UML and Object-Orientation

The Fundamentals

Department of Computer Science,
The University of York, Heslington, York YO10 5DD, England
http://www.cs.york.ac.uk/~paige

THE UNIVERSITY of York

Context of this work

- The present courseware has been elaborated in the context of the MODELWARE European IST FP6 project (http://www.modelware-ist.org/).
- Co-funded by the European Commission, the MODELWARE project involves 19 partners from 8 European countries. MODELWARE aims to improve software productivity by capitalizing on techniques known as Model-Driven Development (MDD).
- To achieve the goal of large-scale adoption of these MDD techniques, MODELWARE promotes the idea of a collaborative development of courseware dedicated to this domain.
- The MDD courseware provided here with the status of open source software is produced under the EPL 1.0 license.

The Unified Modelling Language (UML)

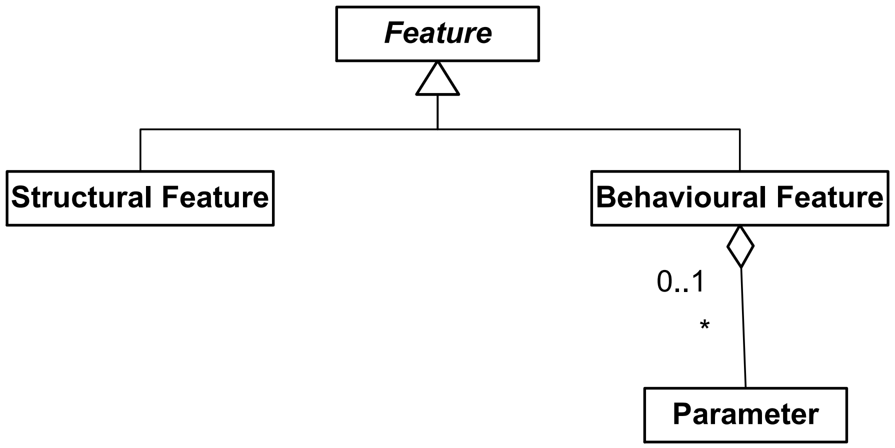
- What is the UML?
  - It is a visual language for describing systems.
- It is a successor to the wealth of OO analysis and design methods of the 80s and 90s.
  - It unifies methods of Booch, Rumbaugh (OMT), and Jacobson.
- It is an OMG standard
  - Widespread use of UML 1.x.
  - Propagating to UML 2.0 (slowly).
- It is a modelling language, and **not** a method (== language + process).

Method vs. Language

- The UML is, in no uncertain terms, a modelling language.
- OMT, Objectory, Fusion, etc., are methods.
- UML consists of two main parts:
  - the graphical notation, used to draw **models**;
  - a metamodel, which provides rules clarifying which models are valid and which are invalid
- UML does not specify a development or management process
  - ... though see the Rational Unified Process.

Aside: What Does a Metamodel Look Like?

- A metamodel captures the well-formedness constraints on diagrams (more on this later).
- e.g., a Feature is either Structural or Behavioural
- Example for UML:



What is UML Used For? (...and why bother!)

- UML is used to model characteristics of systems:
    - **static structural characteristics**, e.g., classes, interfaces, relationships, architectures (class diagrams)
    - **dynamic characteristics**, e.g., object creation, messages, distribution (interaction diagrams)
    - **behavioural aspects**, e.g., reactions to messages, state changes (statecharts)
- It is typically (though not always) used during analysis and design, before code is created.
- Newer methods apply UML in concert with code, eg., agile modelling, round-trip engineering.

UML – Some Criticisms and Responses

- You don't write code immediately.
  - "What are these pictures? You can't execute that? You're wasting time and money! Write code."
  - A short amount of time spent on modelling may save a great deal of time coding.
  - See Extreme Programming/Agile Modelling.
- UML is very complex (200+ page syntax guide).
  - Very true.
  - We'll focus on a selection of very useful diagrams.
- You can easily produce inconsistent models using UML.
  - ie., multiple views of the same system.

UML Diagrams

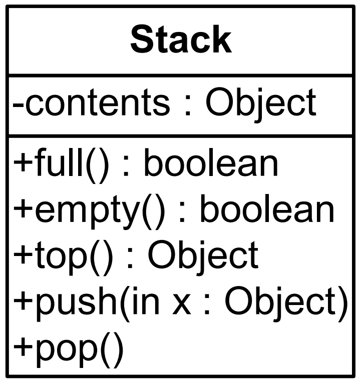
- **Class diagrams**: the essentials and the advanced concepts.
- **Interaction diagrams**: for specifying object interactions and run-time behaviour via message passing (communication diagrams in UML 2.0)
- **State charts**: for specifying object behaviour and state changes (reactions to messages).
- **Use case diagrams:** for capturing scenarios of system use.
- **Physical diagrams**: for mapping software elements to physical components, e.g., hardware devices.
- Lots more: extension mechanisms, activity diagrams, …

The Critical OO Concept

- The most important concept in OO is the **class**.
- class = module + type
  - Module because it has a data part and an operation part.
  - Type because you can declare instances.
  - **Note:** this is not the only definition of class that is out there!
- Classes provide features that are declared in its interface.
- All of OO analysis, design, and programming revolves around classes, which suggests the question

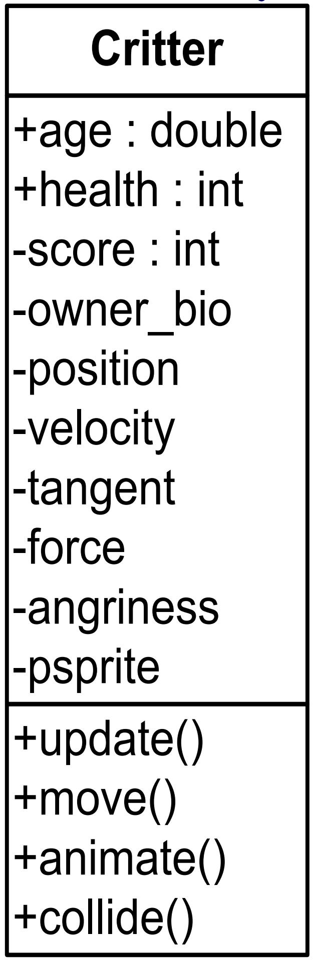
<u>**How do we find the classes?**</u>

Stack in UML



- Three sections in the diagram of a class:
  - class name, attributes, operations
  - CASE tools differ in the syntax used for arguments.
  - variants on this structure (see later)
- Roughly corresponds to entities in E-R models, but these include operations as well as data/state.

Another Example



- Attributes can be as specific as you like (e.g., integer type).
- Optional specification of arguments and result type for operations.
- These must be filled in when you generate code.
- Some tools restrict the allowed types (e.g., Visio), but pure UML has no restrictions.

Class Features

- A class can have two general kinds of features:
  - **fields** or **attributes** used to hold data associated with objects, e.g., `contents:Object;`
  - **operations**, used to implement behaviour and computations, e.g., `pop()`, `top()`
- Features are all accessed using "dot" notation.
  - **o.f(...);**
  - Every feature access has a target and a feature.
- Two kinds of operations: functions and procedures.

Operations and Methods

- An **operation** in UML specifies the services that a class can provide to clients.
  - It is implemented using a method.
- Full UML syntax for an operation:

```
visibility name (param_list):type {constraint}
```

- `visibility` is + (public), - (private), or # (protected)
- The constraint may include pre/post-conditions.
- Think of an operation as an interface that is implemented by one or more methods.
  - Thus, UML classes typically do not include overloaded methods.

Functions and Procedures

- Functions are operations that calculate and return a value.
  - They **do not** change the state of an object, i.e., make persistent, visible changes to values stored in fields.
- Procedures are operations that change the state of an object.
  - They **do not** return any values.
- Separating functions from procedures is useful in simplifying systems, for verification, and testing.

Constructors

- A class should contain special operations called constructors.
    - Invoked when an object is allocated and attached to a variable.
- Constructors (in Java) have the same name as the class. There may be several of them (overloaded).

```
class Person {
   public Person() { name=NULL; age=0; }
   public Person(int a){ name=NULL; age=a; }
   public Person(String n, int a){ ... }
}
```

- Constructors are invoked automatically on allocation
- UML **<<constructor>>** stereotype.

Visibility

- Each feature of a class may be visible or invisible to other classes - clients.
- It is recommended, but not required, to **not** allow clients to write to/change fields directly.
  - **Why?**
- Each feature in a class can be tagged with visibility permissions (Java/UML):
  - **`private (-)`**: the feature is inaccessible to clients (and children).
  - **`public (+)`**: the feature is completely accessible to clients
  - **`protected (#)`**: the feature is inaccessible to clients (but accessible to children).
  - UML allows you to define language-specific visibility tags.

Class Diagrams - Essentials

- A **class diagram** describes the classes in a system and the static relationships between them.
- Two principle types of static relationships:
  - **associations** (e.g., a film may be associated with a number of actors).
  - **generalizations** (e.g., a tractor is a generalization of a vehicle).
- Class diagrams also depict attributes and operations of a class, and constraints on connections.

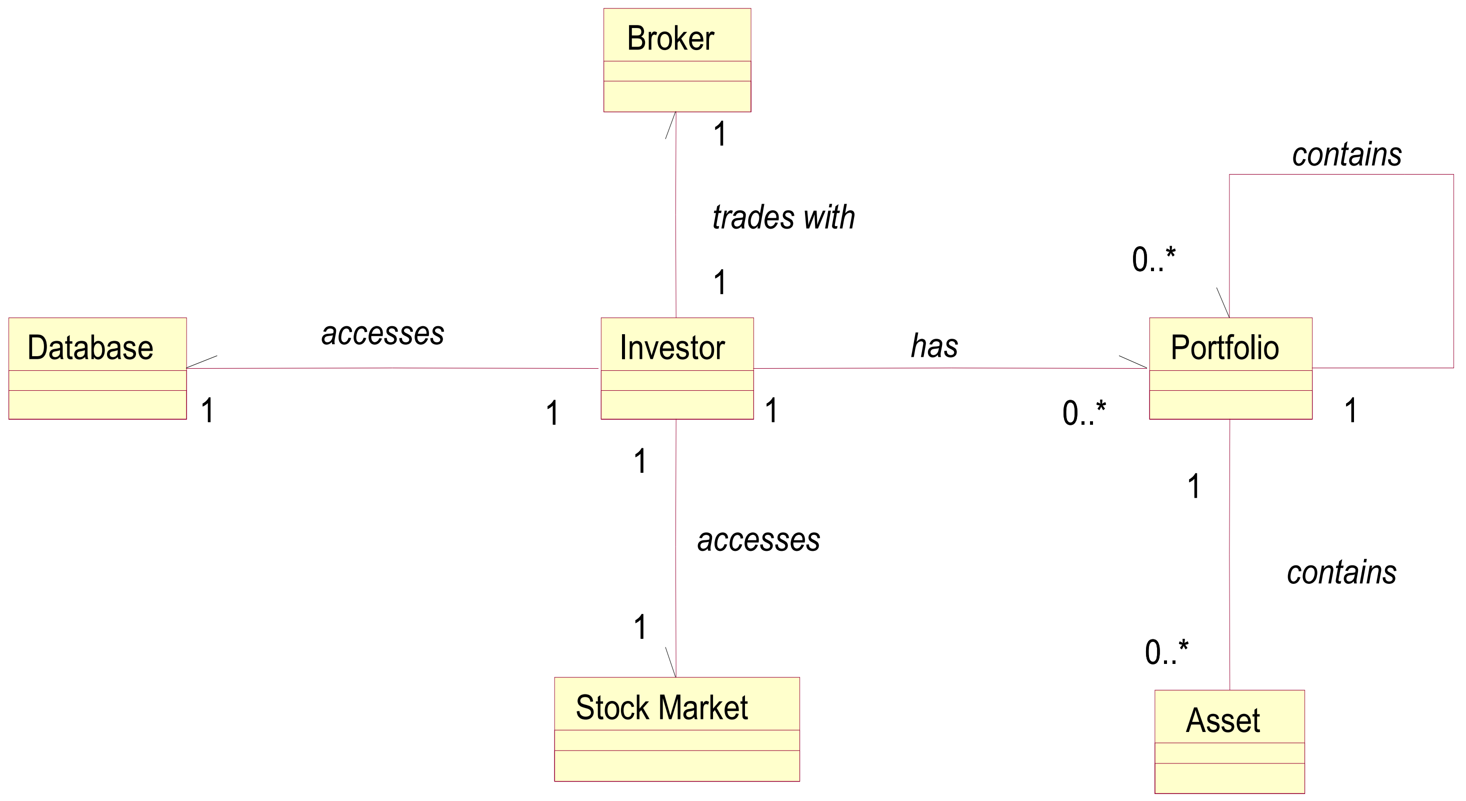
Class Diagrams - Three Perspectives of Use

- Class diagrams can be used in three different ways:
  1. Conceptual modelling: drawn to represent concepts in the problem domain (e.g., cars, street lights, people).
  2. Specification modelling: drawn to represent the interfaces, but not implementations, of software.
  3. Implementation modelling: drawn to represent code.
- Conceptual modelling is often done during analysis and preliminary design.
- Divisions between perspectives are 'fuzzy' and it boils down to who is your intended audience, and what message do you want to get across.

Concepts and Conceptual Modelling

- **Concept**: A real world entity of interest.
- **Structure**: relationships between concepts.
- **Conceptual Modelling**: describing (e.g., using UML) concepts and structures of the real world.
- Class diagrams are commonly used for conceptual modelling.
- They are a way to model the problem domain and to capture business rules.
    - They do **not** depict the structure of software!!!

Example: Conceptual Model/Class Diagram



Concepts and Models

- Concepts can be identified using noun and noun phrases in the system requirements
  - ... at least as a first-pass approximation ...
- The conceptual model is used as the basis for producing a specification model.
  - Classes appearing in the conceptual model may occur in the specification model.
  - They may also be represented using a set of classes.
  - Or they may vanish altogether!

Finding the Concepts/Classes

- All OO methods hinge on this!
  - Much more critical than discovering actors or scenarios.
  - It is the process of finding concepts that really drives OO development - there is nothing OO about use cases!
- We are not finding objects: there are too many of those to worry about - rather, find the recurring data abstractions.

THERE ARE NO RULES!

- ... but we do have good ideas, precedents, and some known pitfalls!
  - ... and we'll look at what others have done.

Noun/Verb Phrase Analysis

- Many books suggest the following superficial approach:

  "Take your requirements document and underline all the verbs and all the nouns. Your nouns will correspond to classes, your verbs to methods of classes."
- **Example:** "The elevator will close its door before it moves to another floor."
- Suggests classes Elevator, Door, Floor, and methods move, close.
- This is too simple-minded:
  - suffers from the vagaries of natural-language description.
  - finds the obvious classes and misses the interesting ones.
  - often finds the totally useless classes

Useless Classes

- Noun/verb approaches usually find lots of useless classes.
- Does Door deserve to be a class?
  - Doors can be opened and closed (anything else?)
  - So include a function door_open and two procedures close_door and open_door in Elevator.
- The relevant question to ask is:

Is Door a separate data abstraction with its own clearly identified operations, or are all operations on doors already covered by operations belonging to other classes?

More Useless Classes

- **Example:** what about class Floor?
  - Are properties of floors already covered, e.g., by Elevator?
- Suppose the only interesting property of floors is their floor numbers; a separate class may not be needed.
- The question is the proposed class relevant to the system? is the critical one to ask.
  - Answer this based on what you want to do with the class.

Missing Classes

- Noun/verb approaches usually miss important classes, typically because of how requirements are phrased.
- **Example:** "A database record must be created each time the elevator moves from one floor to another."
- Suggests Database Record as a class.
  - But it misses the concept of a Move! We clearly need such a class.

```
class Move {
 public Floor initial, final;  // could be int
 public void record(Database d){ ... };
    ...
 }
```

Missing Classes (redux)

- Now suppose the requirement instead read: "A database record must be created for every move from one floor to another."
  - Move is now a noun, and the noun/verb method would have found it.
- So treat this method with caution - it is reasonable to use it as a first-pass attempt, but be aware that it
  - misses classes that do not appear in requirements
  - misses classes that don't correspond to real-world concepts (these are the really interesting ones!)
  - finds classes that are redundant, useless, or serve to confuse.

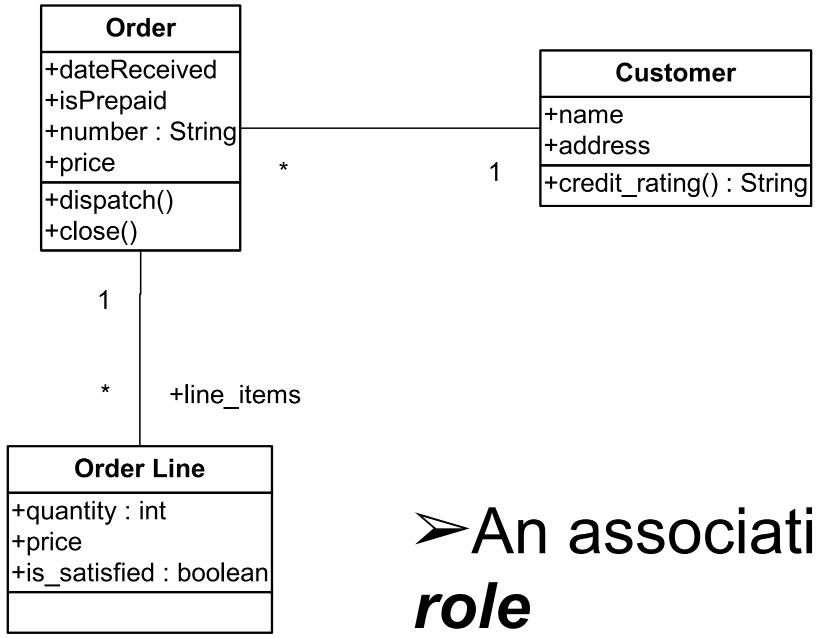
Interesting Classes

- Will have (several) attributes
- Will have (several) operations/methods
  - Will likely have at least one state-change method.
- Will represent a recurring concept, or a recurring interface in the system you are building.
- Will have a clearly associated data abstraction.
- ...

Class Relationships

- Individual classes are boring.
- Connecting them is what generates interesting emergent behaviour.
- Connecting them is also what generates complicated problems and leads to errors!
- So we want to be careful and systematic when deciding how to connect classes.
- In general there are only two basic ways to connect classes:
  - by client relationships (several specialisations in UML)
  - by inheritance relationships (several specialisations)

Associations (Client-Supplier)



- Associations represent relationships between instances of classes (a client and a supplier).
- Each association has two **association ends**.

➢An association end can be named with a ***role***

➢If there is no role, you name an end after the target class.

➢Association ends also can have multiplicities (e.g., 1, *, 0..1)

Multiplicity

- Multiplicities are constraints that indicate the number of instances that participate in the relationship.
- Default multiplicity on a role is 0..*.
- **Other useful multiplicities:**

1:	exactly one instance
*:	same as 0..* (arbitrary)
1..*:	at least one
0..1:	optional participation

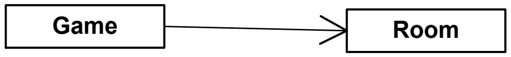
- You can, in general, specify any number, contiguous sequence n..m, or set of numbers on a role.
- Note: multiplicities constrain **instances!**

Associations Imply Responsibilities

- An association implies some responsibility for updating and maintaining the relationship.
- e.g., there should be a way of relating the Order to the Customer, so that information could be updated.
- This is not shown on the diagram: it can thus be implemented in several ways, e.g., via an add_order method.
- Responsibilities **do not** imply data structure in a conceptual model or a specification model!
- If you want to indicate which participant is responsible for maintaining the relationship, add navigability.

Navigability and Associations

- A navigable association indicates which participant is responsible for the relationship.



➢Game is responsible for the relationship.

➢ In an implementation diagram, this may indicate that a Game object refers to ("points to", or "has-a") a Room object.

➢ Note the default "undirected" association really means bidirectional!

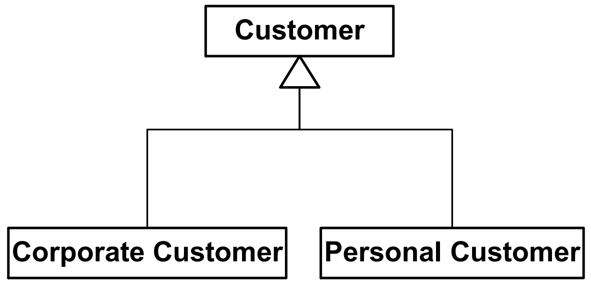
➢Directed associations are typically read as "has-a" relationships.

Attributes vs. Associations

- An **attribute** of a class represents state – information that must be recorded for each instance.
- e.g., name and address for Customer indicate that customers can tell clients their names and addresses
- But the association from Customer to Order indicates that customers can tell clients about their orders.
- So what's the difference between attributes and associations?
  - At the implementation level, usually nothing.
- Think of attributes as "small, simple objects" that don't have their own identity.

Generalization

- Personal and Corporate customers have similarities and differences.
  - Place similarities in a Customer class, with Personal Customer and Corporate Customer as generalizations.



➢Everything we say about Customers can be said about its generalizations, too.

➢At the specification level, the subtype's interfaces must conform to that of the supertype.

➢Substitutability.

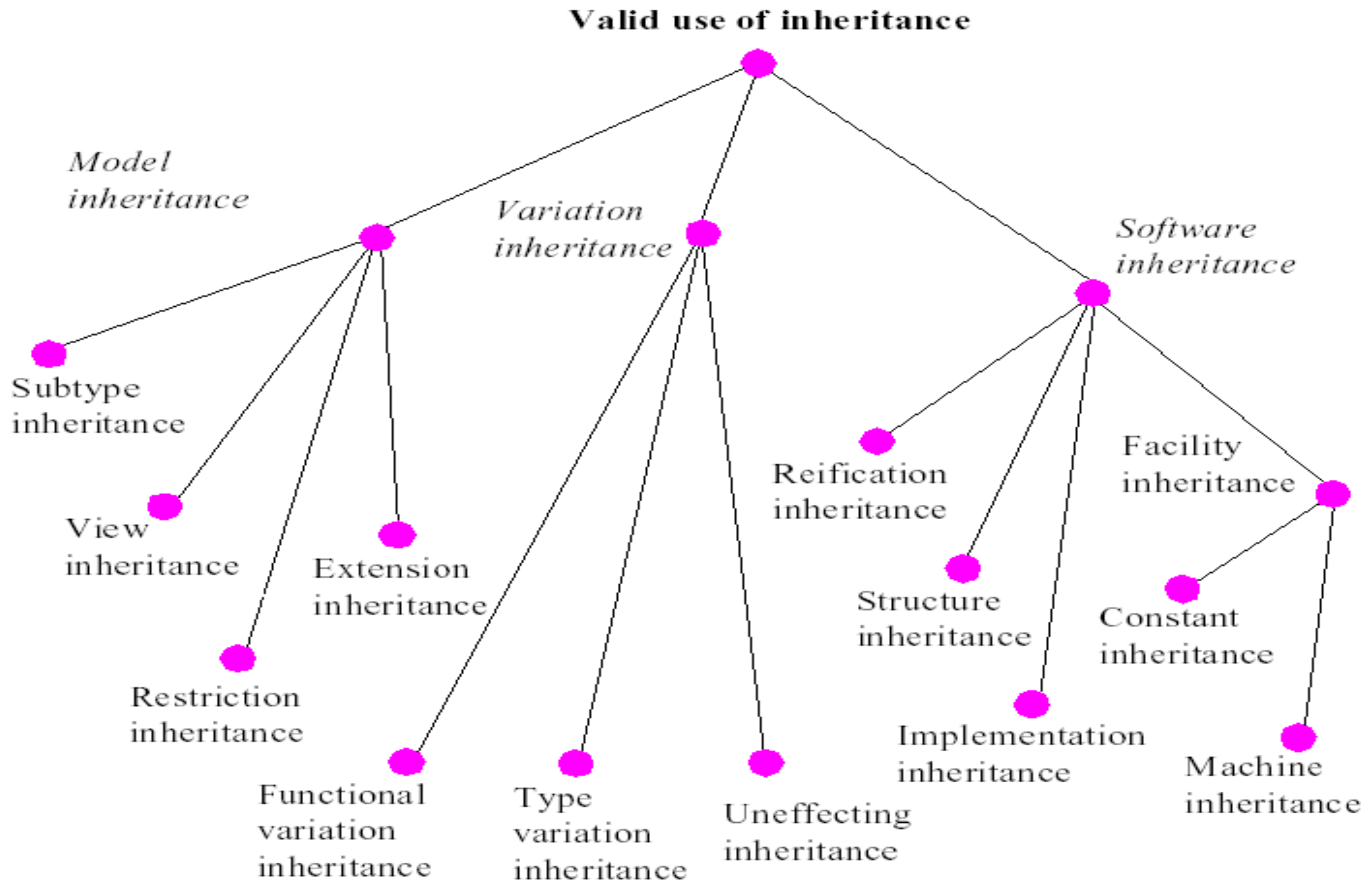
Generalization vs. Inheritance

- Generalization is not the same thing as inheritance (extension in Java).
  - At the implementation level, we may want to implement it using inheritance (but we don't have to – see delegation later).
- <u>**Advice:**</u> always ensure that conceptual generalization applies.
  - This is the "is-a" relationship.
- Makes it easier to implement later on.

Inheritance Modelling Rule

- Given classes A and B.
- If you can argue that B is also an A, then make B inherit from A.
- **Note:** this is a general rule-of-thumb, but there will be cases where it does not apply.
  - If you can argue that B is-a A, it's easy to change your argument so that B **has-a** A.

Different Types of Inheritance



Different Types of Inheritance

- **Subtype:** modelling some subset relation.
- **Restriction:** instances of the child are instances of the parent that satisfy a specific constraint (e.g., RECTANGLE inherits SQUARE)
- **Extension:** adding new features.
- **Variation:** child redefines features of parent.
- **Uneffecting:** child abstracts out features of parent from at least one client view.
- **Reification:** partial or complete choice for data structures (e.g., parent is TABLE, reifying child classes are HASH_TABLE).
- **Structure:** parent is a structural property (COMPARABLE), child represents objects with property.

Problems with "is-a"

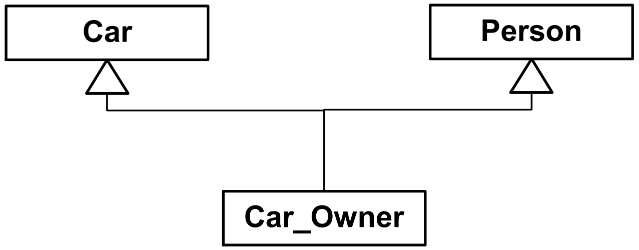
- It is important to be careful with "is-a" relationships, since confusion can arise with instantiation.
- Example:
  1. Donna is a Siberian Husky.
  2. A Siberian Husky is a Dog.
  3. A Dog is an Animal.
  4. A Siberian Husky is a Breed.
  5. A Dog is a Species.
- Combine sentences 1, 2 and 3.
- Combine 1, 4; combine 2, 5. **What's going on?**

Instantiation and Generalization

- The problem is that some of the phrases 1-5 are instantiation phrases, some are generalizations.
  - Donna is an instance of Siberian Husky.
  - Siberian Husky is a generalization of Dog.
- Generalization is transitive; instantiation is not.
- Be careful when using “is-a” - make sure you really have a generalization relationship and not an instantiation.

How Not to Use Inheritance

- Suppose that we have a class Car and a class Person.
  - Put them together to define a new class Car_Owner that combines the attributes of both.



➢ Every Car_Owner is both a Car and a Person.

➢ Correct relationship is association between Car_Owner and Car.

Association vs. Generalization

- Basic rule is simple: directed association represents "has-a", generalization represents "is-a".
  - So why is it sometimes difficult to decide which to use?
- Consider the following statements:
  1. Every software engineer is an engineer
  2. In every software engineer there is an engineer
  3. Every software engineer has an engineer component.
- When the "is-a" view is legitimate, the "has-a" view can be taken as well.
  - The reverse is not normally true.

Rule of Change

- Associations permit change, while generalizations do not.
  - If **B** inherits from ***A*** then every B object is an *A* object and this cannot be changed.
  - If a B object has a component of class *A*, then it is possible to change that component (up to type rules).
- e.g., Person fields can reference objects of type Person, or of any compatible subtype.

Rule of Change

Do not use generalization for a perceived is-a relationship if the corresponding object components may have to be changed at run-time.

Polymorphism Rule

- Very simple: if we want to use polymorphism and dynamic binding, then we use generalization.

Polymorphism Rule

Generalization is appropriate for representing is-a relationships if data structure components of a more general type may need to be attached to references of a more specialized type.

Subtyping and Substitution

- Generalization becomes particularly useful when used with substitution, method overriding, and polymorphism.

```
class Employee extends Person {
  public String job;
  public int salary, employee_id;
  // assume display() inherited from Person
}

public static void main(...){
  Person p=new Person("Homer",38);
  Employee e=new Employee("Homer",38,36000);
  ...
  p = e;          // is this legal?
  p.display();             // is this legal?
}
```

Method Overriding

- Consider a class **`Person`** with method **`display()`**
- The display mechanisms for **`Persons`** are likely inappropriate for **`Employees`** (since the latter will have more or different attributes).
- We want to **override** the version of **`display()`** inherited from **`Person`** and replace it with a new version.

```
 class Employee extends Person {
public void display(){
 super.display();
 System.out.println("Job:"+job_title+"\n");
}
```

Method Overriding Again

- When overriding, in the child class merely provide a new implementation of the method.
- If the signature matches that of an inherited method, the new version is used whenever it is called.
- All methods can by default be overridden in Java
  - No special syntax for overridden operations in UML.
  - Some developers omit the operation in the child class if it is overridden; this can lead to confusion.
  - {leaf} constraint next to the name of an operation prevents overriding in subclasses.

Constraints on Overriding

- You do not have complete freedom!
- Rules have to be obeyed in order to maintain substitutability and static typing in a language.
- Typical rules on overriding:
  - attributes and methods can be overridden by default
  - the overridden method must type conform to an inherited method.
  - the overridden method must correctness conform to the original (see this when we get to contracts).
- **Type conformance:** language dependent; in Java the “exact match rule” is applied.
  - Sometimes called no-variance.
  - In UML, type conformance is a point of semantic variation.

Aside: Contra- and Covariance

- **Contravariance:** replace a type with a more general type.
- **Covariance:** replace a type with a more specific type.
- Also "no-variance" which is Java's "solution"

Contravariance

```
X bar(Y y);  // parent
Par_X bar(Par_Y y);
```

Par_X/Y are supertypes

Covariance

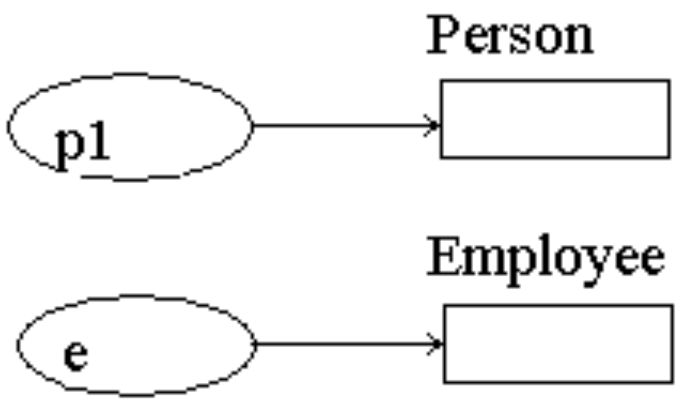
```
A foo(B b);
Child_A foo(Child_B b);
```

Child_A/B are subtypes

Contra- vs. Covariance

- In practice, contravariance hasn't been demonstrated to be at all useful.
  - It is relatively easy to implement.
- No-variance is trivial to implement, but oftentimes is too restrictive, and in Java leads to lots of casting to `Object`.
- Covariance is probably the most useful, but it can require system-level validity checks - very expensive!
  - Recent proposals/work from ETH Zurich attempt to eliminate this and make the mechanism feasible.
  - Prototype implementation in the Eiffel language, work on porting it to the .NET framework.

Dynamic Dispatch

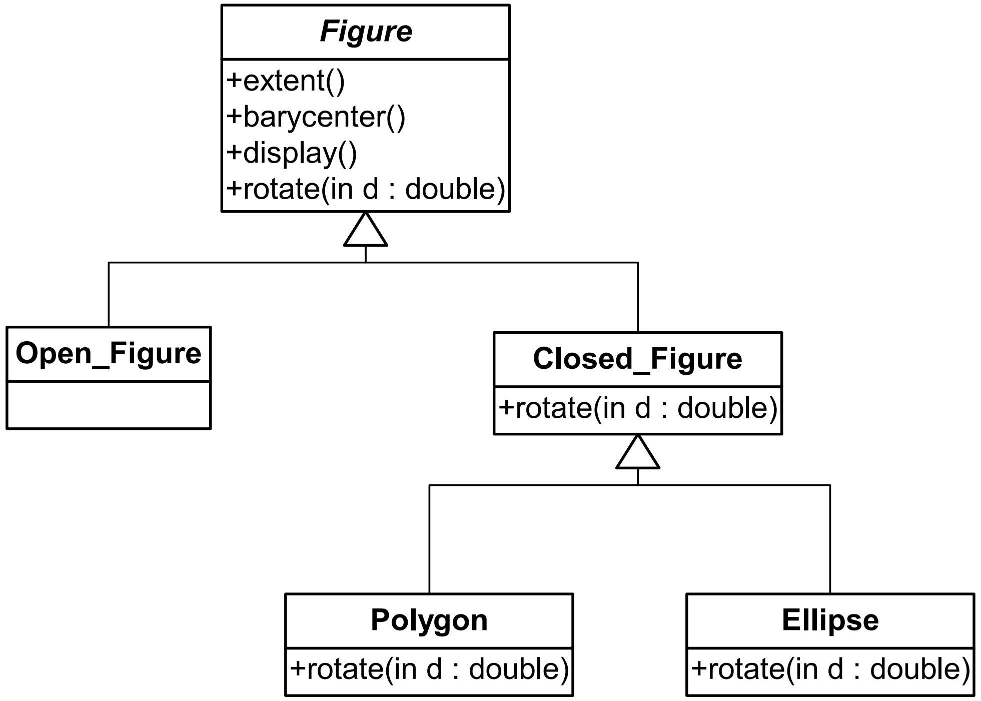


```
p1=new Person();
e=new Employee();
p1 = e;
```

- ➢ Very useful! Invoke methods applicable to the dynamic type of an object.
- ➢ **Dynamic type** is the type of the object attached to variable.
- ➢During execution, a variable can be used to refer to objects of its declared (static) type or of any compatible type, e.g., children and descendents.

Figure Hierarchy

- Typical figure taxonomy excerpt.
  - Open figures: segment, polyline.
  - Other closed figures: triangle, circle, square, ...



Example - Figure Hierarchy

- Consider the hierarchy of figures (Figure, Polygon, Square, etc.) and suppose that we have an array of figures that represent shapes displayed on-screen.
  - All figures are to be rotated by a certain angle around a fixed axis
  - General, maintainable solution required that won't break should we add new figure types.

```
 Figure[] f;     // array of Figures
public void rotate_all(double d){
    int i=0;
    while(i<f.length){
          f[i].rotate(d);     // dynamic
          i++;
    }
}
```

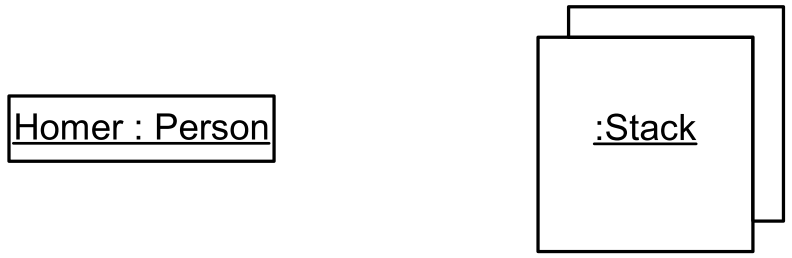
Modelling Dynamic Characteristics

- So far we have considered only how to model static (compile-time) system aspects.
- UML provides facilities for modelling behaviour.
- Behaviour in UML is modelled, in general, by considering:
  - events
  - messages
  - reactions to events
- Some definitions.

Objects

- An OO program declares a number of variables (e.g., attributes, locals, parameters)
  - Each variable can refer (be attached) to an **object**.
- An object is a chunk of memory that may be attached to a variable which has a class for its type.
- Using an object requires two steps: declaration of a variable, and allocation.

Objects in UML



- Each object optionally has a name, e.g., Homer
- Multiple instances of a class represented as on the right.
  - Useful for containers and in representing collaborations of objects.
- Note: this represents declaration+allocation, but **not** allocation by itself! How do we do that?

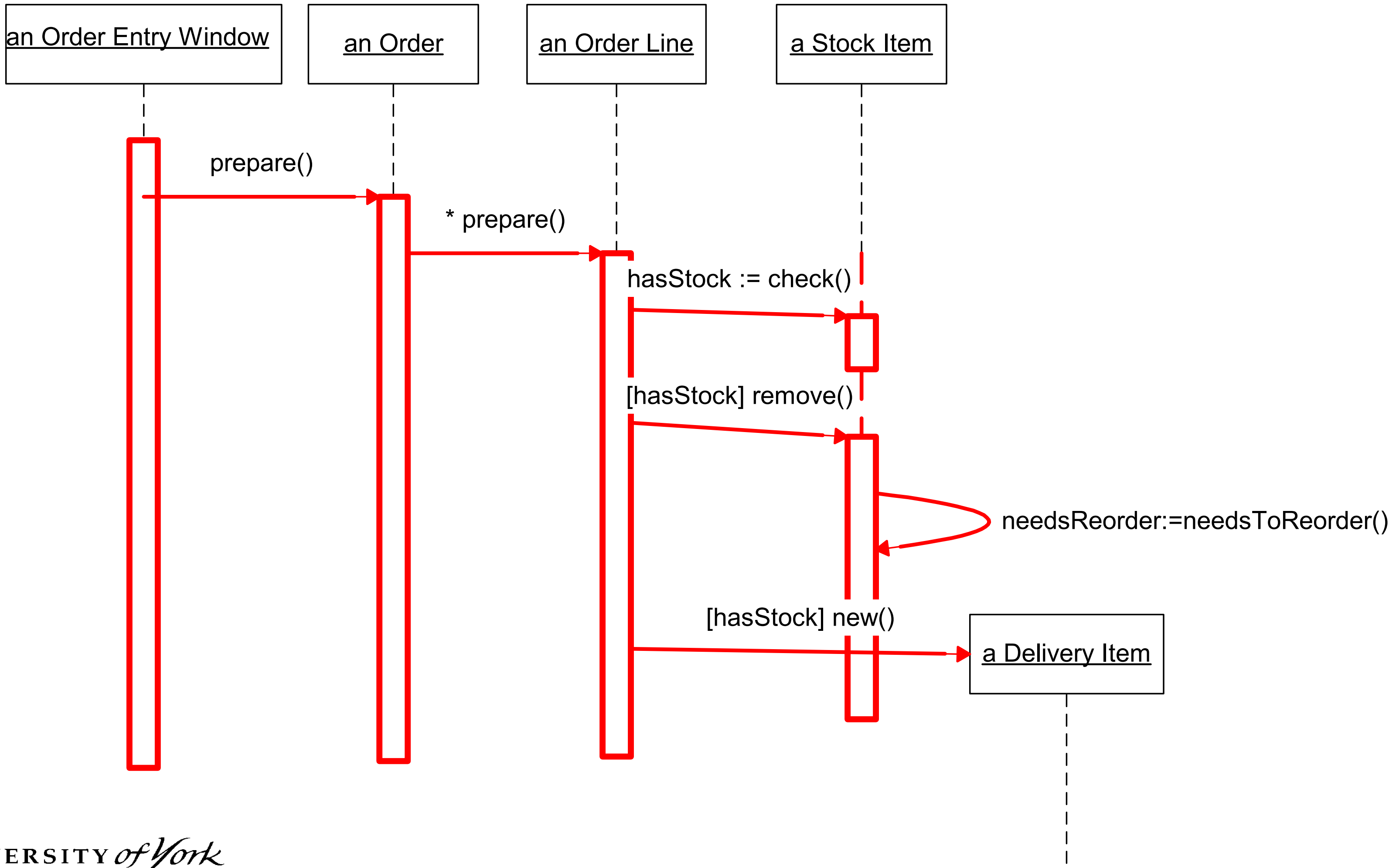
System Events and Operations

- Systems are used by **<u>actors</u>** (e.g., users, other systems)
- Systems are built to respond to **<u>events</u>**, external stimuli generated by actors.
  - When delineating the system borderline, it is often useful to determine which events are of interest to the system.
- **<u>Operations</u>** are executed in response to a system event.
  - The system has control over how it responds to events.
  - Operation execution in UML is usually represented via **messages**
- UML provides diagramming notations for depicting events and responses: interaction diagrams.

Interaction Diagrams

- Interaction diagrams describe how groups of objects collaborate in some behaviour.
- Typically, an interaction diagram captures the behaviour of a single scenario of use.
  - It shows a number of objects and the messages that are passed among these objects within the scenario.
- Two main types of interaction diagrams: sequence diagrams and collaboration diagrams.

Sequence Diagrams



Sequence Diagrams - Notation

- Objects are shown as boxes at the top of dashed vertical lifelines (actors can also be shown).
- Messages between objects are arrows; self-calls are permitted.
  - Conditions (guards) and iteration markers.
- To show when an object is active, an activation box is drawn; the sender is blocked.
- Return from call can be shown as well, but it usually clutters the diagram/confuses things.

Sequence Diagrams - External View of a System

The investor selects to open an existing portfolio.

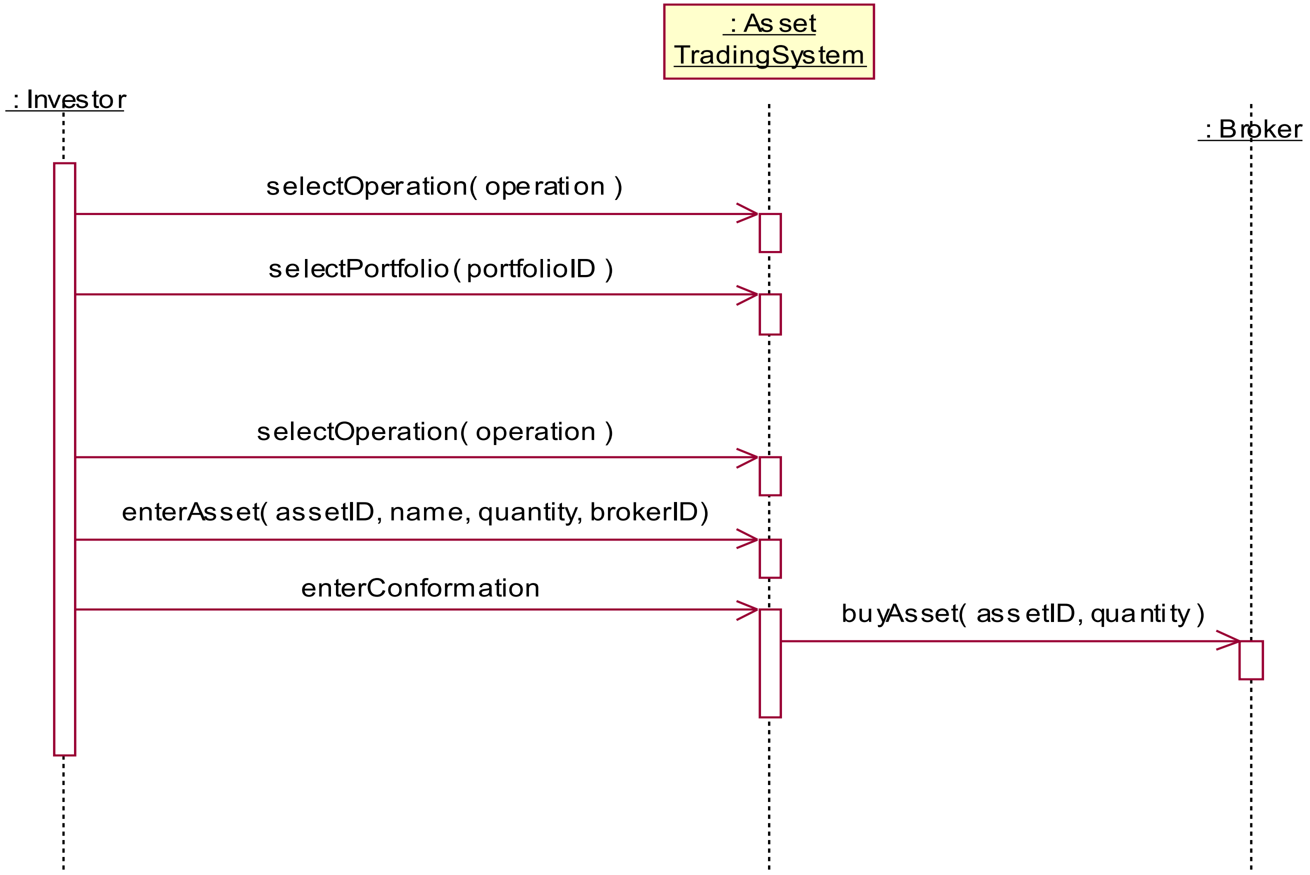
The investor selects the portfolio he wishes to open.

The investor selects the buy asset operation.

The investor enters the asset ID, name and quantity he wishes to purchase and the broker that will perform this transaction.

Before this transaction is Initiated the investor is prompted for conformation.

The system instructs the selected broker to buy the requested asset.



: Investor
: Asset TradingSystem
: Broker
selectOperation(operation)
selectPortfolio(portfolioID)
selectOperation(operation)
enterAsset(assetID, name, quantity, brokerID)
enterConformation
buyAsset(assetID, quantity)

THE UNIVERSITY of York

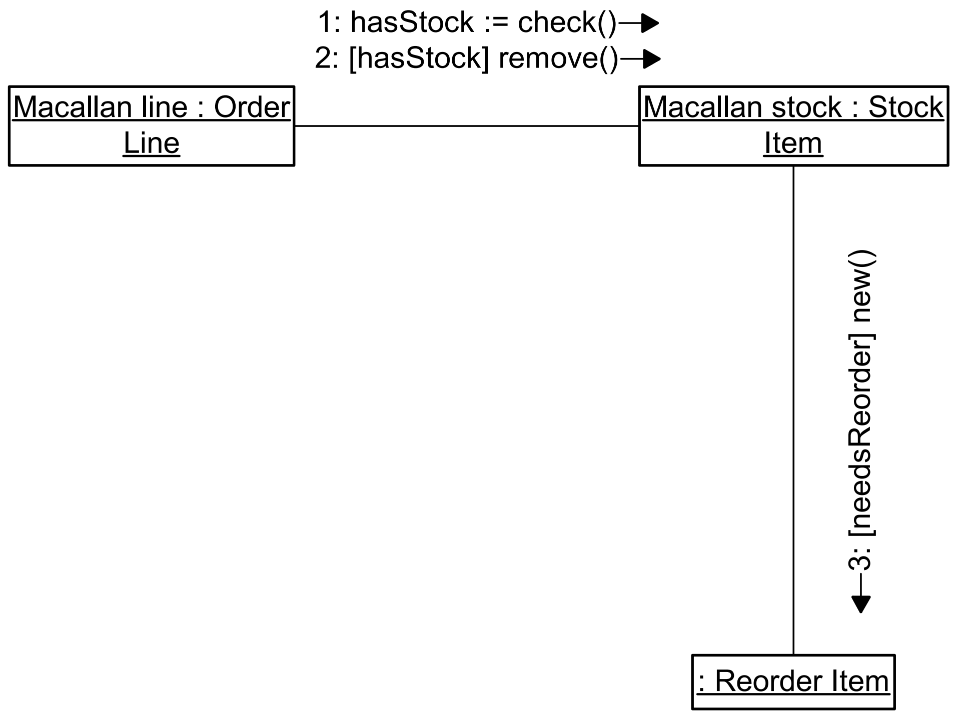
Sequence Diagram Advice

- Typically they are constructed after system services and scenarios of use have been determined.
  - They are good at showing collaborations among objects, not a precise definition of the behaviour.
  - Statecharts are better suited to the behaviour of a single object.
- Build sequence diagrams by identifying events.
  - Is the event generated by an actor or by the system itself?
- Focus on capturing the intent rather than the physical effect (i.e., don't use them to flowchart!)

Collaboration Diagrams

- Semantically equivalent to sequence diagrams.
  - Objects are shown as icons, and can be placed anywhere on the page/screen.
  - Sequence of message firings is shown by numbering the messages.
- Easier to depict object links and layout with collaboration diagrams; they're also more compact.
- Easier to see sequence with sequence diagrams.

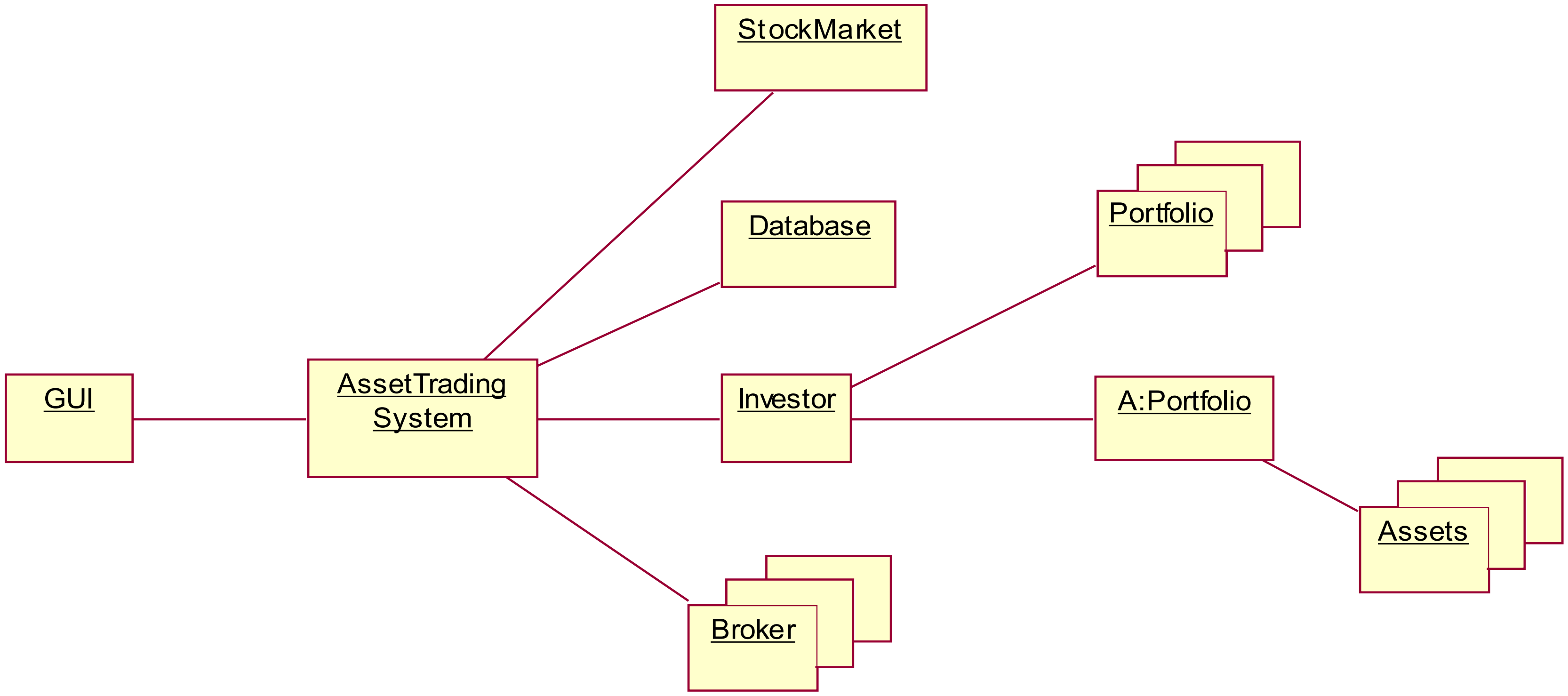
Example - Collaboration Diagram



Collaboration Diagram Notes

- Several numbering schemes for sequences are permitted.
  - Whole sequence numbers (as in example) is the simplest.
  - Decimal number sequence (e.g., 1.1, 1.2, 1.2.1, 1.2.2) can be used to indicate which operation calls another operation.
- Can show control information (guards, assignments) as in sequence diagrams.

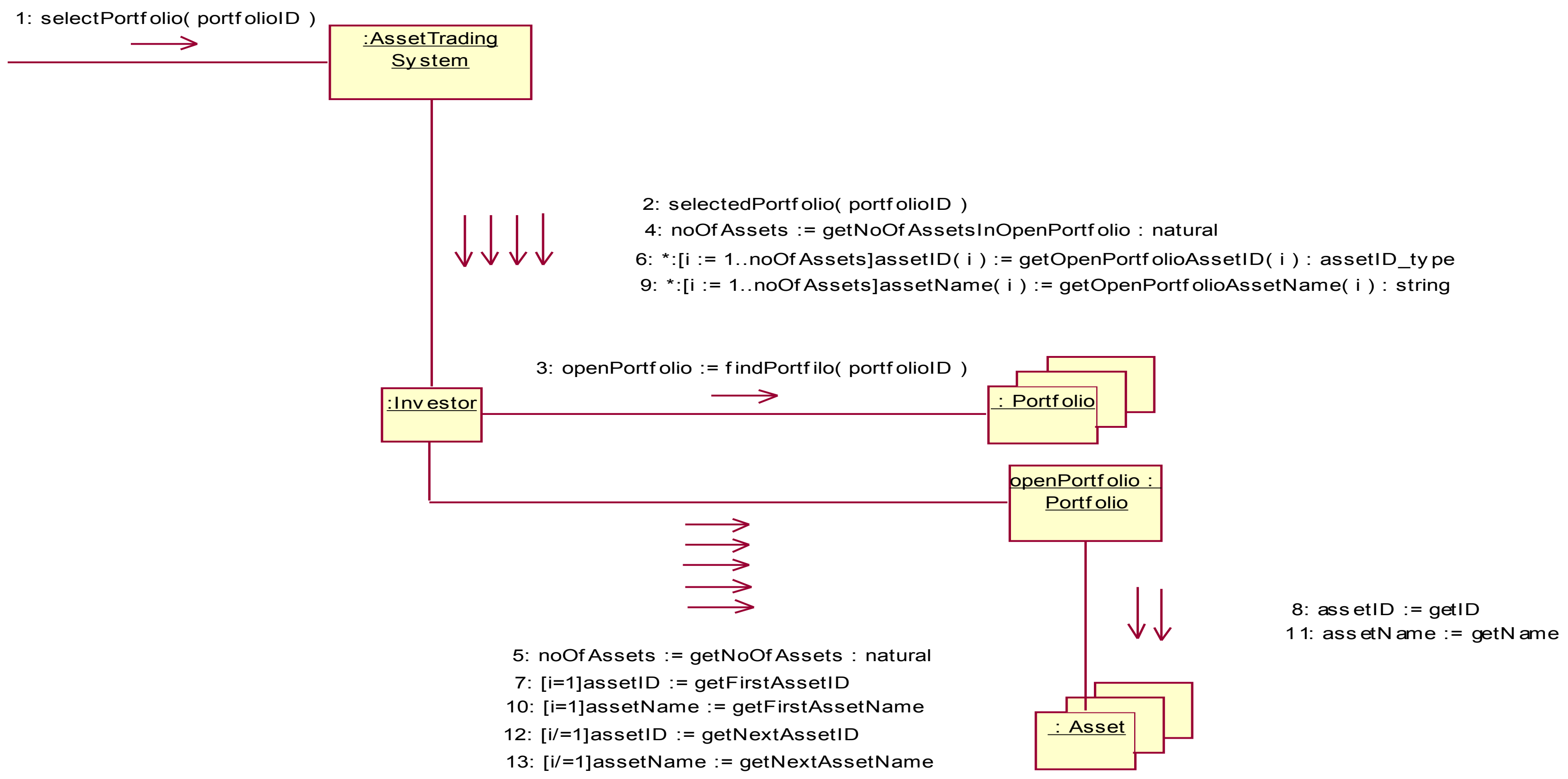
Collaboration Diagram
Initial Model without Messages



Object Diagrams

- A collaboration diagram without messages is also known as an object diagram.
  - The relationships between objects are called **<u>links.</u>**
- An object diagram must be a valid instantiation of a static class diagram.
  - Objects must have classes.
  - Links between objects must be instances of associations between classes.
- Use this as a quick consistency check.

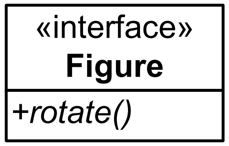
Collaboration Diagrams
More Complex Examples


1: selectPortfolio(portfolioID)
:AssetTrading System
2: selectedPortfolio(portfolioID)
4: noOfAssets := getNoOfAssetsInOpenPortfolio : natural
6: *:[i := 1..noOfAssets]assetID(i) := getOpenPortfolioAssetID(i) : assetID_type
9: *:[i := 1..noOfAssets]assetName(i) := getOpenPortfolioAssetName(i) : string
3: openPortfolio := findPortfilo(portfolioID)
:Investor
: Portfolio
openPortfolio : Portfolio
5: noOfAssets := getNoOfAssets : natural
7: [i=1]assetID := getFirstAssetID
10: [i=1]assetName := getFirstAssetName
12: [i/=1]assetID := getNextAssetID
13: [i/=1]assetName := getNextAssetName
8: assetID := getID
11: assetName := getName
: Asset

THE UNIVERSITY of York

Stereotypes

- The standard lightweight mechanism for extending UML.
  - If you need a modelling construct that isn't in UML but which is similar to something that is, use a stereotype.
  - Textual annotation of diagramming elements.
- Many standard stereotypes; can define own.
- **Example**: UML interface.



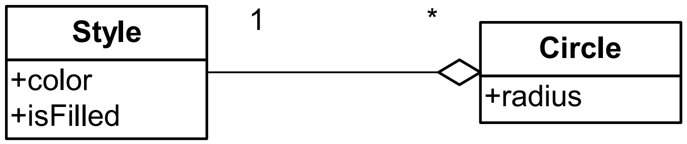
Some Built-in Stereotypes

- <<access>>: public contents of target package are accessible to the source package namespace.
- <<create>>: feature creates an instance of the attached classifier.
- <<friend>>: the source has access to the target of a dependency.
- <<instantiate>>: source classifier creates instances of the target classifier.
- <<invariant>>: constraint that must hold for the attached classifiers/relationships.

Aggregation and Composition

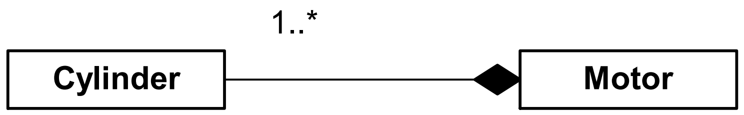
- Associations model general relationships between classes and objects.
  - At the implementation level, they can be defined in terms of reference types.
- Further relationships are provided with UML:
  - aggregation: "part-of"
  - composition: like aggregation but without sharing.
- Troublesome! Let's look at them more closely.

Aggregation



- An instance of Style is part-of zero or more instances of Circle.
- Style instances may be shared by many Circles.
- Semantically fuzzy: what's the difference between this and association with suitable multiplicity?
  - ... and how would you implement it in Java?
- Advice: if you can't be entirely precise about the distinctions between aggregation and other relationships, don't use it.

Composition



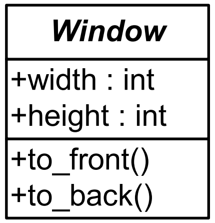
- A Motor is composed of one or more Cylinders.
  - Cylinders are "integral parts" of Motors.
  - The part objects (Cylinders) belong to only one whole.
  - The parts live and die with the whole.
- Sometimes called "value types" or "expanded types".

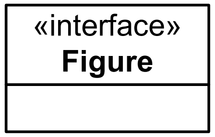
Using Composition & Aggregation

- Use association whenever you are in doubt.
  - Association can always be refined to more specific forms of relationship between modelling elements.
- Use aggregation judiciously - its semantics is extremely fuzzy.
- Associations with 1..1 multiplicity can be considered equivalent to compositions (since they support cascading deletes too).

Interfaces and Abstract Classes

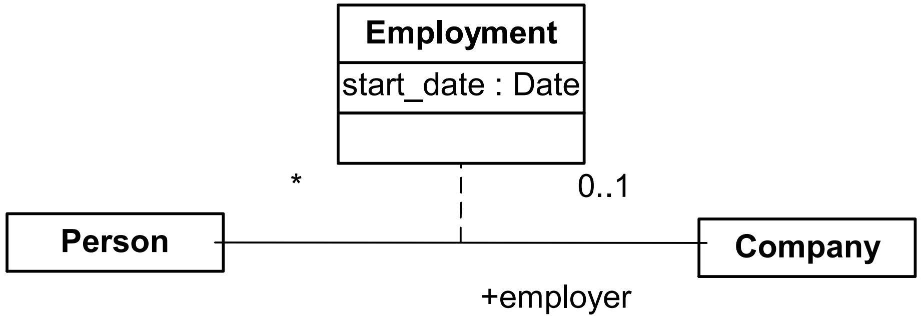
- A pure interface provides no implementation: it declares operations only.
  - Corresponds to `interface` in Java.
- An abstract class may have some implemented methods and fields, but not everything need be implemented.
  - Corresponds to virtual classes in C++.





Association Classes

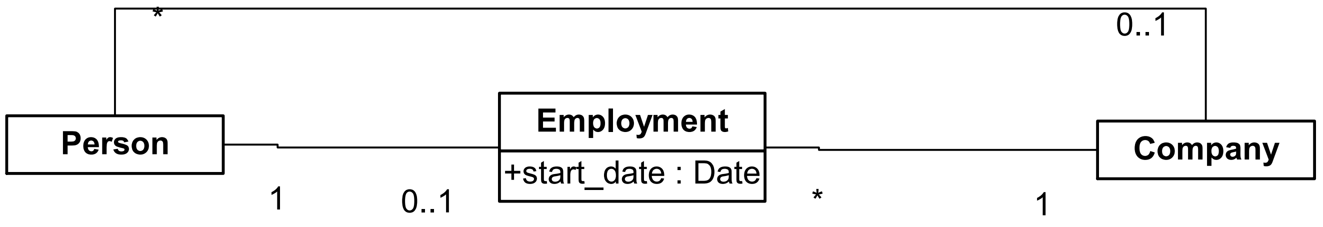
- Association classes can be used to add attributes, operations, and constraints to associations.



➢ Could add an attribute to Person indicating start date of employment, but this is really an attribute of the relationship between Person and Company.

Why Use Association Classes?

- The Employment information can also be expressed as follows.



➢ The association class implicitly includes the constraint that there is only one instance of the association class between any two participating Person and Company objects.

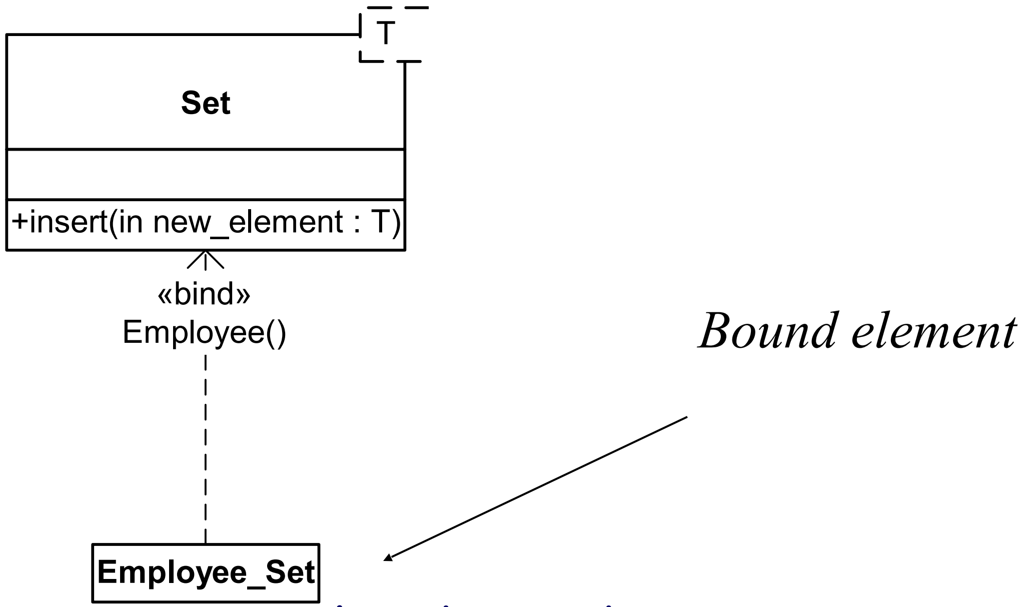
➢ This must otherwise be stated explicitly.

Parameterized Classes (Templates)

- The notion of **parameterized class** is present in Java 1.5 and is available in C++ and Eiffel.
- Lets you define collections of an arbitrary type.
  - Set[T], Sequence[T], Binary_Tree[T]
  - T is a type parameter that must be filled in in order to produce a type that can be used to instantiate objects.
- In C++:

```
class Set<T> {
   void insert(T new_element);
   void delete(T removed_element);
   ...
}
```

Templates in UML

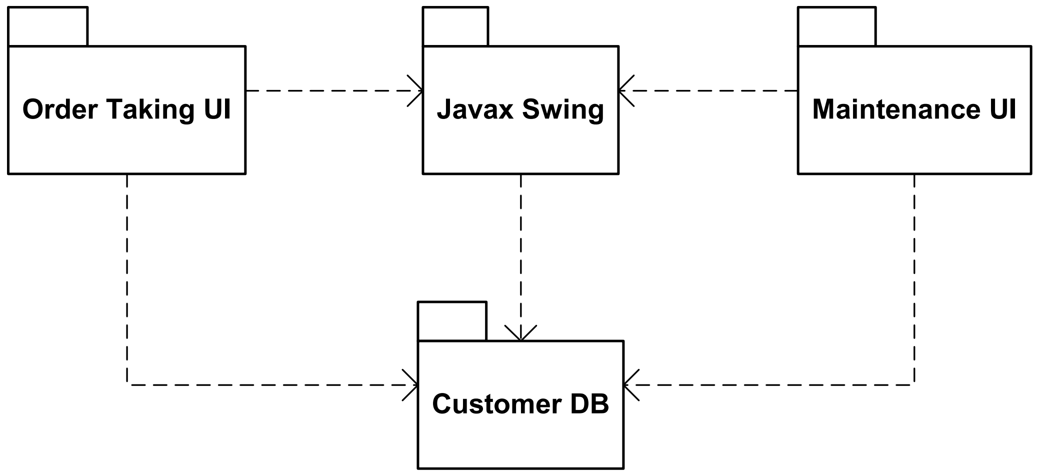


- <<bind>> is a stereotype on the dependency.
- Indicates that Employee_Set will conform to the interface of Set.
- You cannot add features to the bound element

Packages

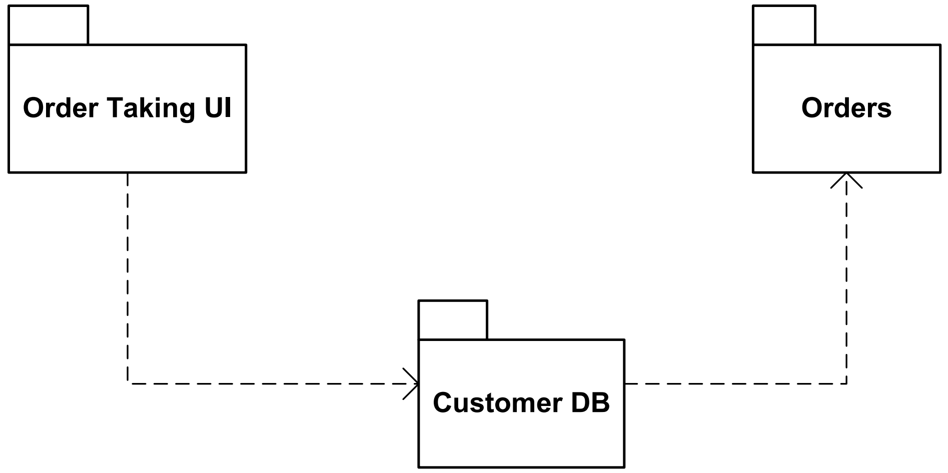
- Packages can be used to group any collection of modelling elements (classes, objects, other packages, etc.)
- Relationships between packages can be expressed in terms of **dependencies**.
    - A dependency exists between two elements if changes to one element (the supplier) may cause changes to the second element (the client).
    - Many types of dependencies in UML. Note that association and generalization are forms of dependencies.

Example - Packages



- If Customer DB changes, then Order Taking UI must be looked at to see if it needs to change.
- If packages contain classes, then a dependency between packages exists if there are dependencies between classes.
- UML 2.0 introduces package merge which is a useful way of composing multiple packages.

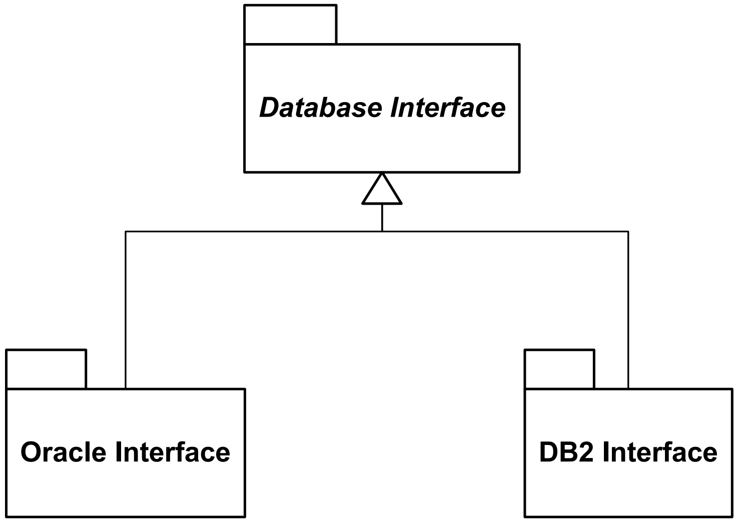
Example: Layered Architectures and Facades



- If Orders changes, Order Taking UI may be shielded from these changes by Customer DB.
- Similar to Java imports, but not C++ include.
- Reduce interfaces of packages by using info hiding and the Facade design pattern (delegating responsibility).

Package Generalization

- Generalization can be applied to packages.
- Defines a subtyping and a dependency relationship between packages.
  - Interface of child must be compatible with parent.
  - Related to concept of an MDA component



Designing a Class's Interface

- A critical step in constructing a class is to design its interface.
  - Particularly critical in multi-person projects, since the interface is used for communication and helps clarify responsibilities.
- Only concerned with client view.
- **Desirable characteristics:**
  - Simple, easy to learn, memorable, easy to change.
- Discuss several small issues in interface design.

Avoid Function Side-Effects

- In C++/Java, it's often standard practice to have functions with side-effects, e.g.,

```
int x;
int C::foo(int a,b) {
      x = a+b;
      return(a-b);
}
```

- **Avoid this** wherever possible - make your functions return values but not change state.
- Difficult to understand; lose referential transparency.

Example - `getint()` in C

- **`getint()`** reads a new input integer and returns its value; this has a side-effect (file pointer).
- If you call **`getint()`** twice you may get different results.
  - **`getint()+getint() != 2*getint()`** in general.
- Thus, we cannot reason about **`getint()`** as if it was a mathematical function (Leibniz).

<u>Function/Procedure Separation Principle</u>

Functions should not produce abstract side-effects.

How It Should Be Done

- Provide a class File.
- input is a variable of type File.
- To read new input:

  input.advance(); n = input.last_int;
- A File object contains attributes for buffering the last inputs.
- **Question:** is it ever reasonable to allow functions to have side effects?

How Many Method Arguments?

- To make classes more reusable, it is worth paying attention to the number of arguments given to methods.
- **Example:** FORTRAN non-linear ODE solver routine has 19 arguments: 4 var parameters, 3 arrays, 6 functions (each with arguments).
- Non-linear solvers in many C++ math libraries have **zero** arguments. How?

Operands and Options

- An argument to a routine is an **operand** or an **option**.
  - An option is an argument for which a default value could have been found if the client hadn't specified it.
  - An operand is needed data.
- As classes evolve, operands tend to stay the same, but options are often added or removed.

Method arguments should be operands only.

- Options to methods are set in calls to separate methods:

```
document.set_print_size("A4");
document.set_colour(); document.print();
```

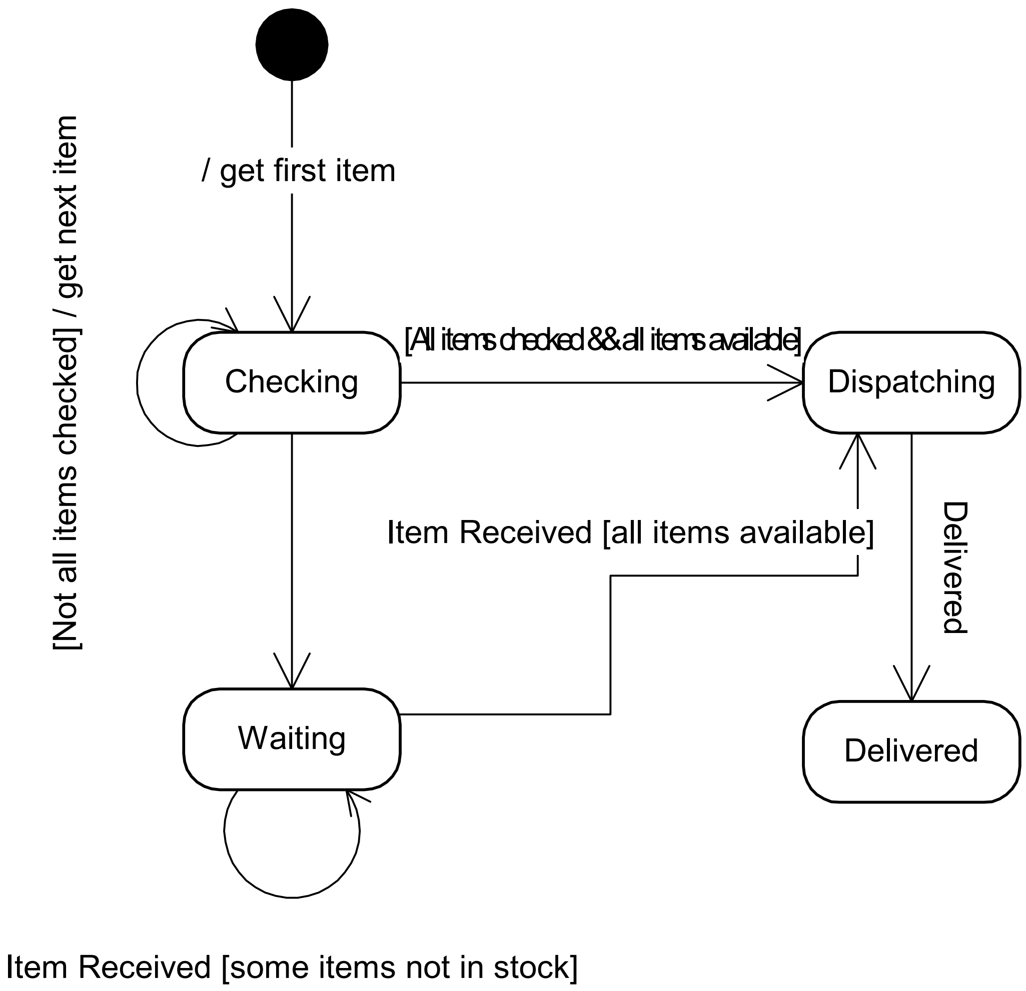
Class Size

- How do we measure the size of a class: #LOC, number of methods, number of inherited methods?
- Does the size of a class matter?
- Paul Johnson says
  - "Class designers are often tempted to include lots of features ... The result is an interface where the few commonly used features are lost in a long list of strange routines."
- Not always the case - if a method is conceptually relevant to a class, **and it does not duplicate an existing method**, then it is reasonable to add it.
- Example: a Complex number class with a + operator as well as an add method; fills different needs.

Statecharts

- Class diagrams and packages describe static structure of a system.
- Interaction diagrams describe the behaviour of a collaboration.
- How about describing the behaviour of a single object when it reacts to messages?
  - constraint language like OCL (which we'll see soon)
  - statecharts
- Statecharts describe all possible states that an object can get in to, and how the object responds to events.

Example: Statechart


/ get first item
[Not all items checked] / get next item
Checking
[All items checked && all items available]
Dispatching
Item Received [all items available]
Delivered
Waiting
Delivered
Item Received [some items not in stock]

Statechart Notation

- Syntax for a transition is

  `Event [Guard] / Action`

- Actions are associated with transitions; they are short, uninterruptible processes.
- Activities are associated with states, and may be interrupted.
- A guarded transition occurs only if the condition evaluates true; only one transition can be taken.
- When in a state with an Event, a wait takes place until the event occurs.

When to Use Statecharts

- They are good at describing the behaviour of an object across several scenarios of use.
- They are not good at describing behaviour that involves a number of collaborating objects (use interaction diagrams for this).
  - Not usually worthwhile to draw a statechart for every class in the system.
  - Use them only for those classes that exhibit interesting behaviour.
  - e.g., UI and control objects.

Use Cases - Review

- Use cases are commonly described as telling a story – of how a user carries out a task.
- A use case is a document that describes the sequence of events of an actor using a system to complete a scenario.
- An actor is external to the system, i.e., a human operator or another system.
- A scenario describes a complete sequence of events, actions and transactions required to produce or complete something of value.

Scenarios

- A scenario is a sequence of steps describing an interaction between an actor and a system.
- Example:

  The customer browses the online catalogue and adds desired items to their basket. When the customer wishes to pay, the customer specifies the mode of shipping and their credit card information, and confirms the sale. The system validates the credit card authorisations, and confirms the sale via an immediate follow-up e-mail.

- This is just one possible scenario; failure of the credit card authorisation would be a separate scenario.

Identifying and Applying Use Cases

- Use cases can interact with any number of actors.
- Discover use cases by first identifying actors. For each actor, consider the scenarios that they may initiate.
- **Common Error**: representing individual system operations as a use-case, e.g., create transaction, destroy record.
- Use cases represent services that may be implemented by multiple operations
  - Usually they are a relatively large process.

Example – Use Case Text
Buy a Product – General Case

1. Customer browses catalogue and selects items to buy, placing them in the shopping cart.
2. Customer goes to check out.
3. Customer fills in desired shipping info (address, type of delivery).
4. System presents full price.
5. Customer provides credit card information.
6. System authorises purchase.
7. System confirms sale via e-mail.

Example Use Case Text
Authorisation Failure

- The authorisation of the credit card may fail (over credit limit, expired card, system down).
- A use case is needed for this scenario.
  - At step 6, the system fails to authorise the credit card purchase.
  - Allow the customer to re-enter credit card information and try again.
- Here, we are extending an existing use case (buy a product) with a new use case.

Example Use Case Text
Alternative – Regular Customer

- We may want to support returning customers, allowing customers to save their credit card or address information in the system.
- A use case is needed for handling this type of interaction.

3.a) System displays current shipping, pricing information and last four digits of previously entered credit card.

b) Customer may accept or override these data.

Return to primary scenario at step 6.

How Much Detail is Needed?

- Clearly, you can spend a lot of time writing out use cases in tedious detail.
- The amount of detail that you need depends on the risk inherent in the use case.
- **Advice**:
  - during elaboration, go into detail on only a few (critical) use cases
  - as you iterate through development, you'll add more detail as it becomes necessary to implement each use case.

Types of Use Cases

- Use cases come in several flavours. The distinctions aren't always useful.
- Use cases can be categorized as :
  - Primary: describes major system scenarios
  - Secondary: describes minor or rare scenarios
  - Optional: describes scenarios that may not be implemented.
- Very useful for allocation of resources and timetabling.

Asset Trading System

Actor	Description
Investor	A person controlling portfolios of assets.
Database	A system that maintains a permanent record of investor portfolios
Broker	A person or system that allows an investor to buy or sell assets.
Stock market	A system that allows the investor to examine assets that are currently being traded.

Use case	Description
Access Database	Load / save portfolio data to / from persistent storage
Sell Asset	Sell an asset to a broker
Buy Asset	Buy an set from a broker
Browse Portfolio	Browse assets in a portfolio
Browse Stock Market	Browse stock market listings of asset details
Calculate Portfolios value	Calculate the value of the assets stored in a portfolio using the current stock market pricing.

Example Use Case
Buy Asset

Use case :	Buy Asset
Actors :	Investor, Broker
Type :	Primary

Description :

The investor selects an asset to be bought. This information is passed to a specific broker, who performs this transaction. When complete the purchased asset is added to the investor's portfolio and his current account debited by the cost of the assets plus the brokers fee.

Use Case (Expanded with Detail)
Buy Asset

- Use case : Buy Asset
- Actors : Investor, Broker
- Purpose : Buy assets to add to a portfolio
- Type : Primary, Essential
- Description : The investor selects an asset to be bought. This information is passed to a specific broker, who performs this transaction. When this transaction is complete the selected asset is added to the investors portfolio and his current account debited by the cost of the assets plus the brokers fee.
- Cross Reference : System function 1.

Use Case Course of Events
Buy Asset

- It is sometimes useful to specify a very detailed sequence of events for a scenario.
- An event is something interesting with respect to a system.
- Note similarity to event-driven programming model.
- Example:
  - <u>Actor action</u>: the investor selects the "buy asset" operation.
  - <u>System response</u>:
    - check investor's current balance and portfolio.
    - if investor's current balance > credit limit then see section "Credit Limit Reached"
    - if investor's portfolio has 10 assets then see section "Asset Limit Reached"

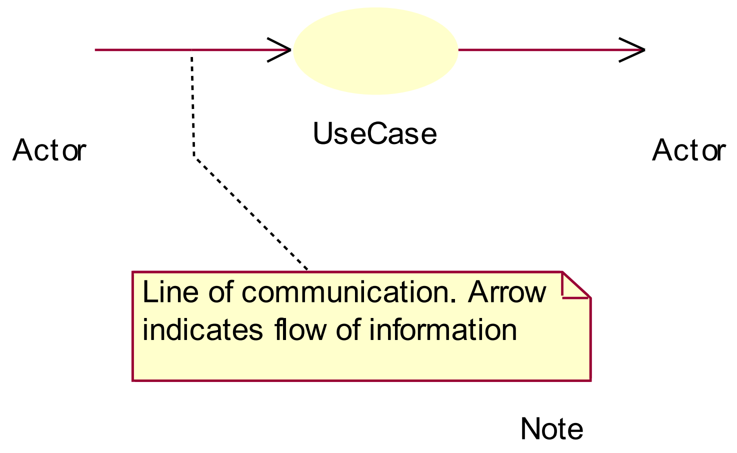
Example Use Case
Calculate Portfolio Value

Use case :	Calculate Portfolio Value
Actors :	Investor, Stock Market
Type :	Secondary

Description : The investor selects a portfolio to be valued. For each asset contained within this portfolio the system retrieves its current traded value from the stock market. The value of the assets contained within this portfolio is now calculated.

Use Case Diagrams

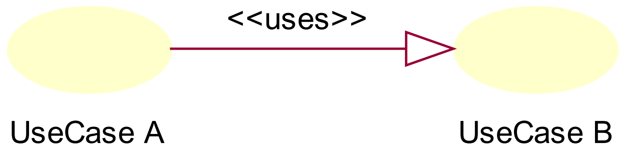
- These illustrate the relationships that exist between a set of use cases and their actors.



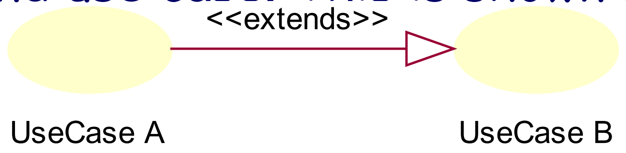
- Their purpose is to allow a quick understanding of how external actors interact with the system.

Use Case Diagrams
Relationships

- If one use case initiates or duplicates the behavior of another use case it is said to '**use**' the second use case. This is shown as:

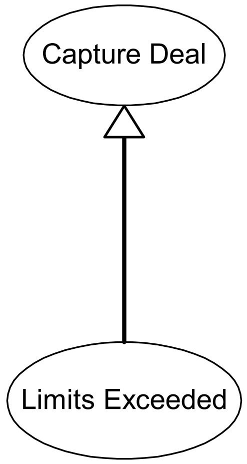


- If one use case alters the behavior of another use case it is said to '**extend**' the second use case. This is shown as:



Use Case Diagrams
Relationships

- If one use case is similar to another, but does a little bit more, you can apply **generalization.**



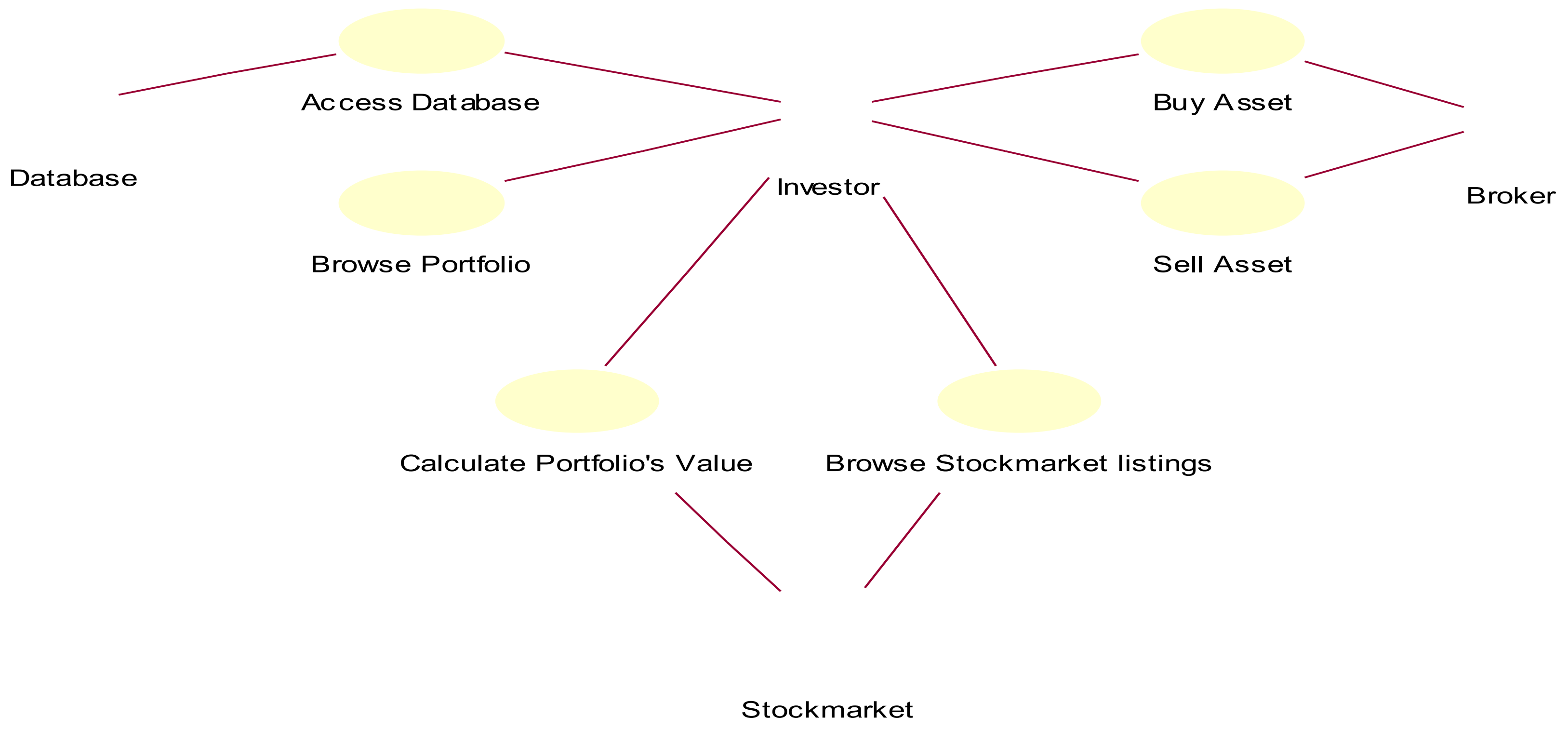
What is the difference between <<extend>> and generalization?

With <<extend>>, the base use case must specify extension points.

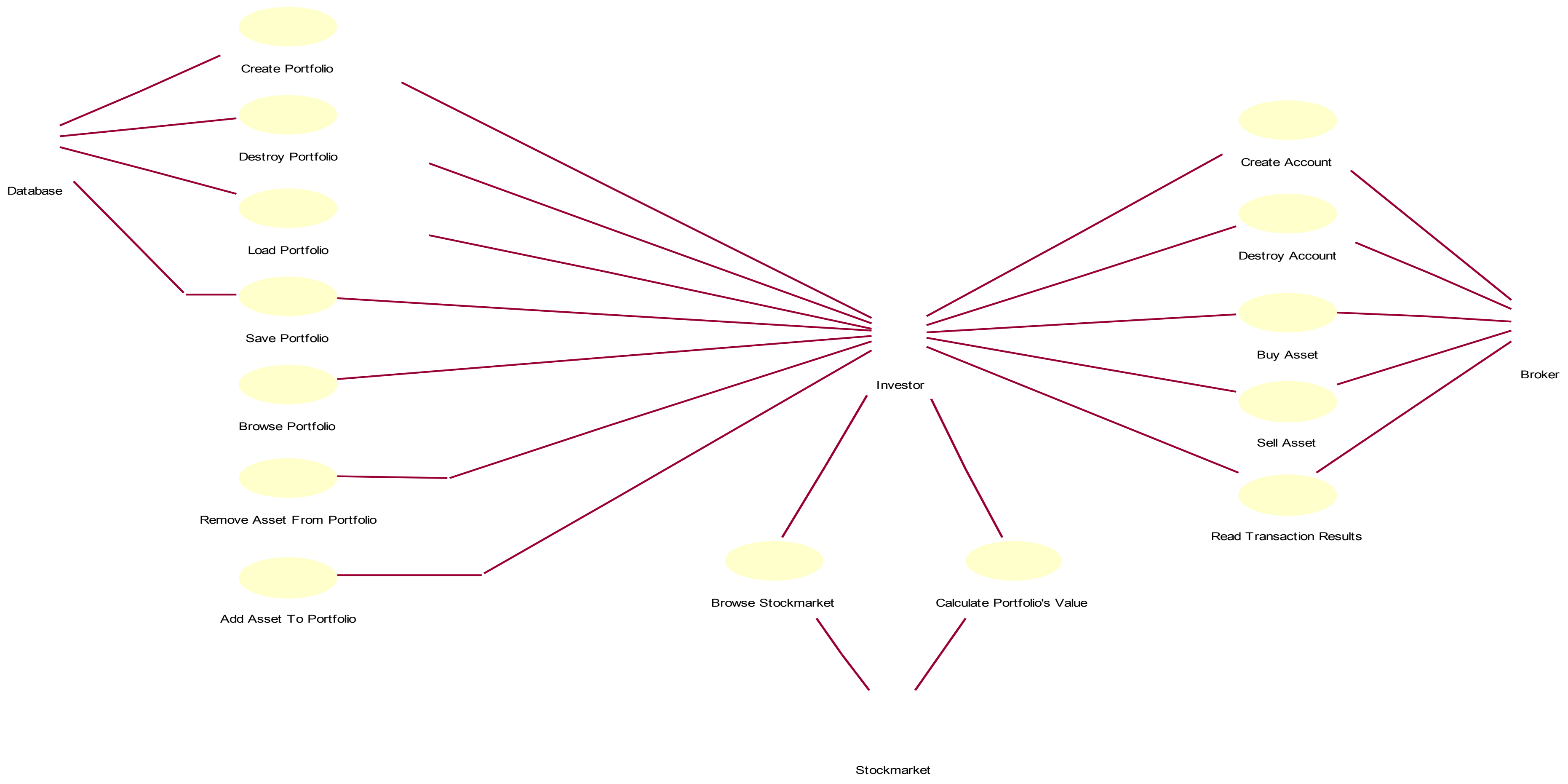
Use Case Relationships – Some Guidelines

- It's usually easier to think about normal cases first, and worry about variations afterwards.
- Use **<<use>>** when you are repeating yourself in two separate use-cases.
- Use generalization when you are describing a variation on a behaviour that you want to capture informally.
- Use **<<extend>>** when you want to more accurately capture a variation on behaviour.

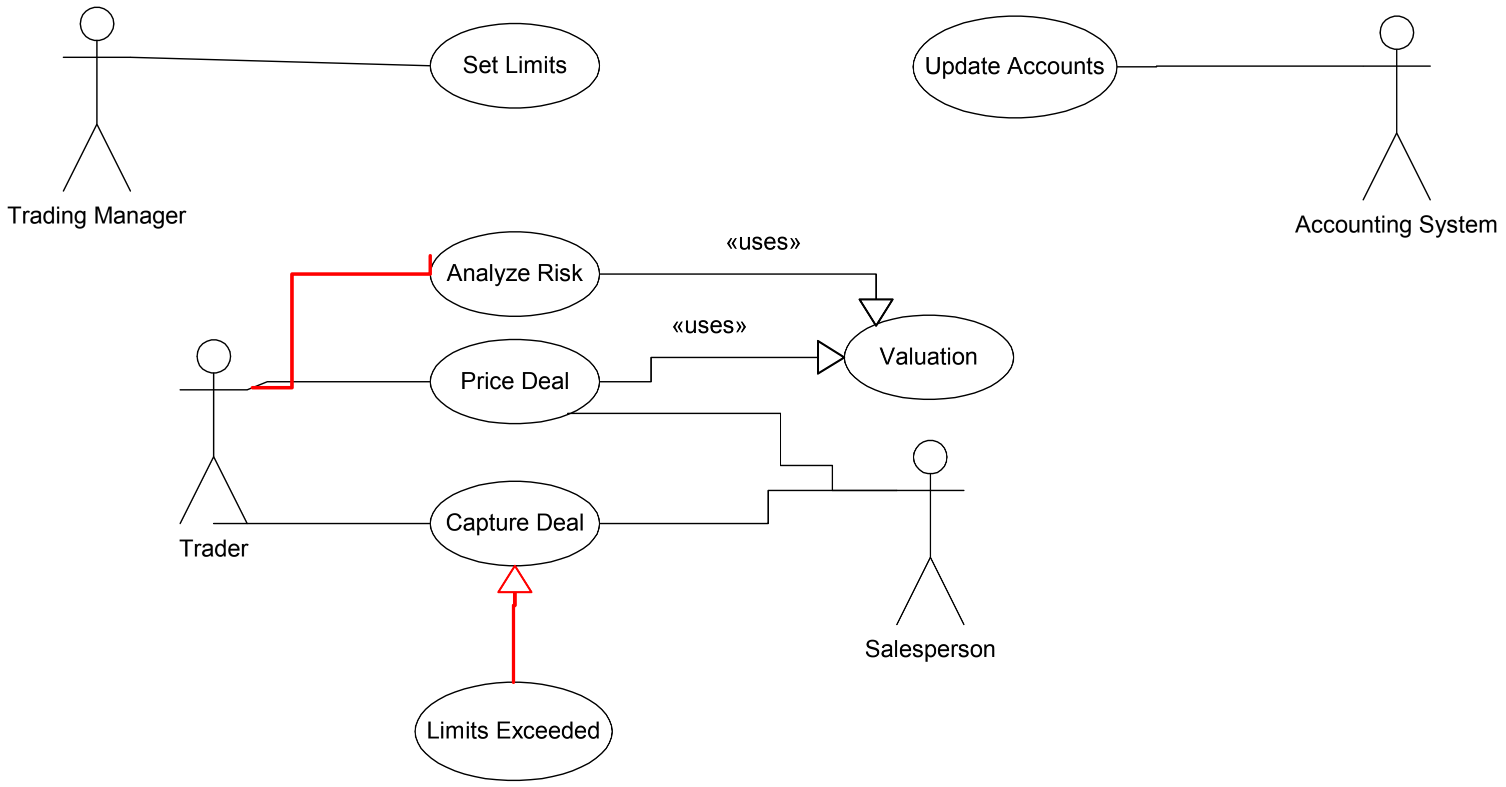
Example Use Case Diagram (…without stereotypes…)



Example Use Case Diagram
Is this really helpful?



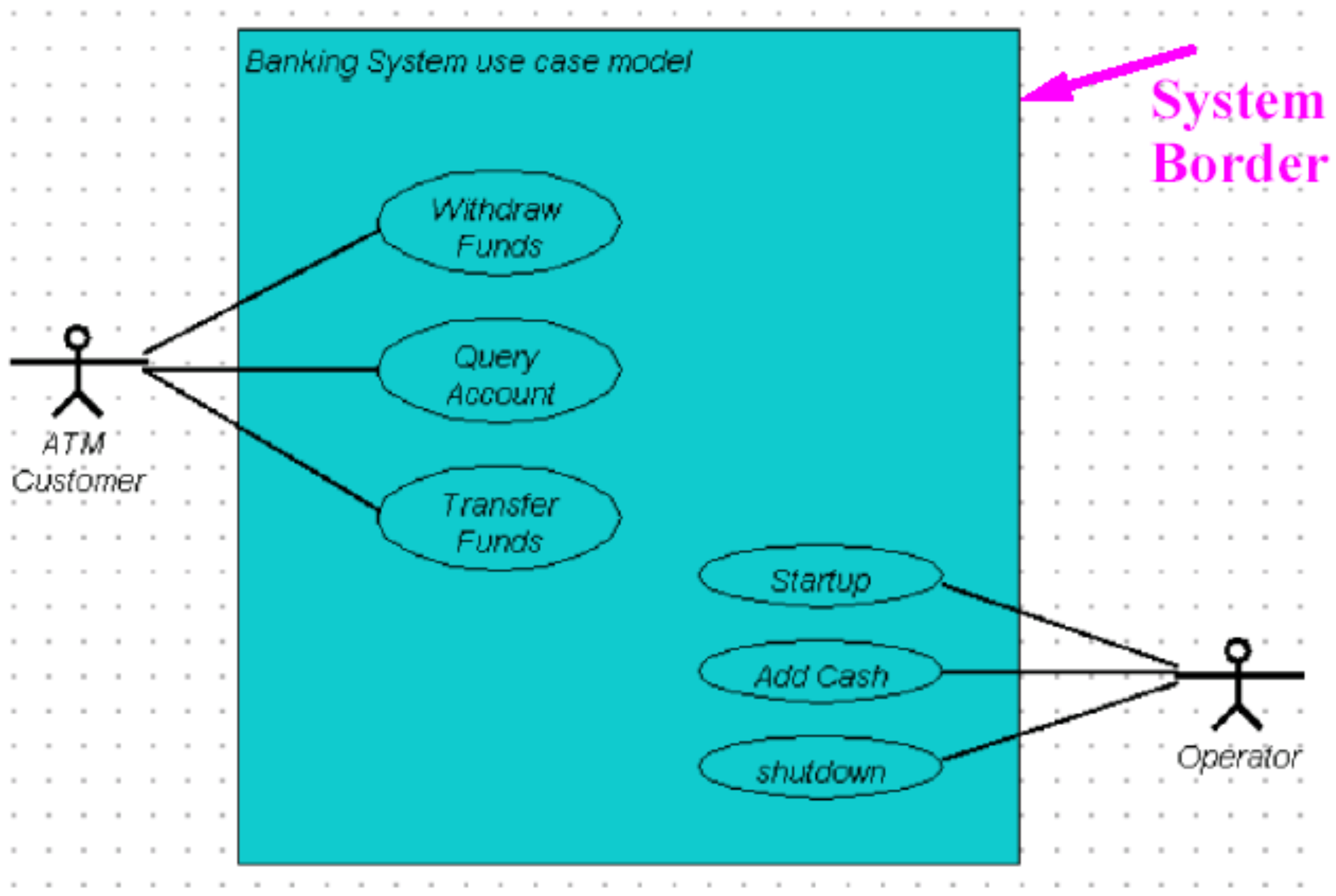
Use Diagram
Another Example



Requirements for an ABM

- A bank has several ABMs connected via a WAN to a central bank server.
- Each ABM has a card reader, cash dispenser, keyboard, display, and a receipt printer.
- A customer can withdraw funds from chequing or savings accounts, query their balance, or transfer funds from one account to another.
- The customer PIN needs to be validated against the server. The card is confiscated if a third validation attempt fails. Cards that have been reported lost or stolen are also confiscated.
- An ABM operator may start/shutdown the machine to replenish cash.

Use Case Diagram



Detailed Use Case for Withdraw Funds

- Summary: customer withdraws a specific amount of funds from a valid bank account.
- Actors: ABM customer.
- Dependencies: include “Validate PIN use case”.
- Precondition: ABM is idle displaying a welcome message.
- Detailed Description: …

Detailed Description of Withdraw Funds Use Case

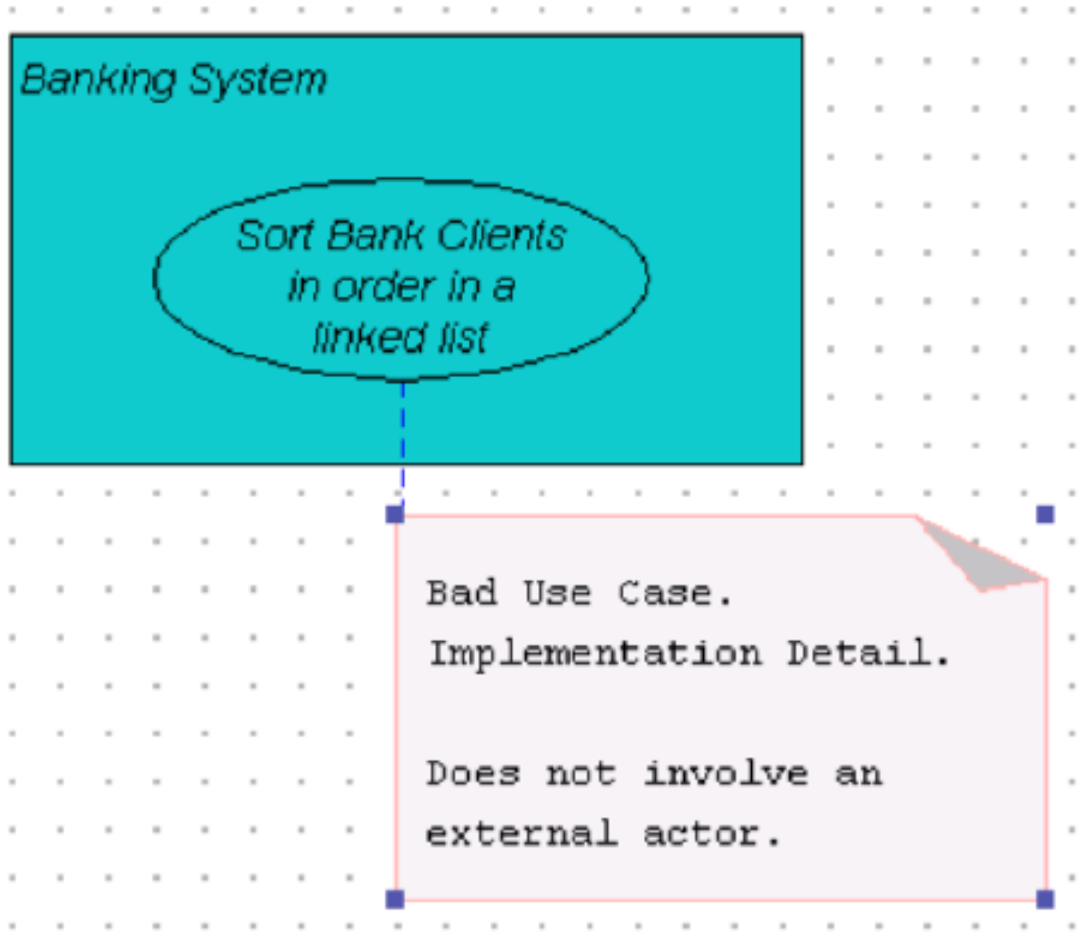
1. Include "Validate PIN" use case
2. Customer selects Withdraw Funds, enters the amount, and selects the appropriate account.
3. System checks whether customer has enough funds in the account.
4. …
7. System ejects card.
8. System displays welcome message.

- What about alternative cases?

Alternative Cases & Postcondition

- If the system determines that the account is invalid, it displays an error message and ejects the card.
- If the ABM is out of funds the system displays an apology.
- Others?
- What about a postcondition for the use case?
  - Interesting! Need a postcondition for different parts of the use case, e.g., successful withdrawal, cancelled transaction, completion of the scenario, etc.

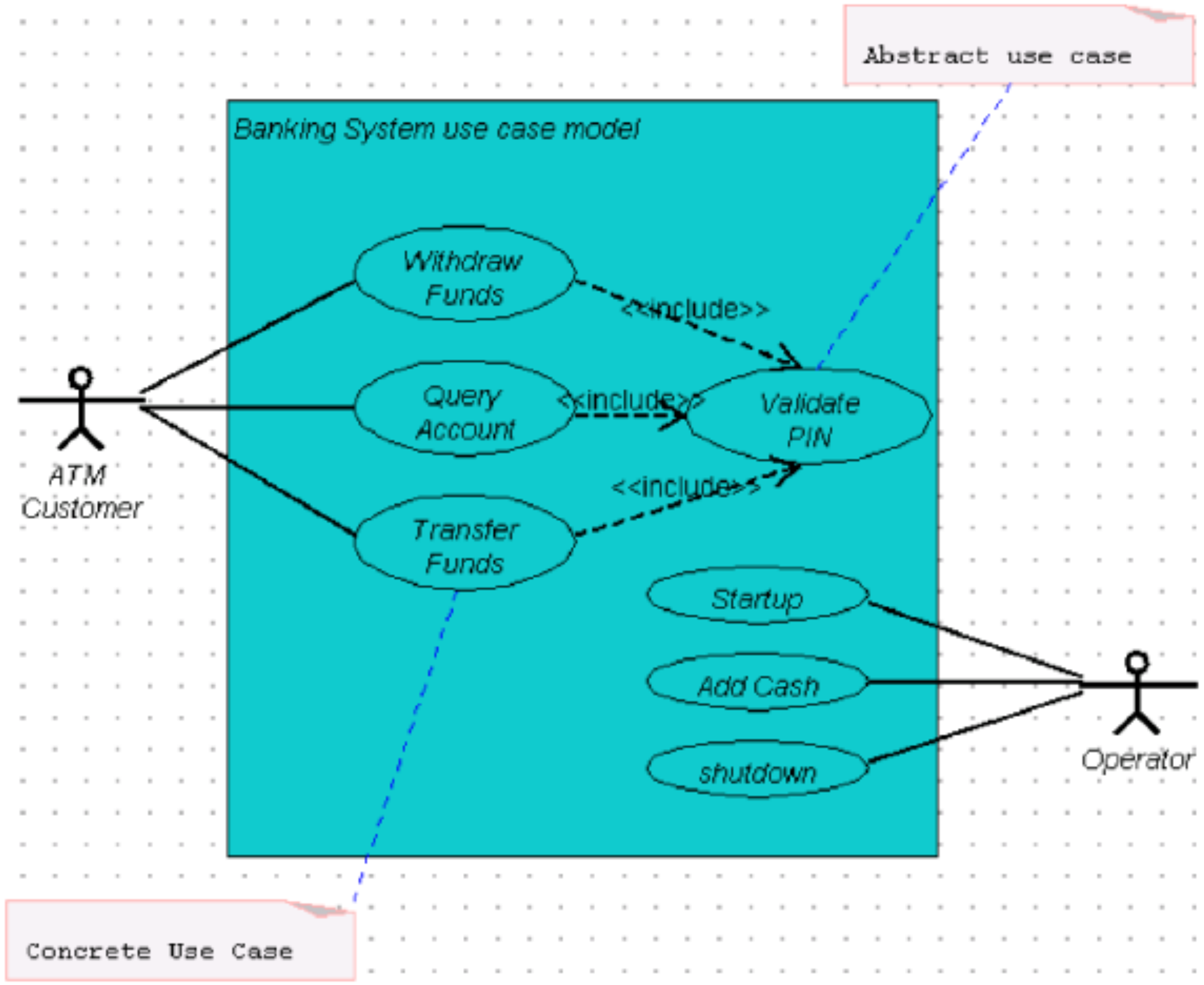
A Bad Use Case



```
Bad Use Case.
Implementation Detail.

Does not involve an
external actor.
```

Factoring Use Cases

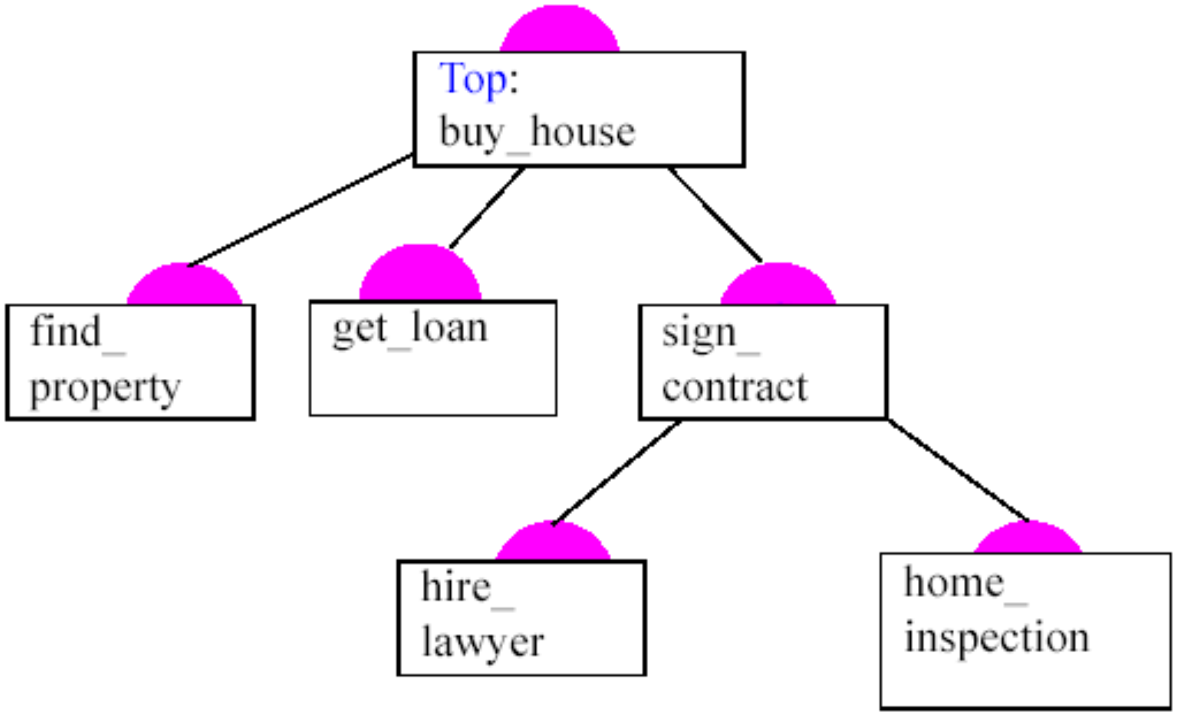


Risks Using Use Cases

- Use cases emphasise ordering.
  - Take the use case “Place An Order”.
  - A credit card is validated, database is updated, and a confirmation number is issued.
- Ordering sequences of actions at the requirements stage may be premature for OO development - leads to fragility.
- In OO we don’t focus on functions, i.e., “do a then b”. Instead we find abstractions.
- Use cases make it easy to get caught in a top-down (functional) development style.

Example: Buying a House

- The structure of the use case looks like this:



Sequencing Too Soon!

- The OO solution will be a single class:

```
class PURCHASE_PROPERTY {
  boolean property_found();
  boolean loan_approved();
  /** *@post property_found(); **/
  void find_property();
  /** *@post loan_approved(); **/
  void get_loan();
  /** *@pre property_found() && loan_approved() **/
  void sign_contract();
}
```

- Finding a property and getting a loan can happen concurrently.

Use Case Warning!

- Use cases are reasonable for late requirements (i.e., system boundary).
- Except with an experienced development team, use cases can be dangerous for OO decomposition.
- Use cases can be converted into interaction diagrams.
    - ... which can in turn be converted into test cases, which can be used to validate system designs.