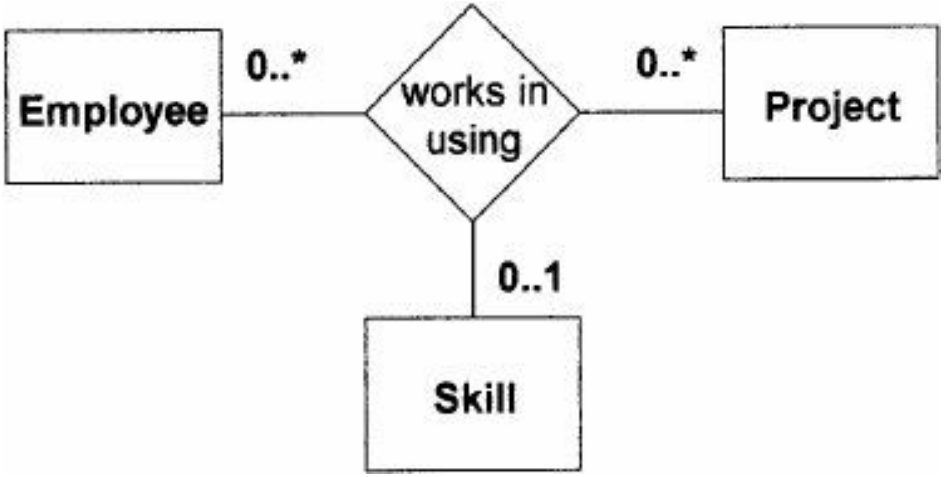


Benefits and challenges of inheritance

- Benefits of Inheritance
  - Abstraction mechanism/ classify entities
  - Mechanism for reuse
- Challenges
  - We can only understand classes if we know their superclasses
  - Sometimes the inheritance graph is not compatible with efficiency

N-ary associations

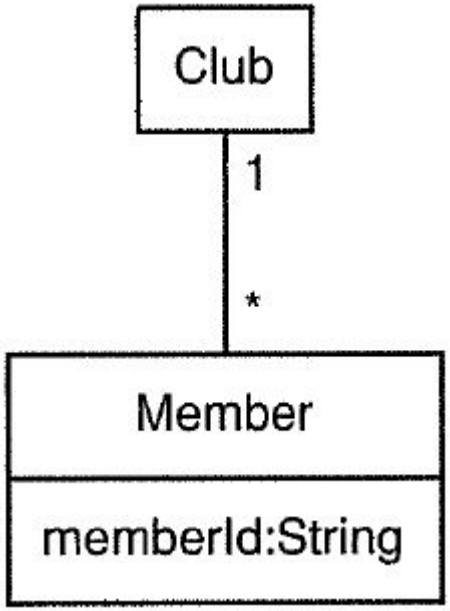
N-ary association is represented by a diamond connected to each of the 3 (or more) associated classifiers



Advanced association types

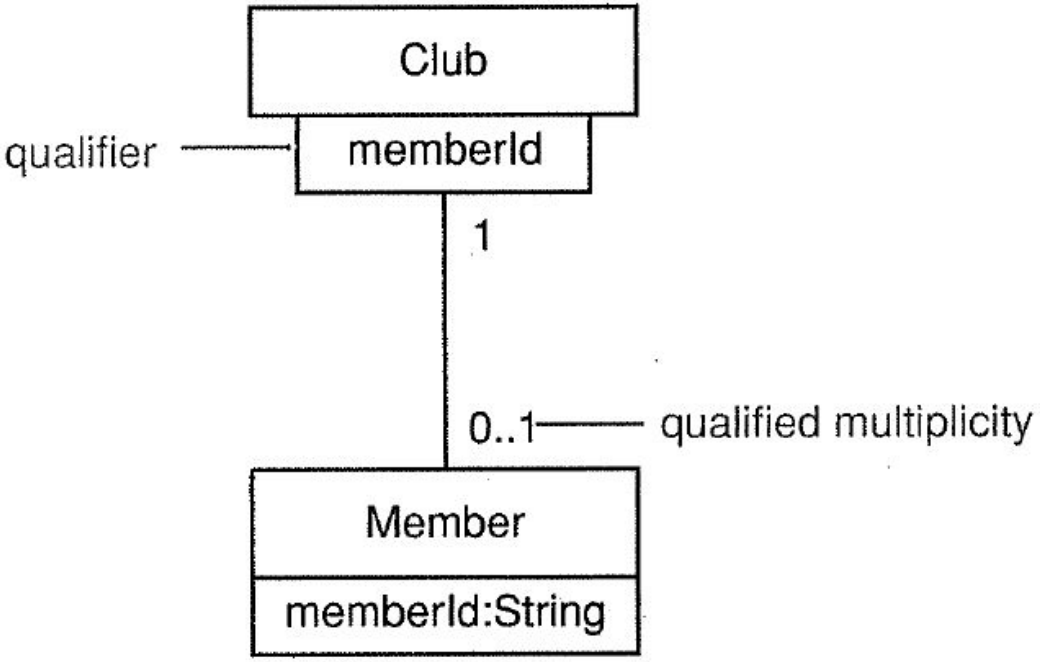
Qualified associations and why do we need them

How can you navigate from a Club to a specific Member?



A qualified association selects a single member from the target set

- The qualifier is expressed as a unique property of the type in the target set, or at least an expression that leads to a unique element of the target set
- The qualifier belongs to the association, rather than to the Club class
- Now, there is at most one member of the Member class that can be qualified for selection
- Rather than a one to many association, we now have a one to at most one relationship

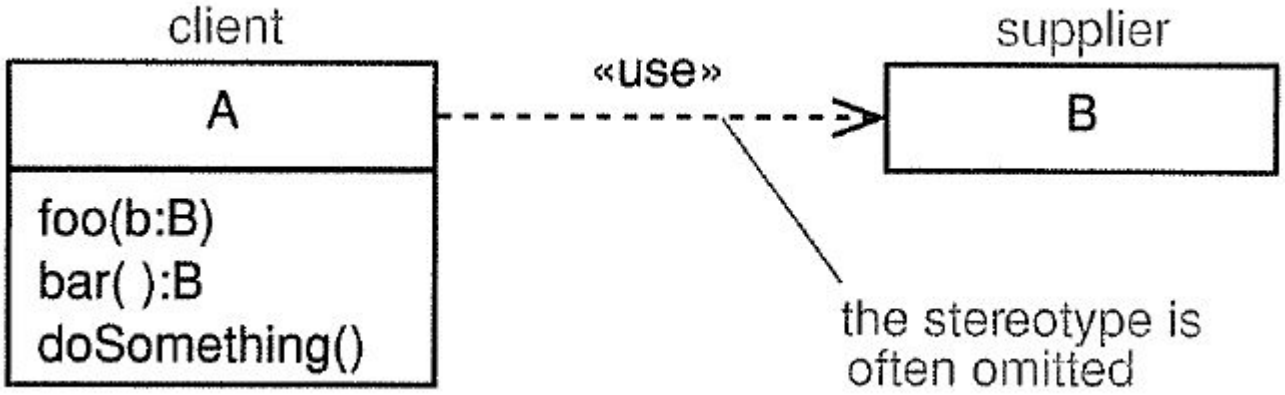


A dependency establishes that an object of the client class depends, in some way, on the supplier

- Sometimes, the dependency is there but it is not really an association or generalization (e.g. an object of a given type is passed to an object of another class as an operation parameter)
- We consider three kinds of dependencies
  - **Usage -** the client uses services from the supplier to implement some of the client's behavior
  - **Abstraction -** the supplier is more abstract than the client (e.g. an analysis class is more abstract than its corresponding design class)
  - **Permission -** The supplier offers some kind of access to its contents to a client

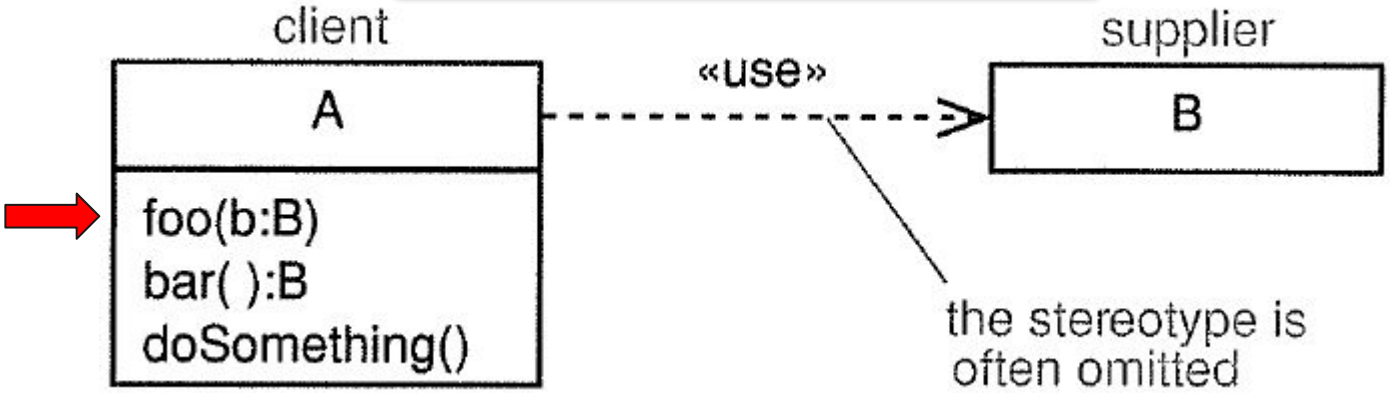
<<use>> dependency

- An operation of class A needs a parameter of class B
- An operation of class A returns an object of class B
- An operation of class A does something that uses an object of class B
  - For example, doSomething() may create a local variable of the class B type



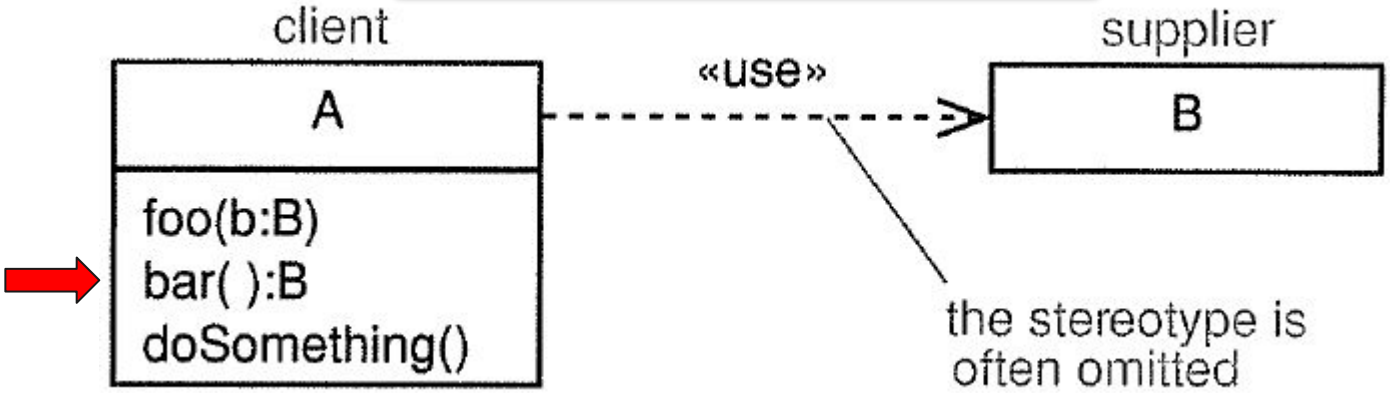
<<use>> dependency: An operation of class A needs a parameter of class B

```
class A {
   ...
   void foo(B b) {
      ...
   }
}
```



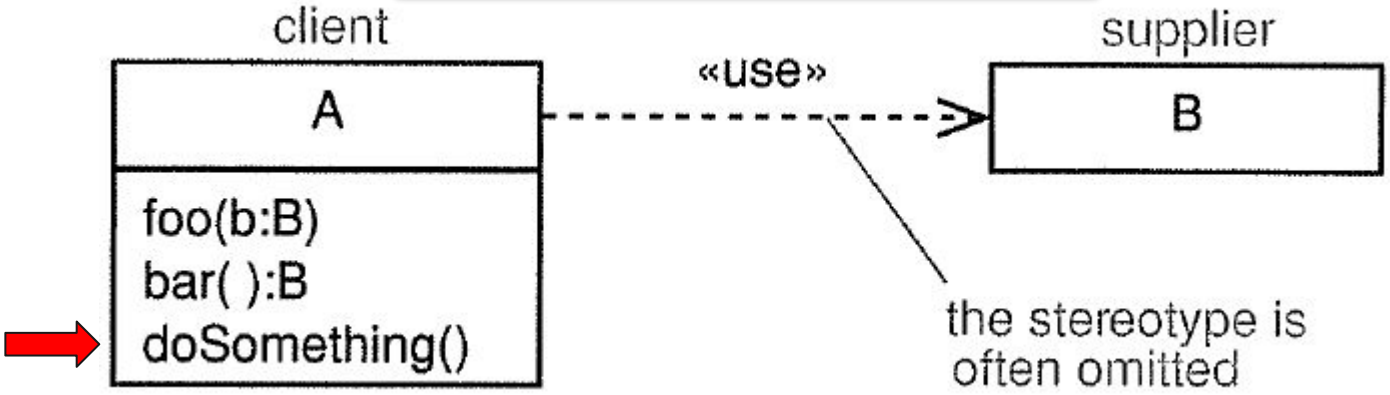
<<use>> dependency: An operation of class A returns an object of class B

```
class A {
   ...
   B bar() {
      ...
   }
}
```



<<use>> dependency: An operation of class A does something that uses an object of class B

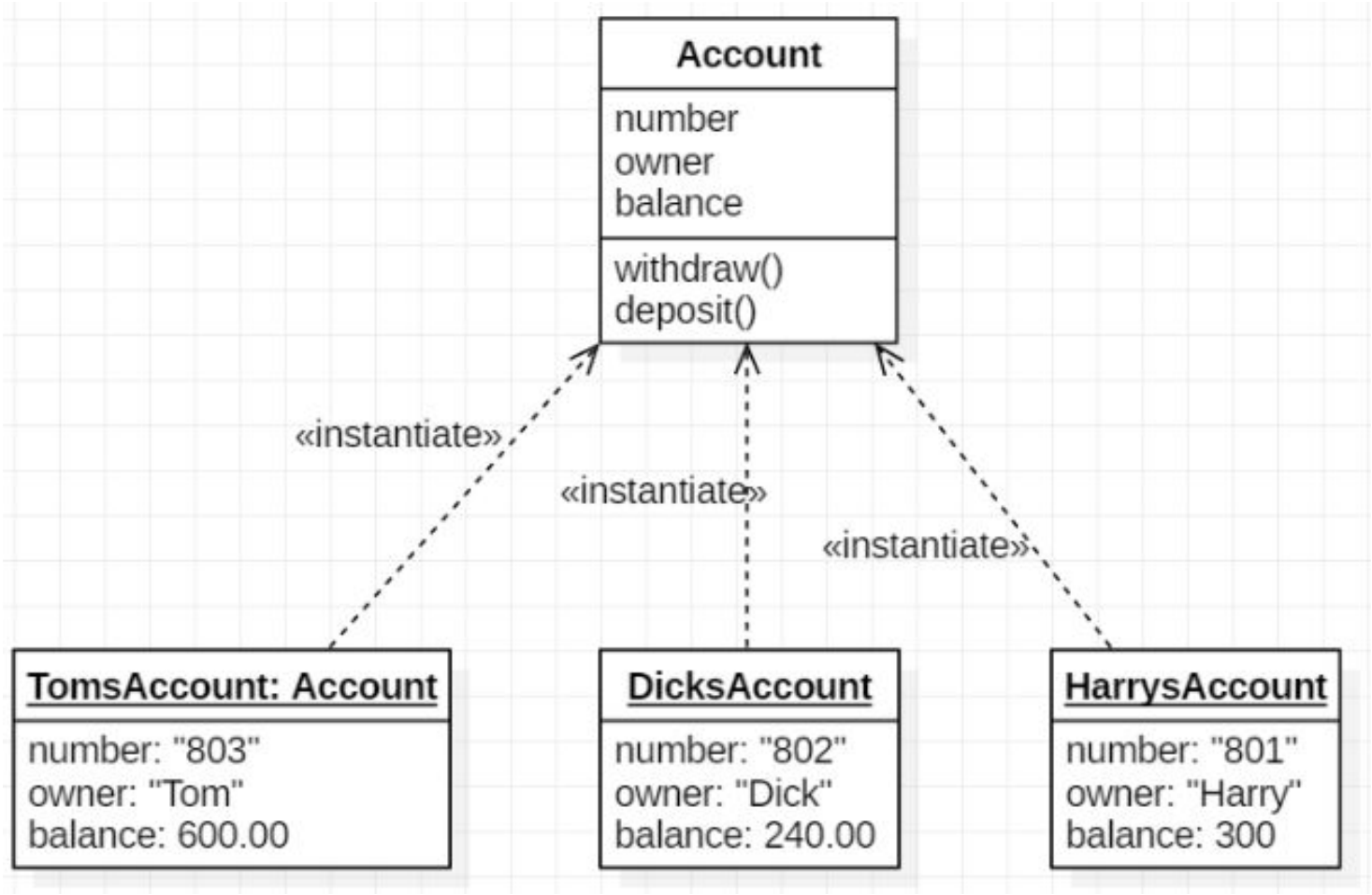
```
class A {
   ...
   void doSomething() {
      B myB = new B();
      // use myB in some way
      ...
   }
}
```



Other dependencies

- `<<call>>`
  - Dependency between operations - the client operation invokes the supplier operation
- `<<parameter>>`
  - The supplier is used as a parameter of the client operation
- `<<send>>`
  - The client is an operation that sends the supplier (which must be a signal) to some unspecified target
- `<<instantiate>>`
  - The client is an instance of the supplier

<<instantiate>> dependency: The client is an instance of the supplier

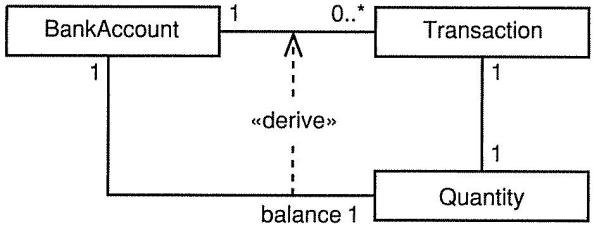
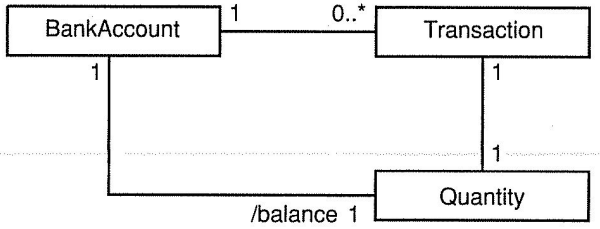
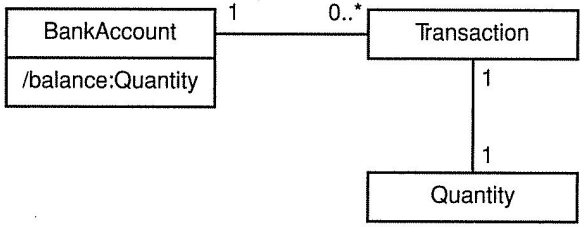


Yet some more advanced dependencies

Abstraction dependencies

- `<<trace>>`
  - The client and the supplier represent the same concept but are in different models
- `<<substitute>>`
  - The client may be substituted by the supplier at runtime
- `<<refine>>`
  - The client is a different (refined) version of the supplier (e.g. the client is an optimized version of the supplier
- `<<derive>>`
  - A thing can be derived from some other thing

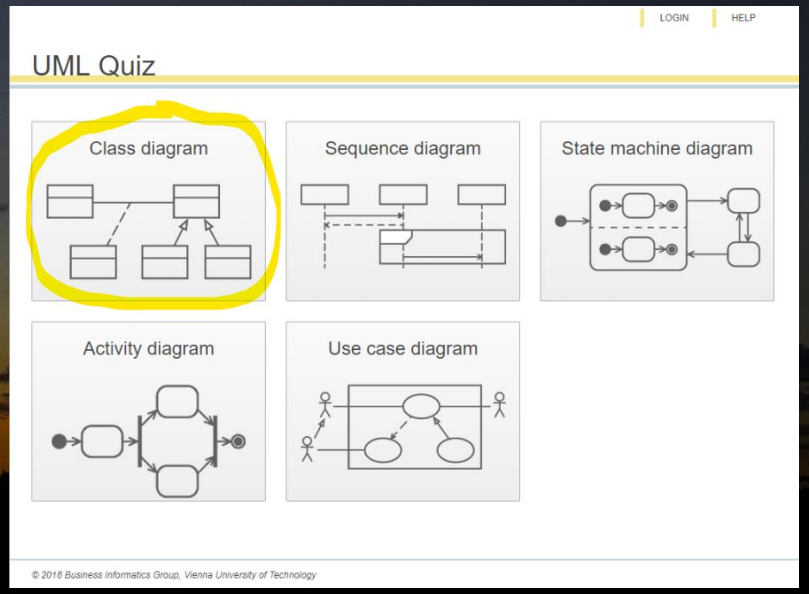
<<derive>> can be presented in 3 different ways: the first alternative is the most common

Model	Description
	The BankAccount class has a derived association to Quantity where Quantity plays the role of the balance of the BankAccount This model emphasizes that the balance is derived from the BankAccount's collection of Transactions
	In this case a slash is used on the role name to indicate that the relationship between BankAccount and Quantity is derived This is less explicit as it does not show what the balance is derived from
	Here the balance is shown as a derived attribute – this is indicated by the slash that prefixes the attribute name This is the most concise expression of the dependency

Permission dependencies

- `<<access>>`
  - This is a dependency between packages where the client package can access all of the public contents of the supplier package, while the packages namespaces remain separate - we will revisit this when discussing package diagrams
- `<<import>>`
  - This is a dependency between packages where the client package can access all of the public contents of the supplier package, while the namespaces of the packages are merged - we will revisit this when discussing package diagrams
- `<<permit>>`
  - This is a controlled violation of encapsulation where the client may access the private members of the supplier - this is not only often NOT supported by tools and should be avoided if possible - programming languages like Java and C# do not support this mechanism, unlike older languages, such as C++

Did you really understand Class Diagrams? Test yourself at: http://elearning.uml.ac.at/



Bibliography

Jim Arlow and Ila Neustadt, “UML 2 and the Unified Process”, Second Edition, Addison-Wesley 2006

- Chapters 7, 8, 9, 10

Package diagrams show the structure of the designed system, at the level of **Packages**

The Package is the UML mechanism to group things

- Provides an encapsulated namespace within which all names must be unique
- Groups semantically related elements
- Defines a semantic boundary in the model
- Provides units for parallel working and configuration management
- Packages are a **logical grouping mechanism**
  - Physical grouping is achieved with components

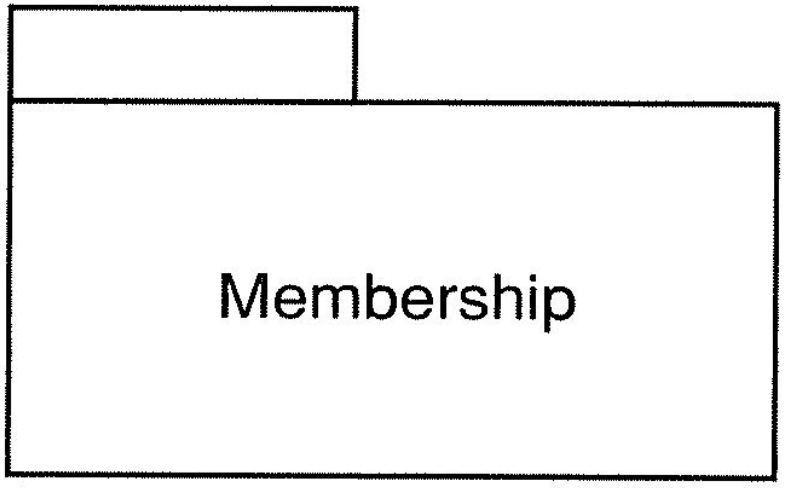
Every model element is owned by one package

- The ownership hierarchy forms a tree
- The top level package is stereotyped as <<toplevel>>
- By default, all model elements are owned by the top level package
- The package hierarchy forms a namespace hierarchy where the top level package is the namespace root

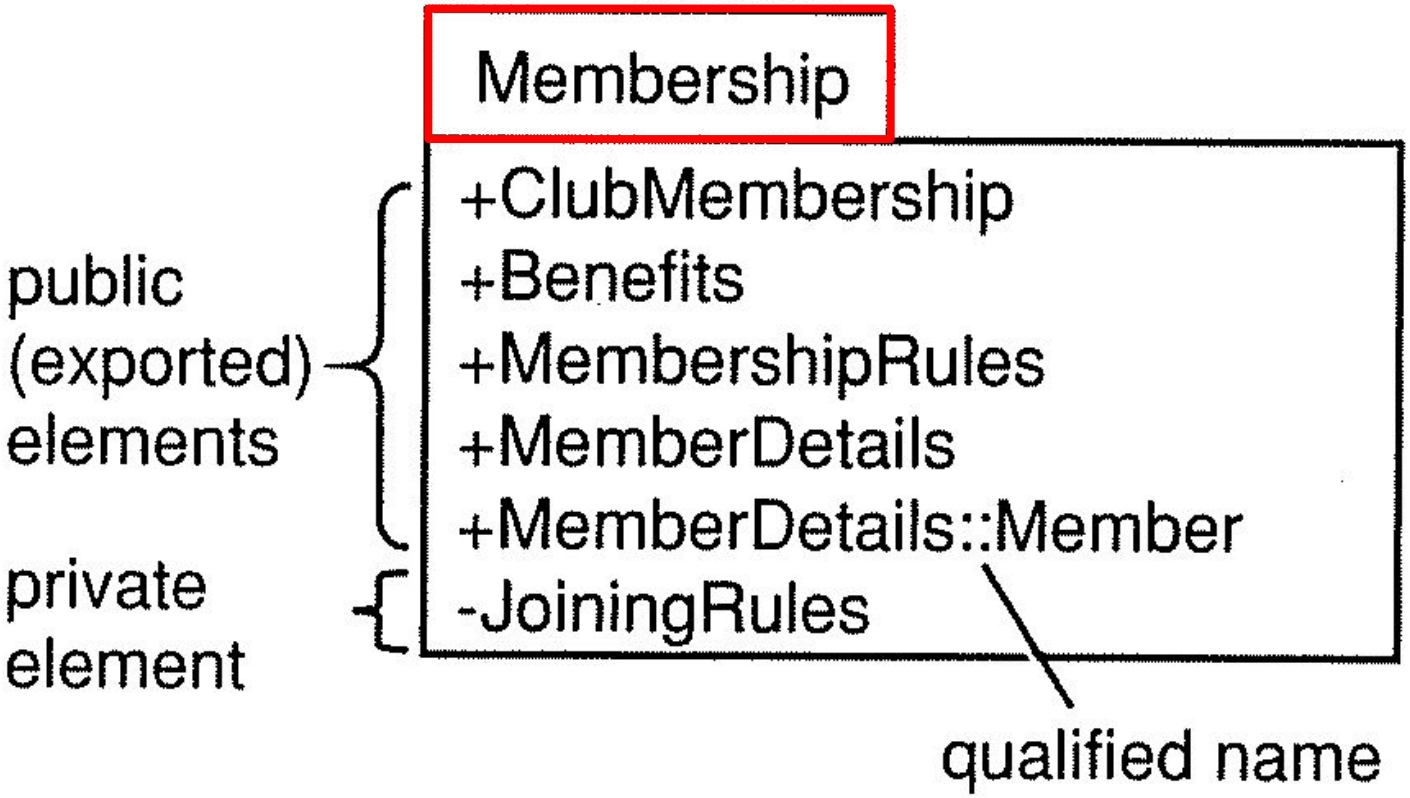
Analysis packages should contain

- Use cases
- Analysis classes
- Use cases realizations

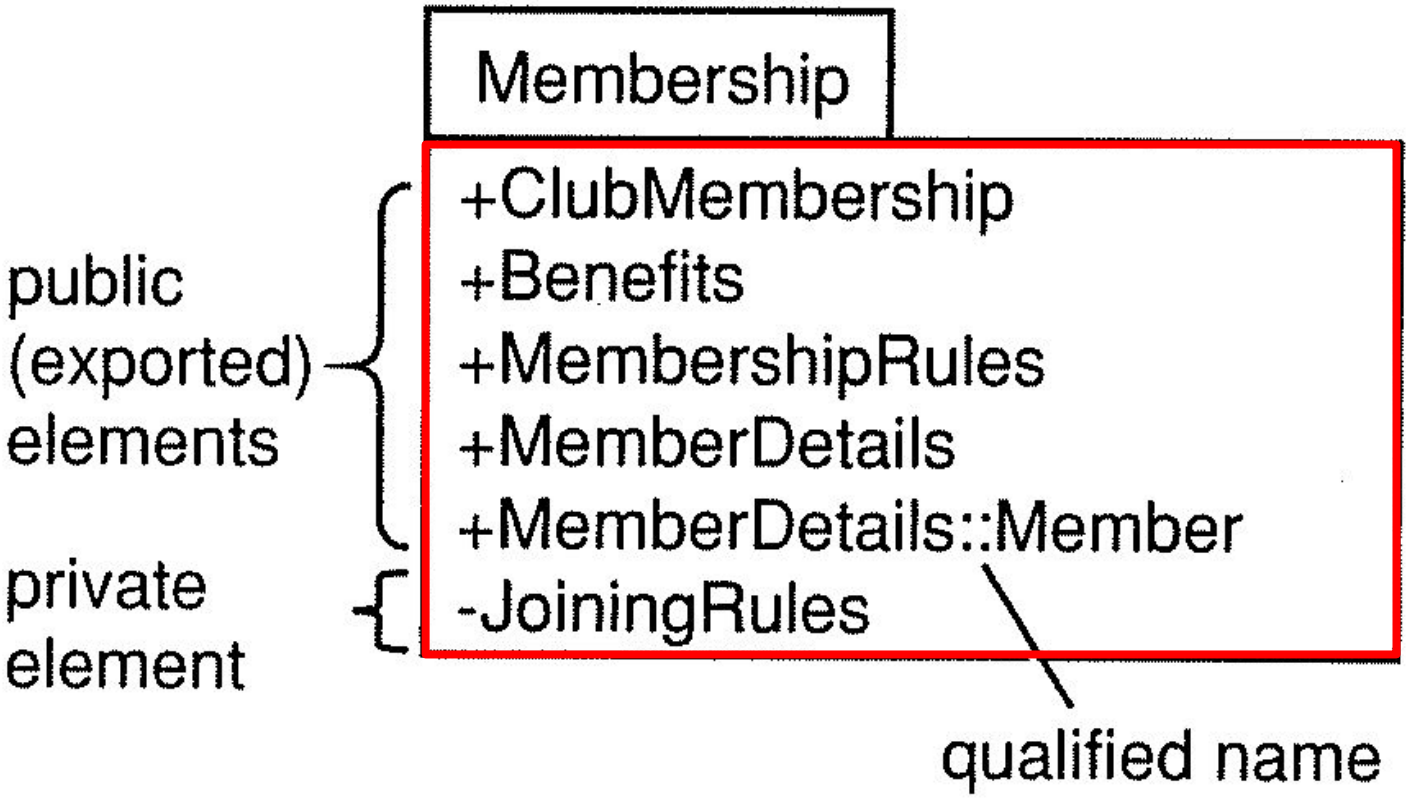
A UML package is represented as a folder



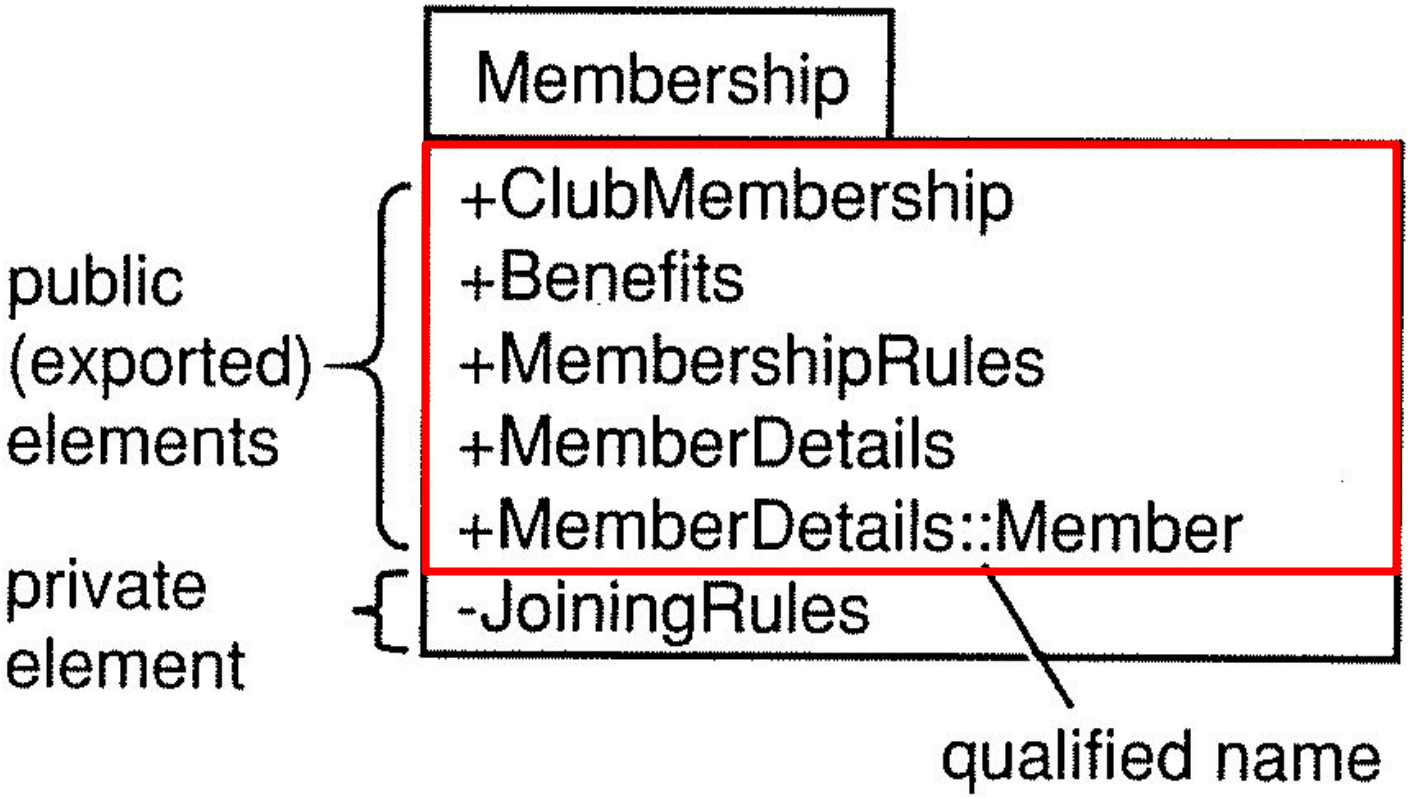
The package name can be represented in the tab, if package contents are shown



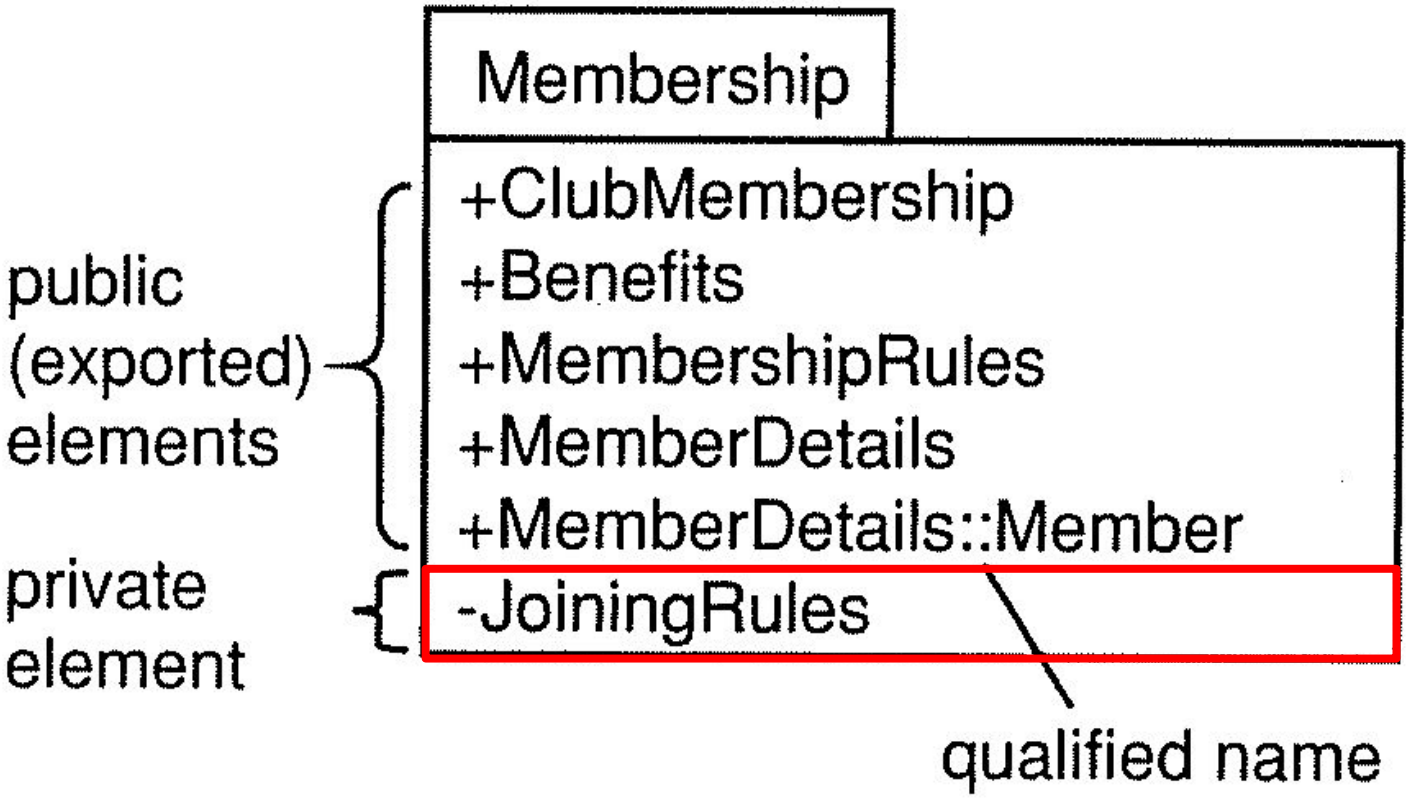
The package name can be represented in the tab, if package contents are shown



Elements with public visibility (+) are visible to elements outside the package



Elements with private visibility (-) are visible to elements outside the package



Visibility determines whether a package element is visible outside the package, or not

- Use visibility to control the amount of coupling between packages
- Keep the package interface small and simple
  - Minimize the package elements with public visibility
  - Maximize the package elements with private visibility

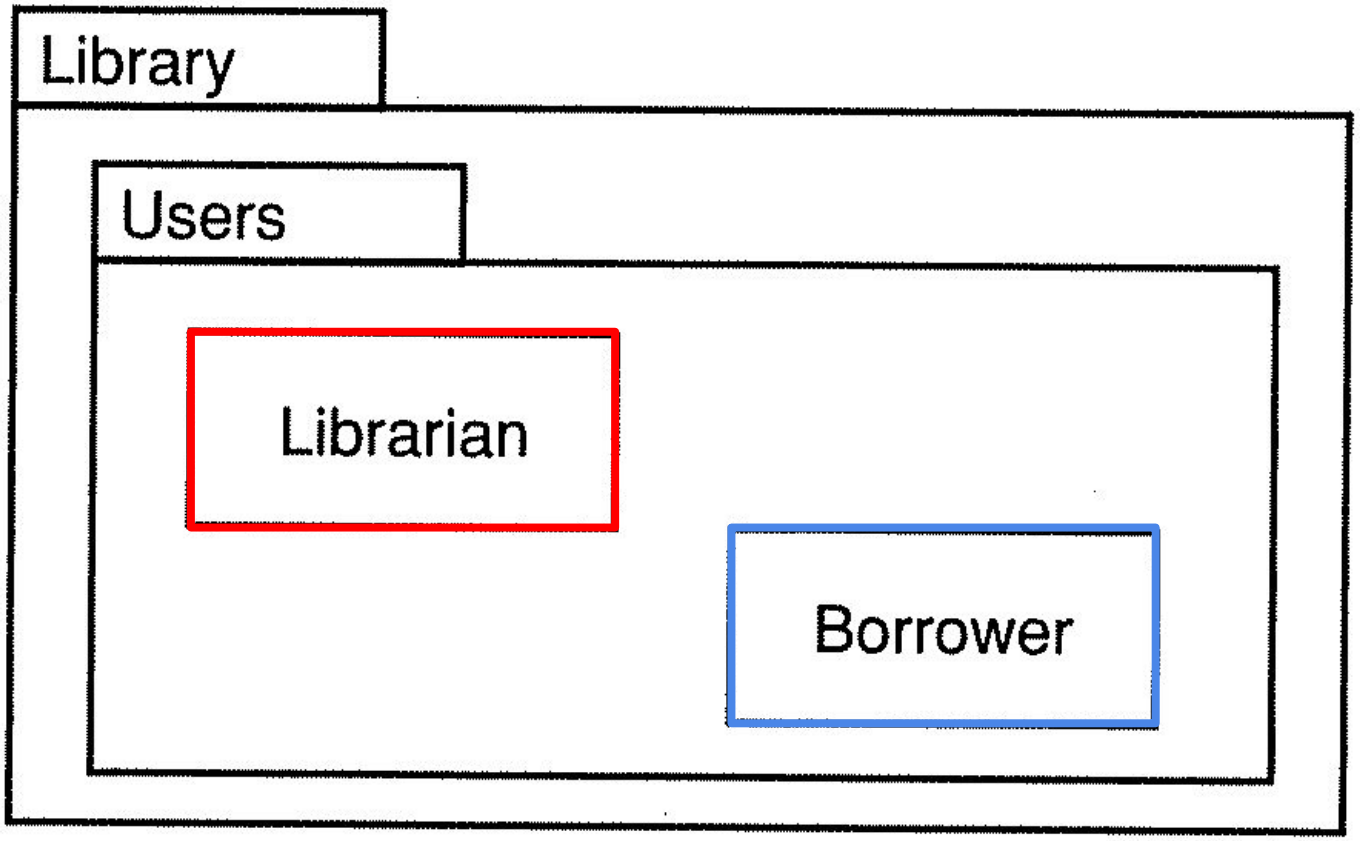
Packages and namespaces

- A package defines an **encapsulated namespace**
- Within a namespace, all element names are unique
- When referring to an element of another namespace, we need to use its qualified name
- Qualified names are similar to pathnames in directory structures

Example: Qualified names of Librarian and Borrower

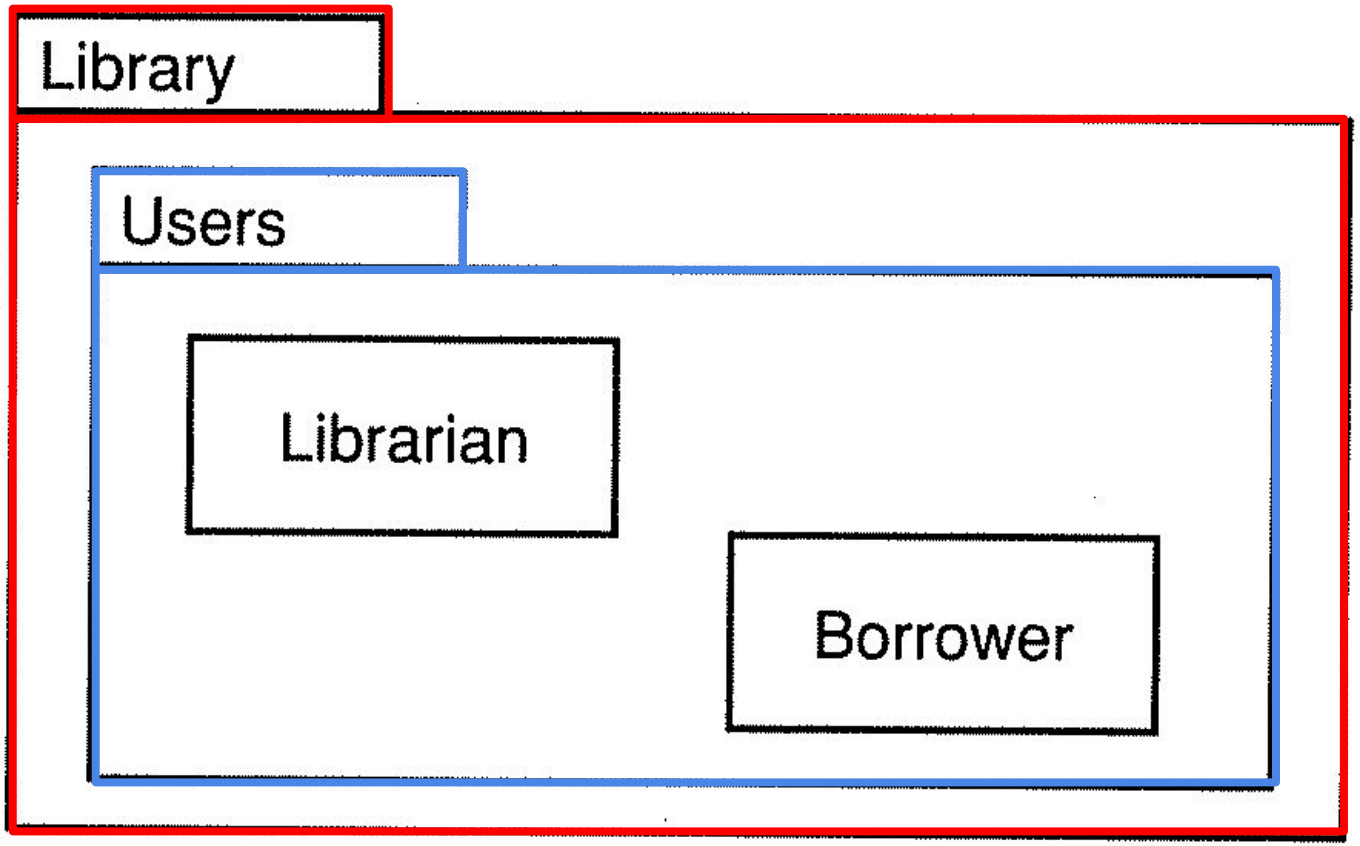
Library::Users::Librarian

Library::Users::Borrower

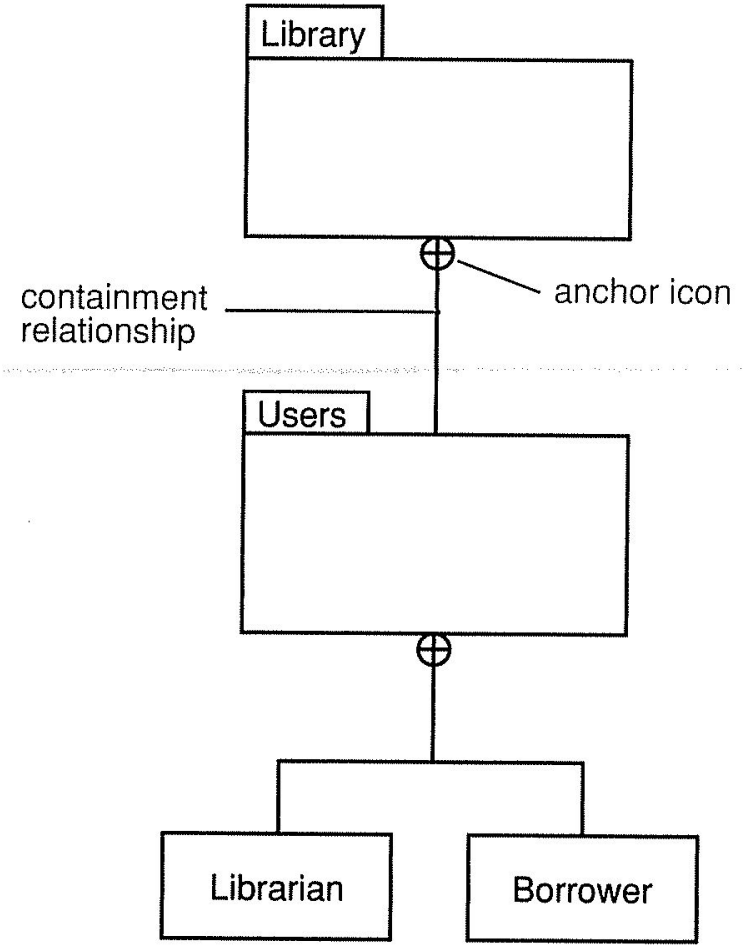


Packages can be nested into other packages

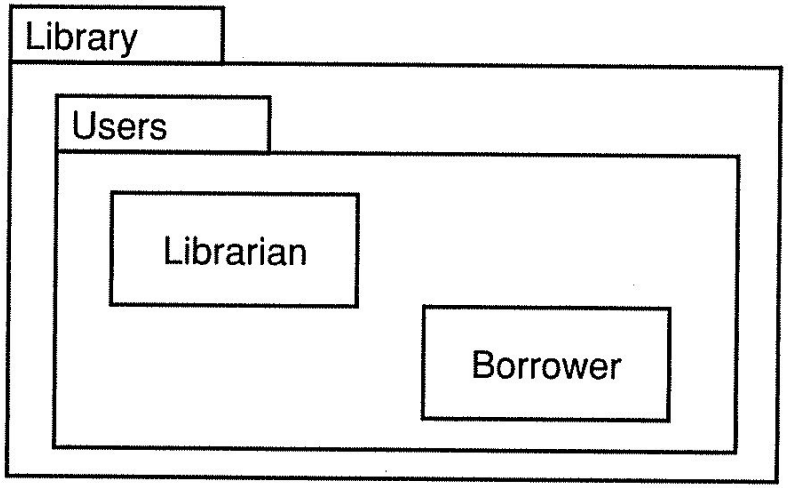
Users is nested into Library

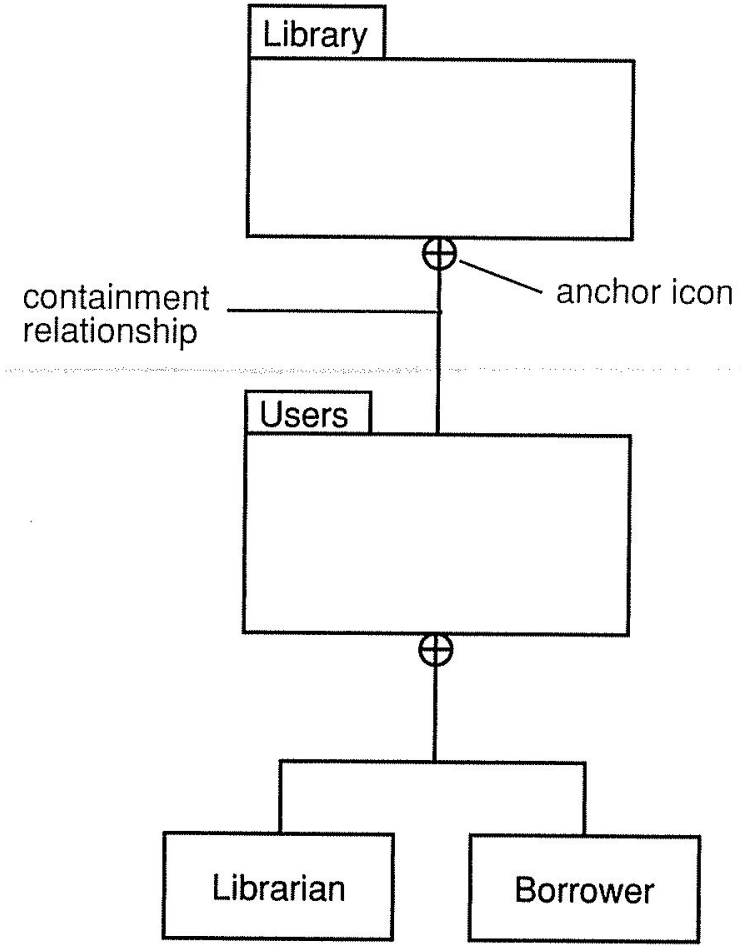


Nested packages may also be represented with an alternative syntax

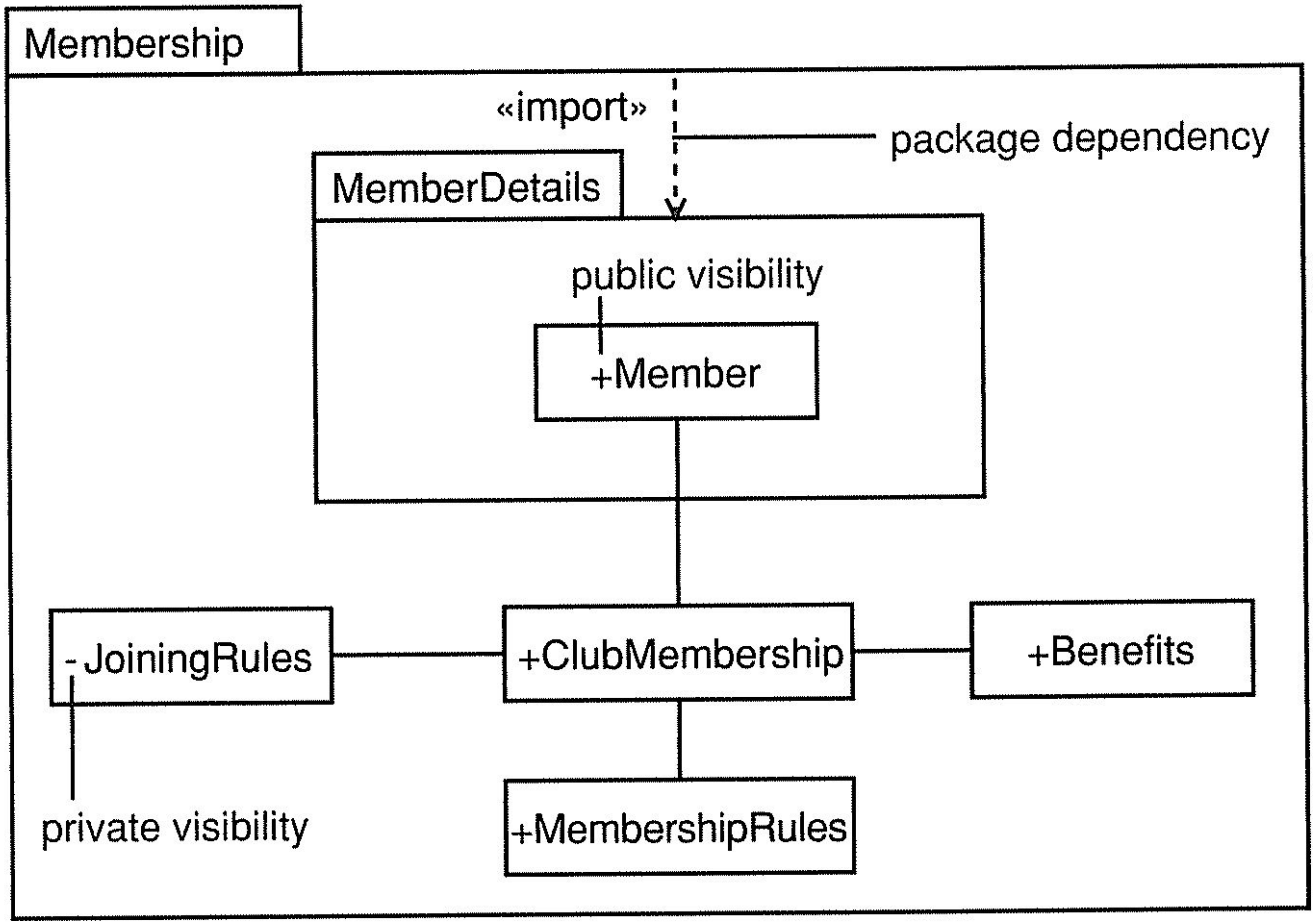


These are equivalent: use the one which is most effective for communication in each case



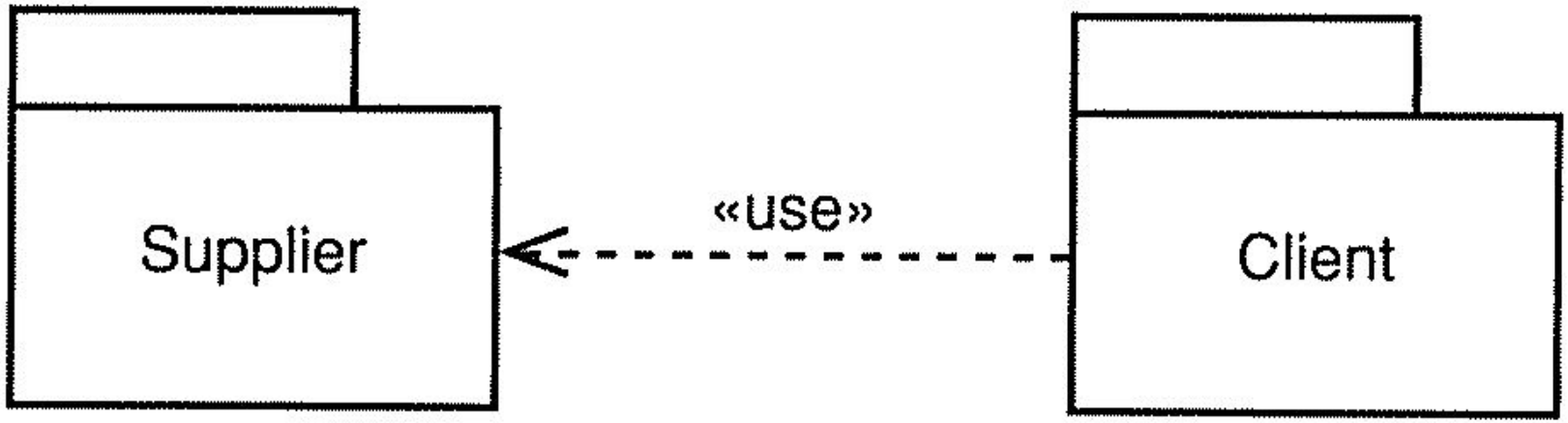


A dependency relationship indicates that one package depends in some way on another package



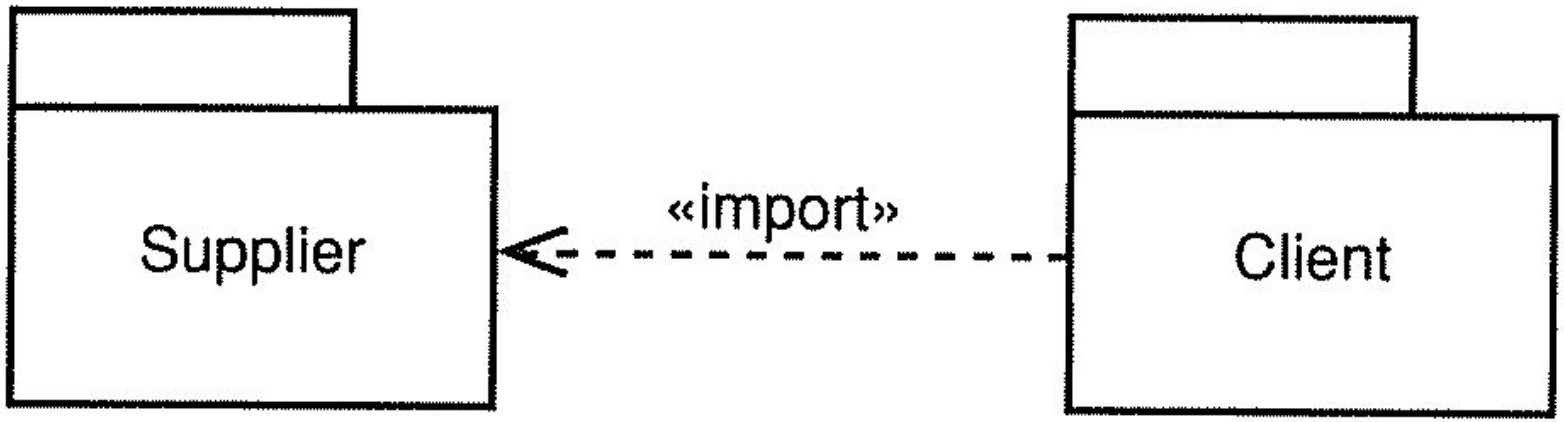
<<use>> dependency

- An element in the client package uses a public element in the supplier package in some way
  - The client **depends** on the supplier
- This is the default dependency



<<import>> dependency

- Public elements of the supplier namespace are added as public elements of the client namespace
- Elements in the client can access all public elements in the supplier without qualified names

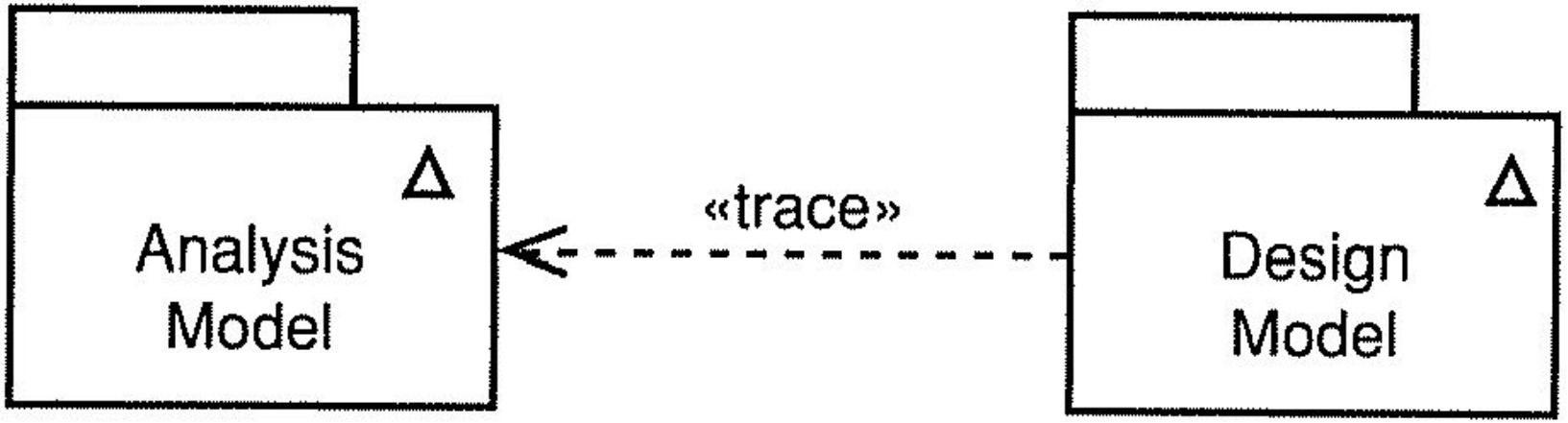


<<access>> dependency

- Public elements of the supplier namespace are added as private elements of the client namespace
- Elements in the client can access all public elements in the supplier without qualified names

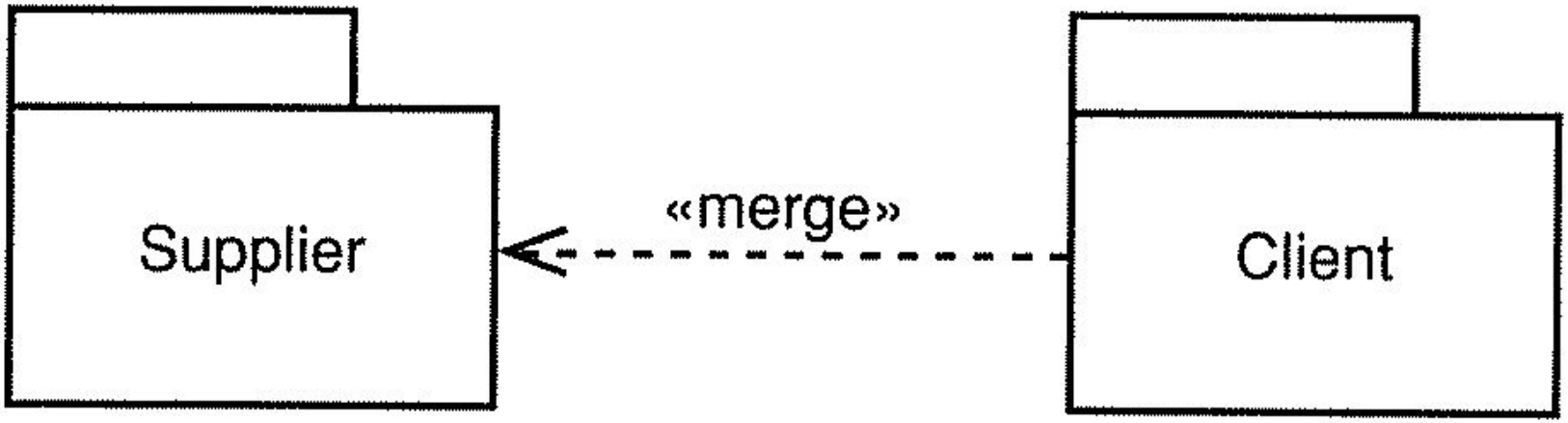
<<trace>> dependency

- <<trace>> usually represents a historical development of one element into another more developed version of it
  - A relationship between models rather than elements



<<supplier>> dependency

- Public elements of the supplier package are merged with elements of the client package
- This is only used in metamodelling
  - Metamodelling is used for modelling modelling languages

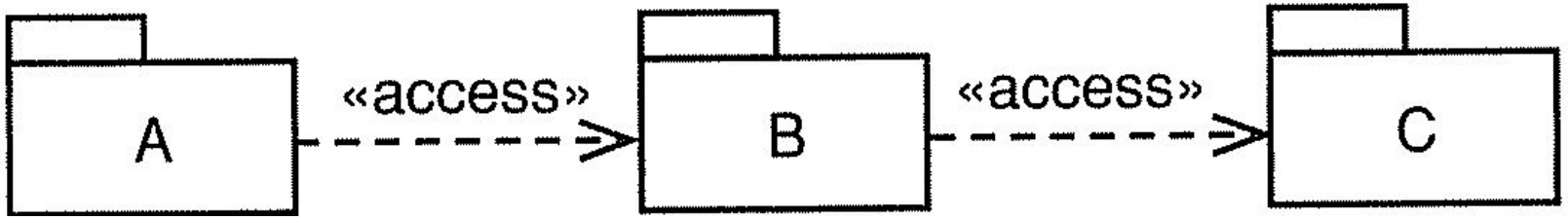


Dependency transitivity

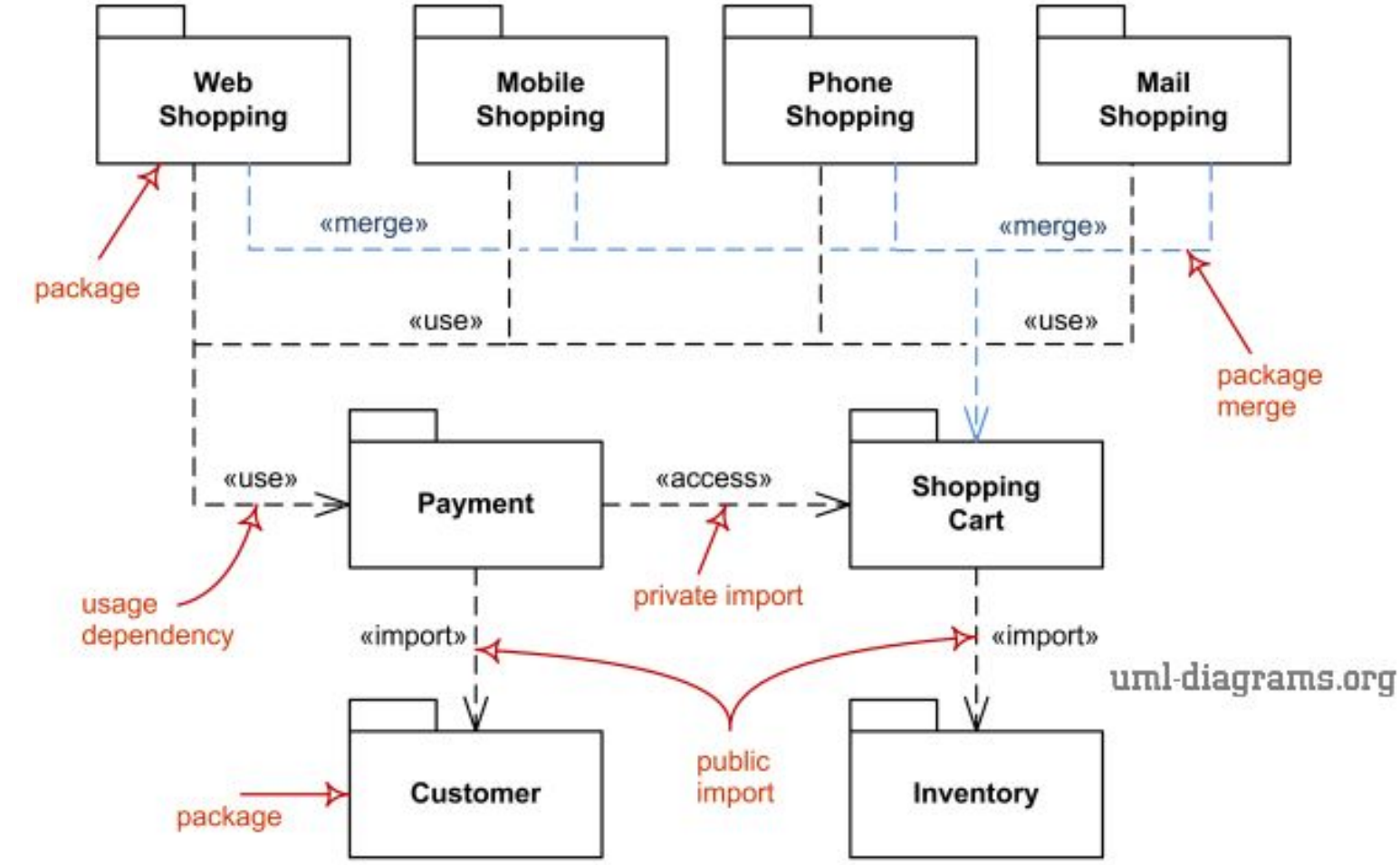
- Some dependencies are transitive, other are not
  - <<import>> is transitive
    - Imported elements are made public
  - <<access>> is not transitive
    - Accessed elements are made private

Lack of transitivity in <<access>> allows you to manage coupling and cohesion

- Nothing is accessed unless it is explicitly accessed
  - public elements in package C become private elements in package B
  - public elements in package B become private elements in package A
  - elements in package A have no access to elements in package C



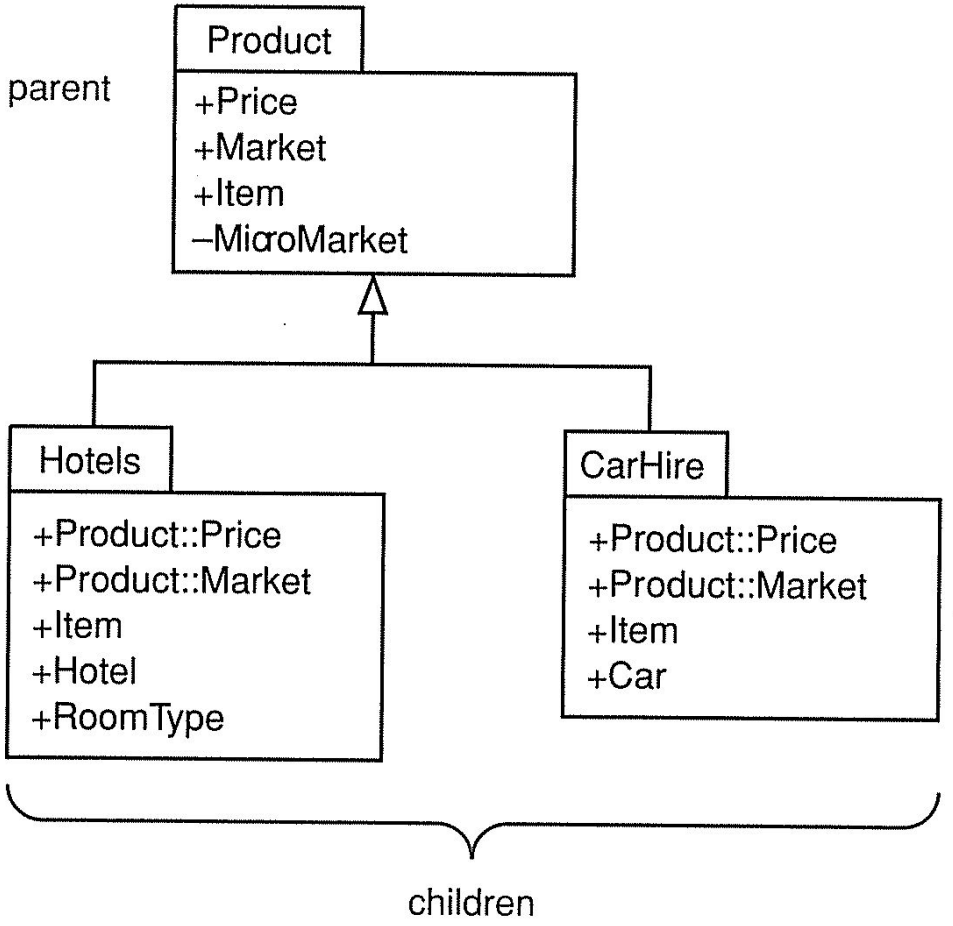
Elements of a Package Diagram: summary



Package model development considerations

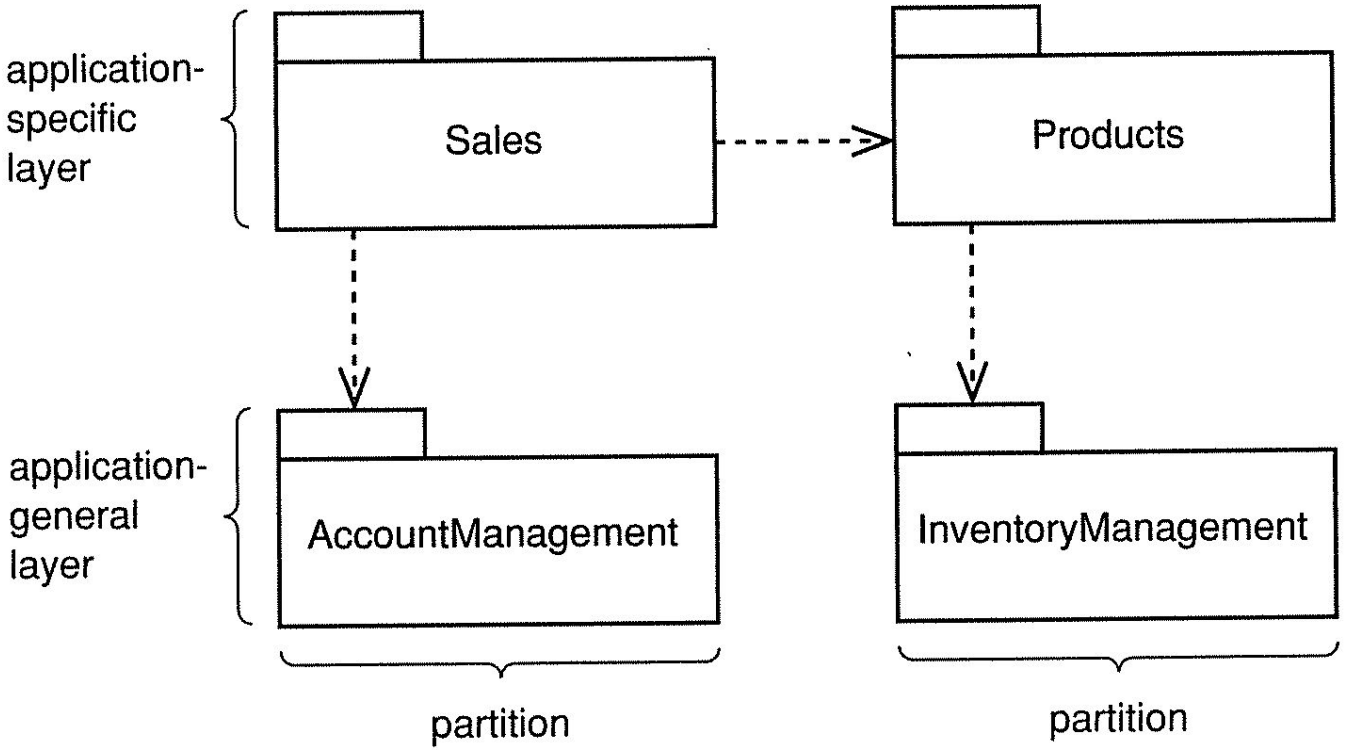
Package generalization

- Child packages inherit elements from their parent
- Child packages may override parent elements
- Child packages may add new elements
- The substitutability principle must apply



Architectural analysis

- Partitions related classes into analysis packages and then layers the packages



Architectural analysis goals

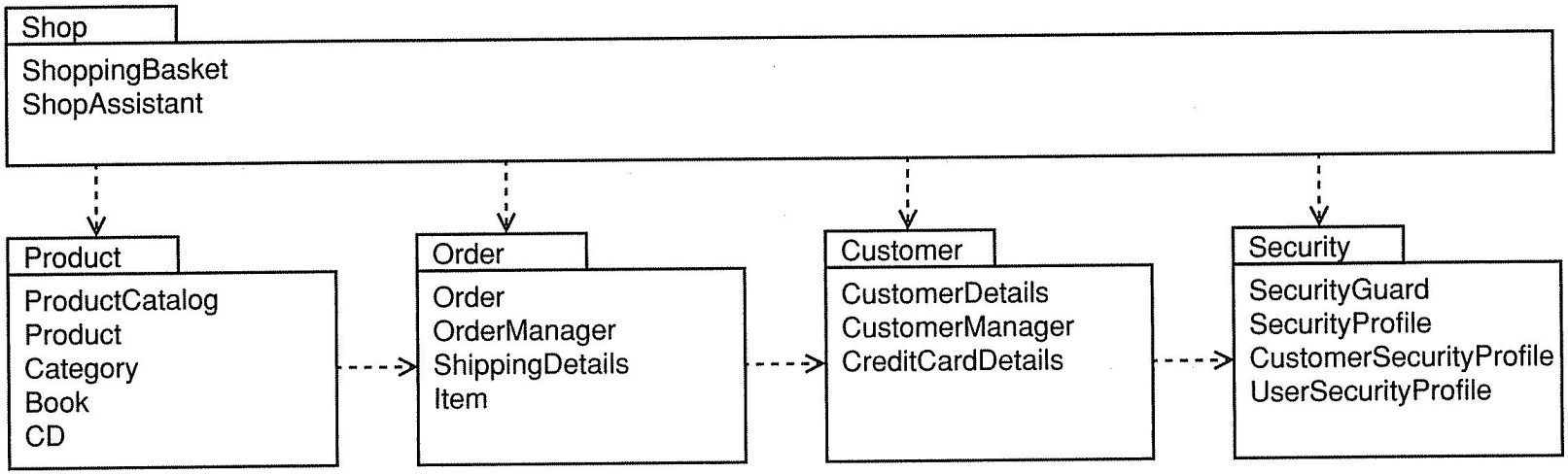
- Minimize dependencies between analysis packages
- Minimize the number of public elements in each analysis package
- Maximize the number of private elements in each analysis package
- **Goal: minimize coupling between packages**

Finding analysis packages

- Look for classes that form a cohesive structure
- Look for inheritance hierarchies
- Use cases may also be a source for packages
  - Packages may be cohesive from a business perspective
  - This may not be the case, as a use case may use cross-cutting classes - those used in several packages

Cleaning-up your packages may involve

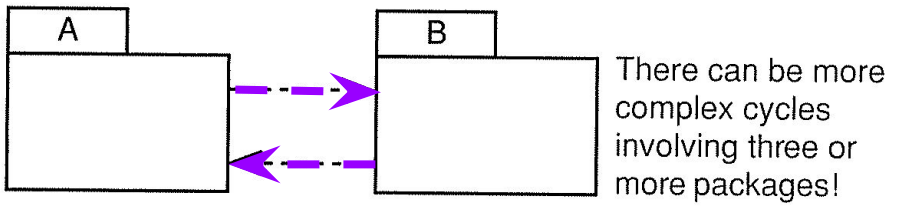
- Moving classes between packages
- Adding packages
- Removing packages
- Aim for:
  - Low coupling **between packages**
  - High cohesion **within each package**

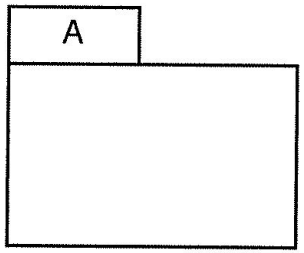


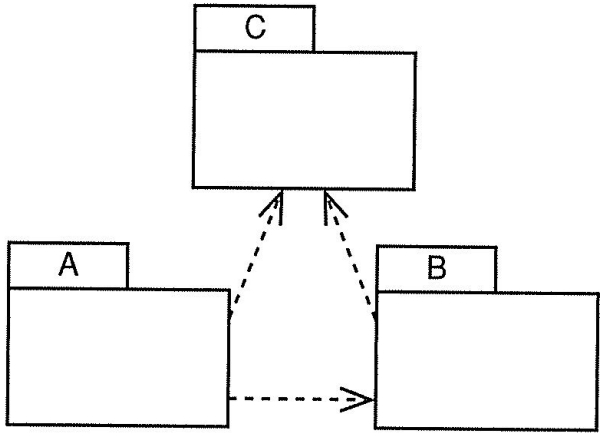
Avoid cyclic packages dependencies

Possible workarounds:

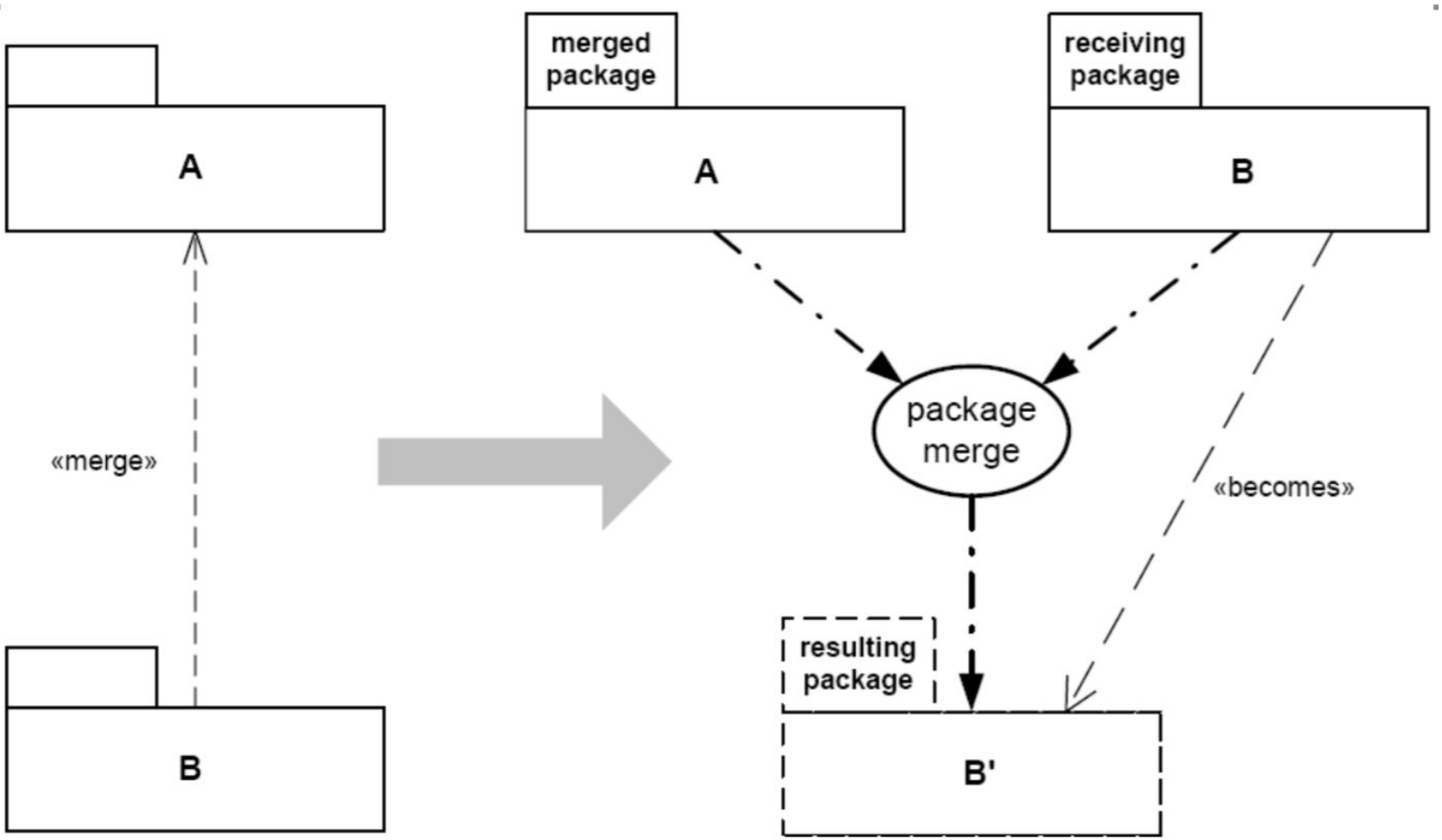
- **Merge** packages
- **Split:** Factor common elements to a third package, have the two packages original packages depending on the new one and recalculate dependencies





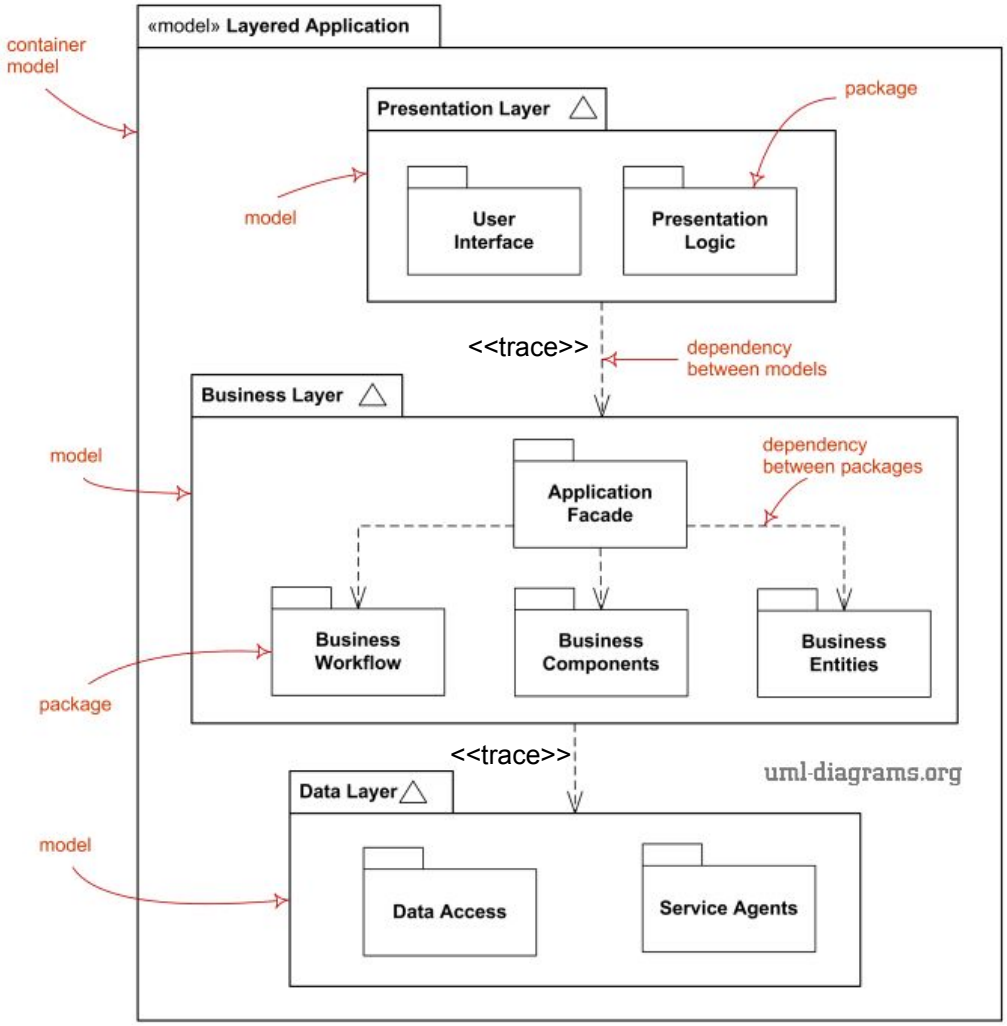


What really happens when we merge packages



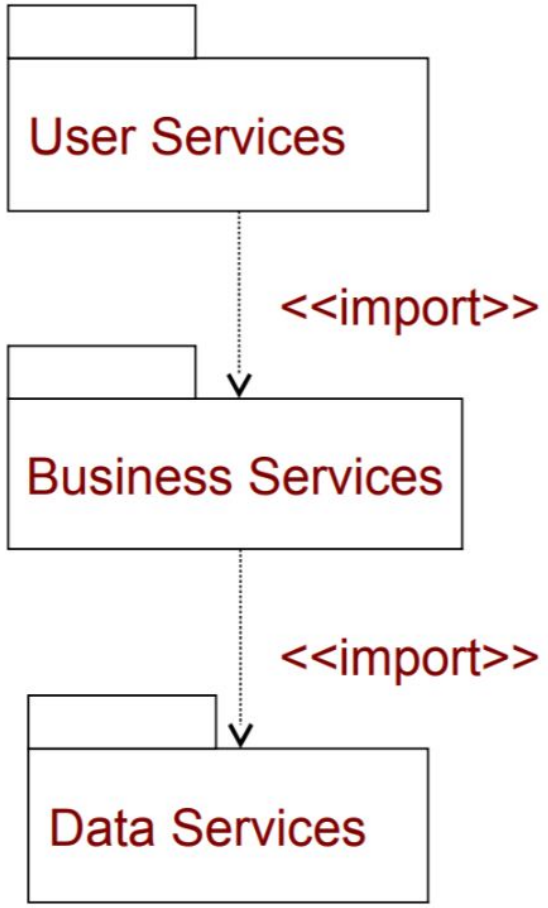
Model diagrams show some abstraction or specific view of a system, to describe some architectural, logical, or behavioral aspects of the system

Elements of a Model Diagram

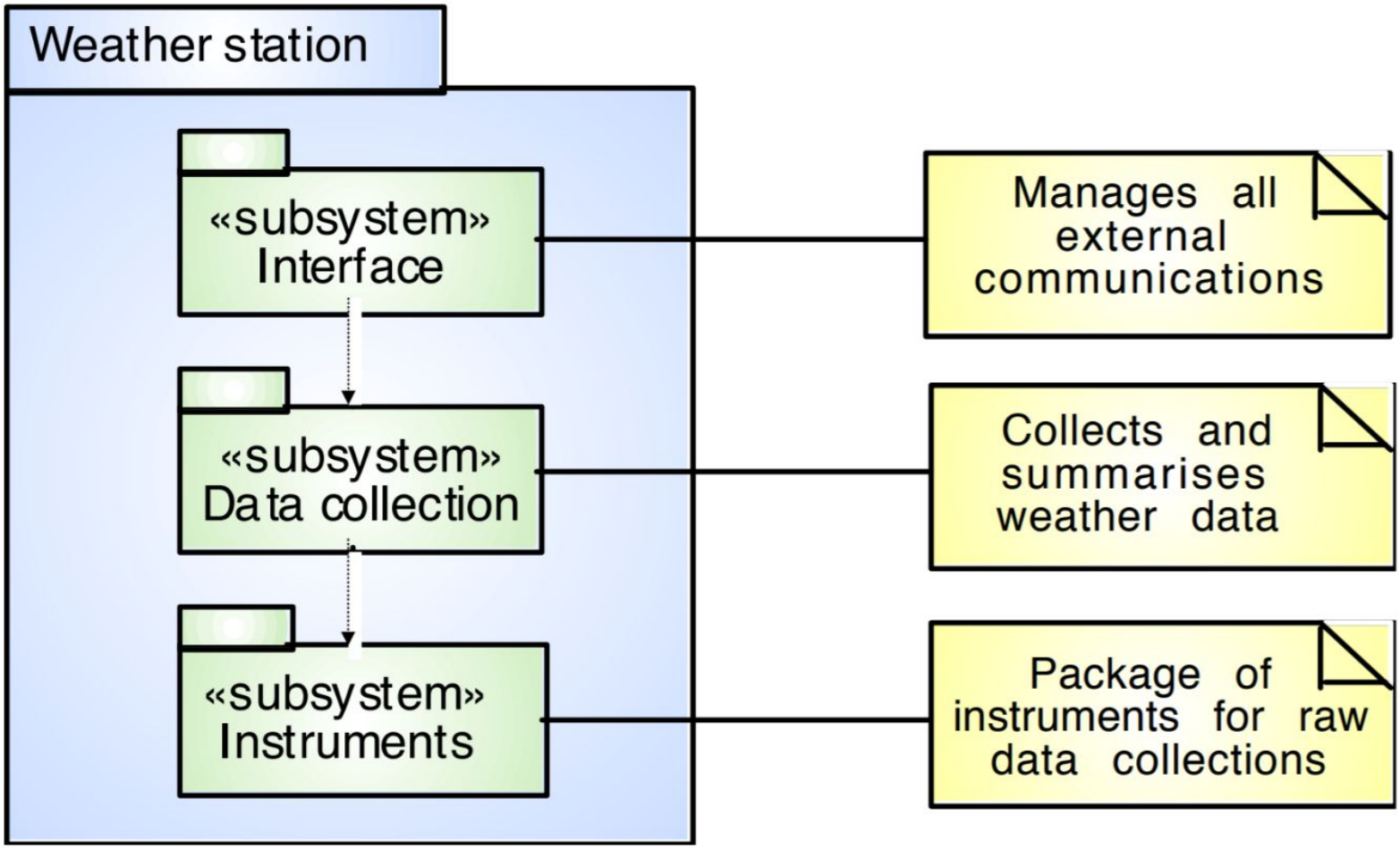


Three-tier architecture

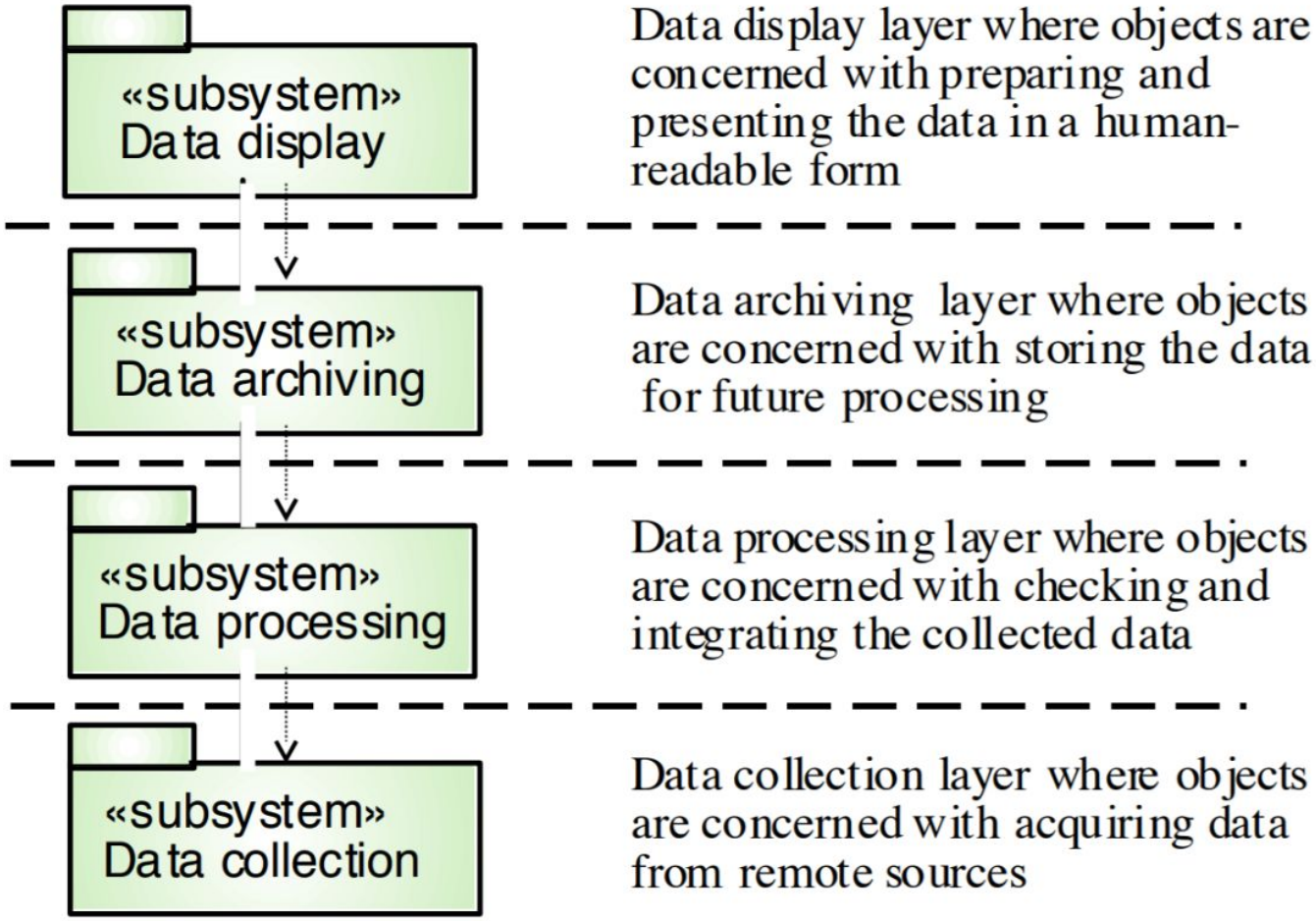
- Each tier directly assesses to the tier right below it
  - Top tiers depend on those below them
  - Bottom tiers are “unaware” of top tiers



Example of a three-tier architecture



Layered architectures (aka n-tier architectures)



Bibliography

Jim Arlow and Ila Neustadt, “UML 2 and the Unified Process”, Second Edition, Addison-Wesley 2006

- Chapter 11