5COSC001W - OBJECT ORIENTED DEVELOPMENT

All you need to know about UML

Dr Dimitris C. Dracopoulos

email: d.dracopoulos@westminster.ac.uk

1 What is UML?

A collection of techniques (diagrams) used for object oriented software design.

- It unifies the methods of Booch, Rumbaugh and Jacobson, known as the *three amigos* in the software engineering community.
- It is a modelling language not a method.

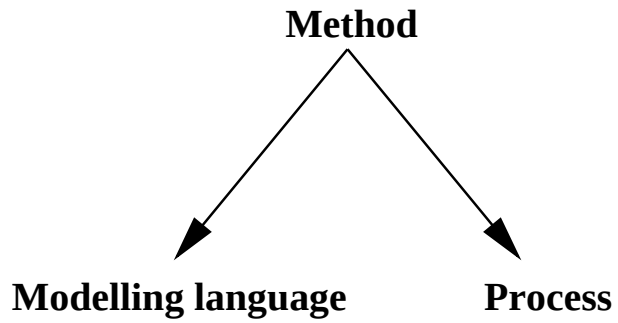

Figure 1: A design method consists of a modelling language and a process.

As shown in Figure 1 a method consists of:

- A *modelling language*: notation that methods use to express designs.
- A *process*: What steps to take in doing a design.

2 UML techniques

The most well known techniques currently used in UML are (but UML still evolves):

- Use case diagrams
- Class diagrams
- Object diagrams
- Interaction diagrams
  - Sequence diagrams
  - Collaboration diagrams
- Statechart diagrams
- Activity diagrams
- Physical diagrams

3 Use Case Diagrams

- Describes *what* we are building (helps to build the right system).
- It is a snapshot of one aspect of the system. The sum of all use cases is the external picture of the system.

A use case diagram consists of:

- *Actors:* an actor is a role that a user plays in respect to the system.
- *Use cases:* a use case is a scenario for a single task or goal.
- *Communications:* the associations between the actors and the use cases.

4 An example of a use case diagram

A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot. The patient visits the doctor and then he pays the bill. Figure 2 shows the use case diagram for this scenario.

5 How to draw a use case diagram

Use cases identify key features in the system that will reveal some of the fundamental classes which will be used in the implementation.

Ask the questions:

- Who will use this system?
- What can those actors do with the system?

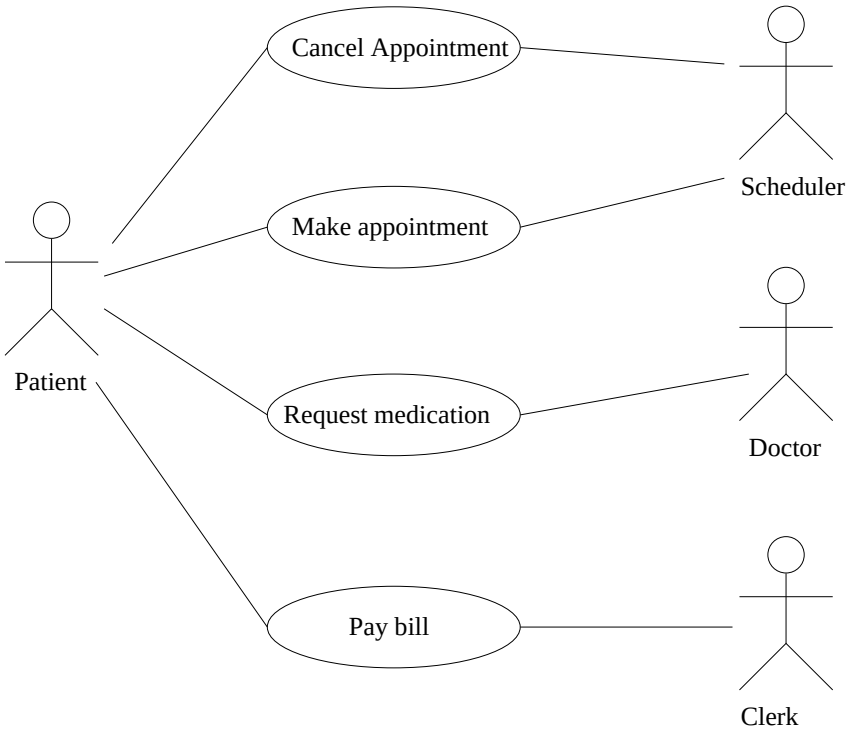

Figure 2: An example of a use case diagram.

- How does *this* actor do *that* with this system?
- How else might this work if someone else were doing this, or if the same actor had a different objective? (to reveal variations)
- What problems might happen while doing this with the system? (to reveal exceptions)

6 Class Diagrams

A class diagram describes the types of objects in the system and the static relationships that exist among them.

There are two main kinds of static relationships:

- *Associations:* For example, a customer may rent a number of videos.
- *Generalisation:* For example a manager is a kind of employee.

A navigability arrow on an association shows which direction the association can be traversed or queried. For example in Figure 3 an `OrderDetail` can be queried about its `Item` but not the other way around. Associations without arrows are assumed either bi-directional or without known navigability. A specific project should state which of the two it assumes.

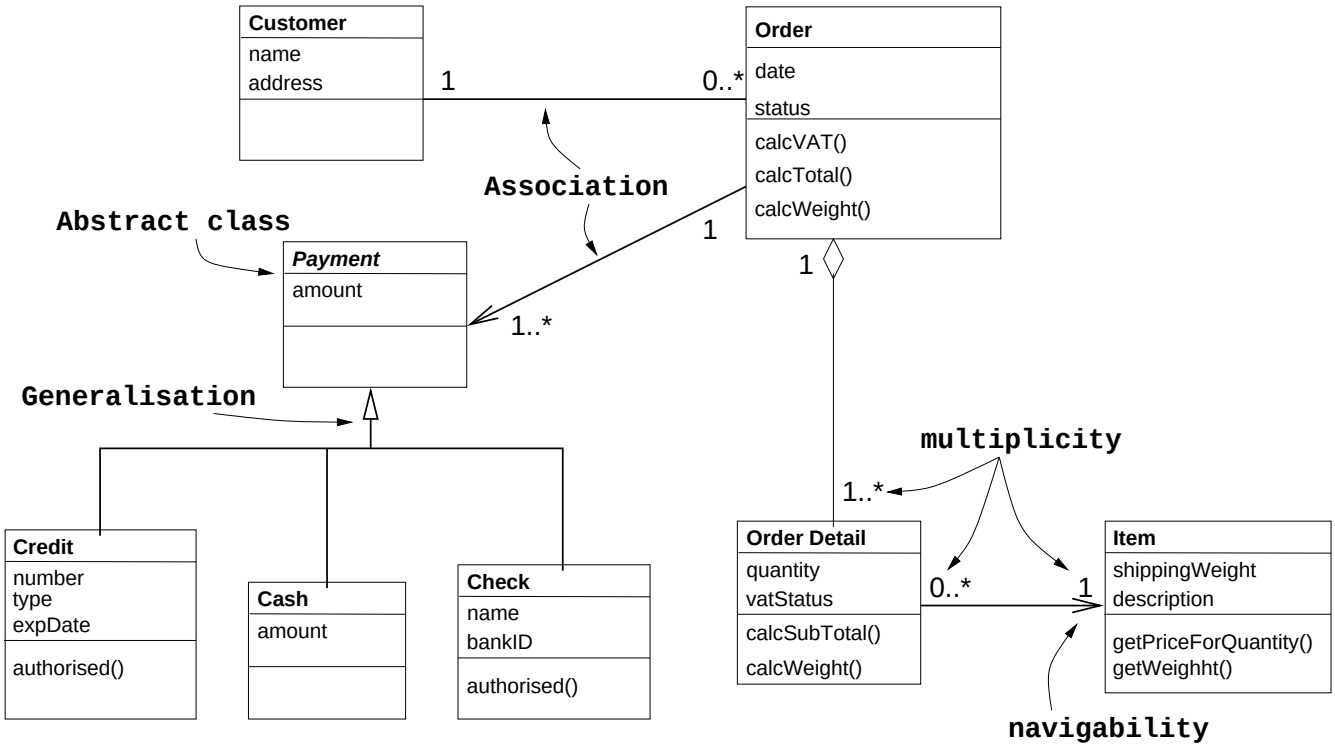

Figure 3: An example of a class diagram

7 Generalisation and Inheritance

```
class Employee {
   private int employeeNo;
   private double salary;
}
class Manager extends Employee {

}
```

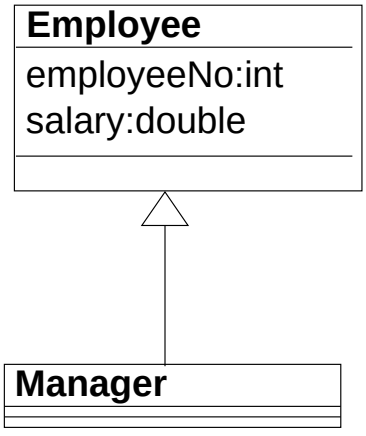

Figure 4: Representing inheritance in UML.

Generality is based on the *is-a* relationship. `Manager` *is a* `Employee`.

A subclass inherits all the methods and fields of the superclass but may override inherited methods, i.e. provide different implementations which can change the functionality of the inherited method, but not its interface!

Another way of thinking of generalisation involves the principle of *substitutability*. I should be able to substitute a `Manager` within any code that requires an `Employee`. This means that if some code assumes a `Customer` type then any subtype of `Customer` can be used instead. This is true for subtypes of `Customer` that did not even exist at the time that the initial code was written. This is one of the fundamental issues in software reusability and extensibility.

8 The Class in a class diagram

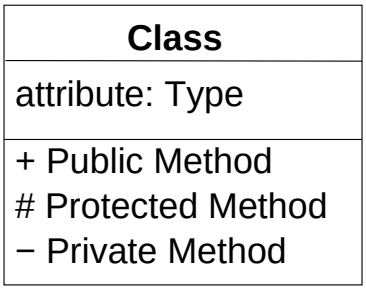

Figure 5: The representation of a class in UML.

The full UML syntax for operations is:

visibility name(parameter-list): return-type{*property-string*}

where

- *visibility* is: `+` (public), `#` (protected), or `-` (private).
- *name* is a string.
- *parameter-list* contains comma-separated parameters similar to function parameters.
- *return-type* is the type(s) of the return value(s). Notice that more than one return value is allowed in UML.
- *property-string* indicates property values that apply to the given operation.

For example: *balanceOn(date: Date): Money*

9 Interfaces and Abstract Classes

- Realisation is similar to generalisation. It indicates that a class implements an interface.

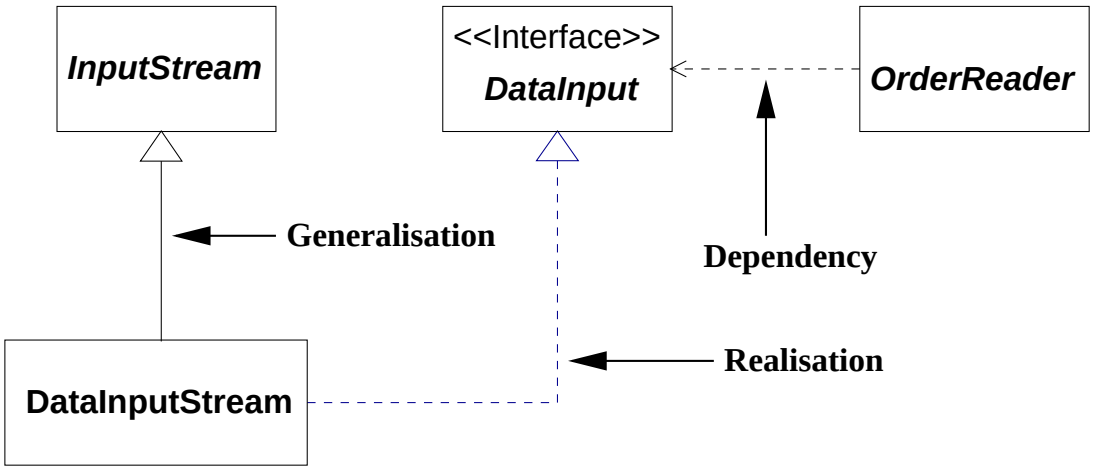

Figure 6: An example showing a realisation and a dependency.

- A dependency indicates that if a class (interface) changes then the dependent class (component) may has to change as well (i.e. not necessarily change). Dependencies are not necessary associated with interfaces, i.e. a class can depend on part of the implementation of another class.

Interfaces and abstract classes (methods) have their name shown in italics. Notice the different arrows and line styles used for realisation and dependencies. Dependencies should be minimised in a system.

An alternative notation for realisation and dependency on interfaces is shown in Figure 7.

10 Abstract classes vs Interfaces

```
abstract class Shape {
    public abstract void draw();
    void print_hello() {
        System.out.println("Hello!");
    }
}
interface Geometry {
    double calcArea();
}
class Circle extends Shape implements Geometry {
    private double radius;
    public void draw() {
        System.out.println("Drawing circle");
    }
    public double calcArea() {
        return(...);
    }
}
```

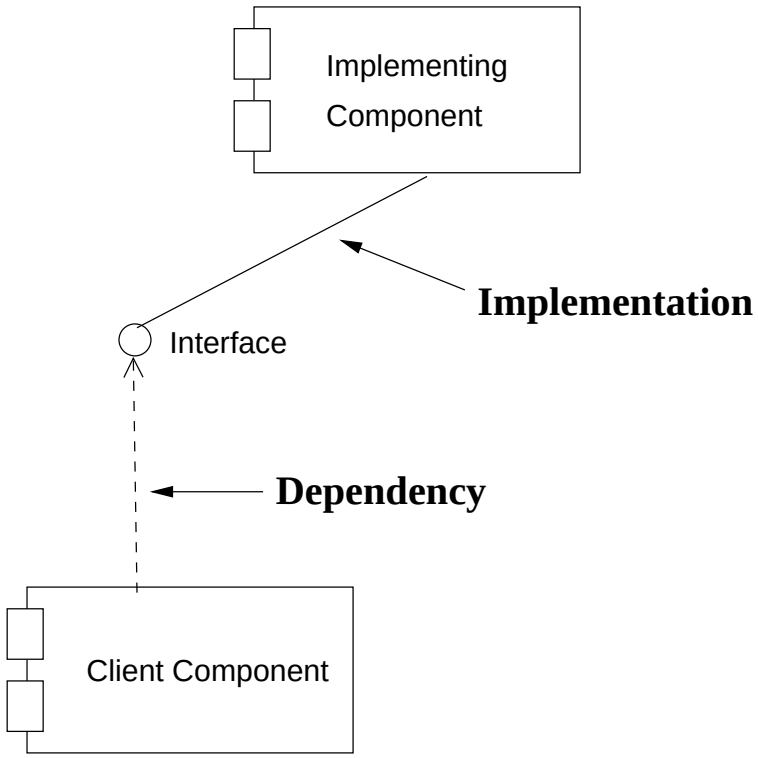

Figure 7: An alternative representation for realisation and dependency of interfaces.

11 Aggregation and Composition

Aggregation is the *part-of* relationship. An object is part of another object. Composition is a stronger variety of aggregation.

In composition the part object may belong to only one whole. The parts are expected to live and die with the whole. Usually any deletion of the whole is considered to cascade to the parts.

In Figure 8 an instance of *Point* may be in either a *Polygon* or a *Circle* but not both. An instance of *Style*, however, may be shared by many *Polygons* and *Circles*. Deleting a *Polygon* causes its associated *Points* to be deleted, but not the associated *Style*.

12 Composition in Java

```
class Engine {
    // ... details omitted
}

class Car {
    Engine eng;

    Car() {
        eng = new Engine();
    }
```

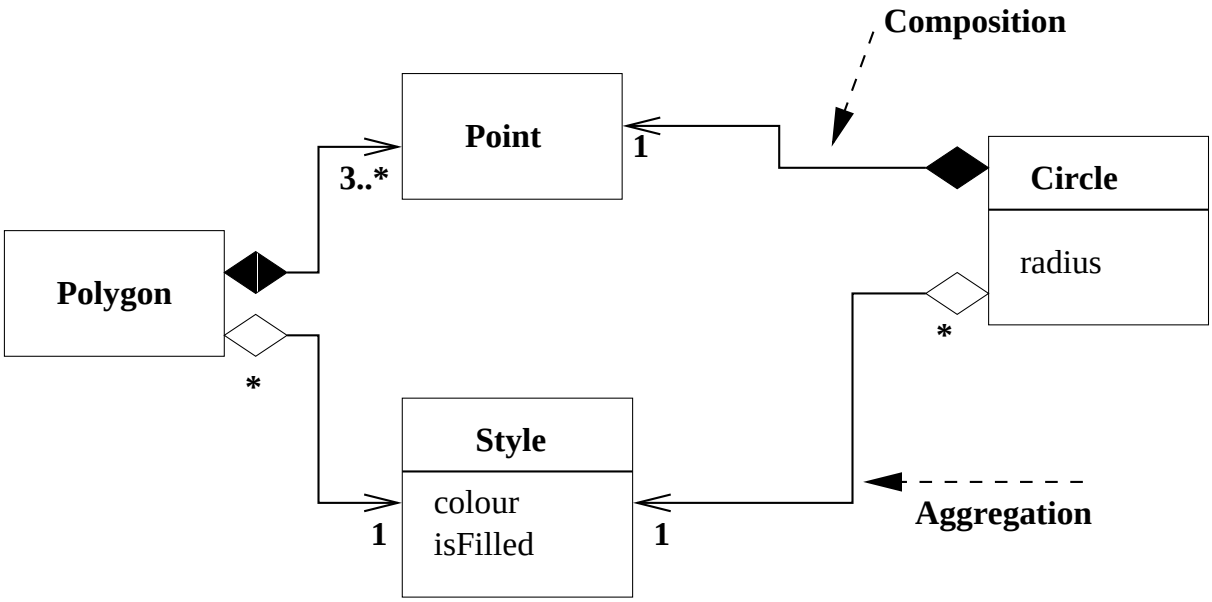

Figure 8: Composition and aggregation.

```
    public static void main(String[] args) {
        Car c = new Car();
    }
}
```

13 Aggregation in Java

```
class Color {
    // ....
}
class Fill {
    // ....
}
class Style {
    private Color color;
    private Fill isFilled;
}
class Circle {
    private Style style;
    Circle(Style s) {
        style = s;
    }
    // ... the rest of details omitted
}
class Polygon {
    private Style style;
    Polygon(Style s) {
        style = s;
```

```
    }
    // ... the rest of details omitted
}
class Paint {
    public static void main(String[] args) {
        Style style = new Style();
        Circle c = new Circle(style);
        Polygon pol = new Polygon(style);

        c = null;
        // object style still exists even if
        // garbage collector recycles!
    }
}
```

14 Object Diagrams

An object diagram is a snapshot of the objects in the system at a point in time. No messages are shown (see Figure 9).

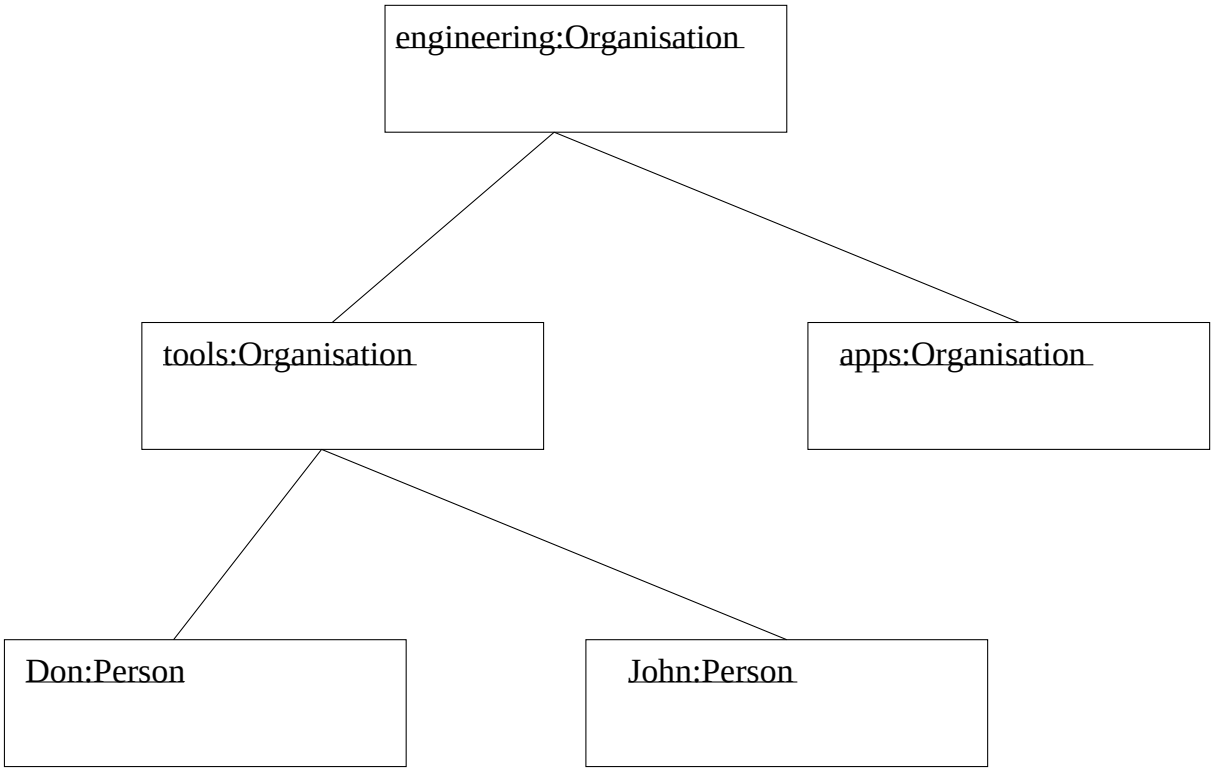

Figure 9: An example of an object diagram.

Names are underlined so as to indicate that these are objects, i.e. instances of classes.

15 Statechart Diagrams

Describe the state of a system by showing all the states of objects and how an object's state changes as a result of events.

There are many different variations of state diagrams.

Associated with each state there is an event list which defines the reactive behaviour in that state.

Each entry in the event list has the form:

```
event expression[guard expression]/action expressions
```

Two special events are:

- *entry*: occurs when the state is entered.
- *exit*: occurs when the state is exited.

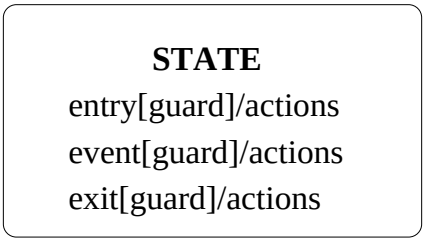

Figure 10: The three possible state event list entries.

Statechart example: The train ticket machine

Figure 11 shows a part of a statechart for an automatic train ticket system. A traveller is able to select the destination and the type of the ticket.

16 Sequence Diagrams

Consider the following system:

1. An Order Entry window sends a "prepare" message to an Order.
2. The Order then sends "prepare" to each Order Line on the Order.
3. Each Order Line checks the given Stock Item.
   - If this check returns "true", the Order Line removes the appropriate quantity of Stock Item from stock and creates a delivery item.
   - If the Stock Item has fallen below the reorder level, that Stock Item requests a reorder.

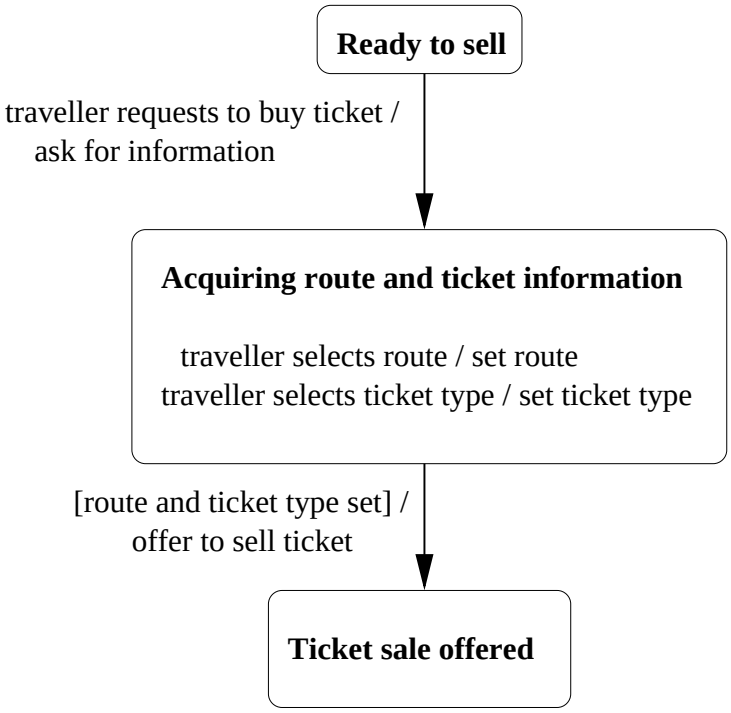

Figure 11: Part of the statechart for a train ticket machine.

Figure 12 shows the sequence diagram for this system.

A sequence diagram shows how operations are carried out. It shows messages exchanged between objects ordered by the time that they happen. Time progresses from the top to the bottom of the page.

The vertical lines are called the objects' lifelines. An asterisk denotes iteration, in the sense that a message is sent many times to multiple receiver objects, as it would happen when you iterate over a collection of objects. Dashed lines with an arrow, indicate a return from a message, not a new message.

17 Concurrency in Sequence Diagrams

Consider the following Java code. If this is reversed engineered then the equivalent UML diagram is shown in Figure 13.

```
class Transaction extends Thread {
    private Account account;
    Transaction(Account accnt) {
        account = accnt;
    }
    public void run() {
        account.transferMoney(100);
    }
}
```

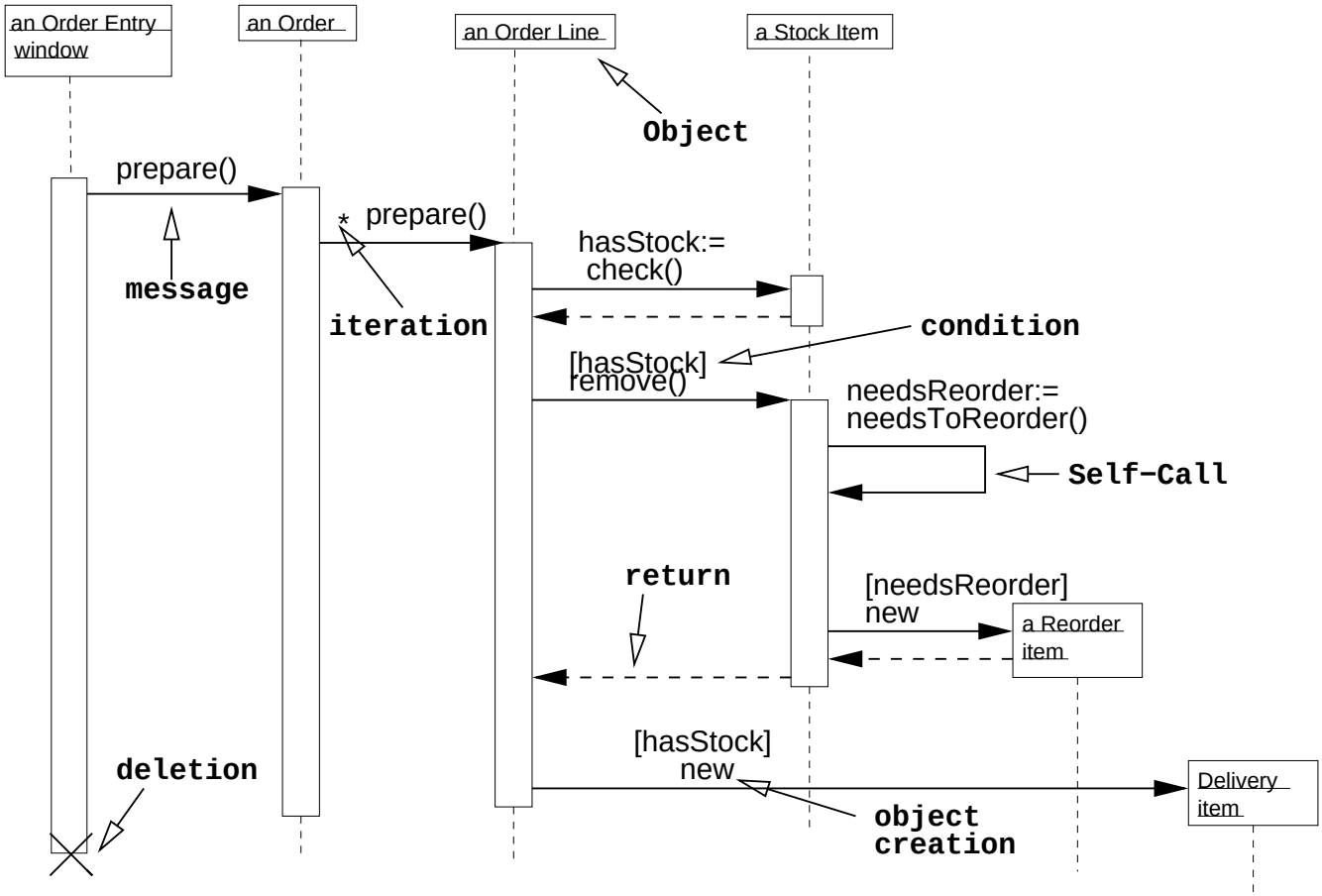

Figure 12: An example of a sequence diagram.

```
class Account {
    public synchronized void transferMoney(int amount) {
        withdrawMoney(amount);
        doActualTransfer(amount);
    }
    public void withdrawMoney(int amount) { /* ...*/ }
    public void doActualTransfer(int amount) { /*...*/ }
}

class Bank {
    public static void main(String[] args) {
        Account currentAccount = new Account();
        for (int i=0; i < 2; i++)
            new Transaction(currentAccount).start();
    }
}
```

The half arrowheads indicate an asynchronous message. An asynchronous message does not block the caller, so it can carry on with its own processing. Object deletion is shown with a large **X**.

18 Collaboration Diagrams

Collaboration diagrams contain similar information as the sequence diagrams. However, the sequence of messages is indicated by numbering the messages. With collaboration diagrams it is more easier to see how the objects are linked together and what is their role. Figure 14 shows an example.

19 Activity Diagrams

A fancy flowchart borrowing ideas from flowcharts, Petri nets, event diagrams and state modelling. Figure 15 illustrates an example.

20 Physical Diagrams

Physical diagrams consist of:

- *Deployment diagram:* shows the physical relationships among software and hardware components in the delivered system.
- *Component diagram:* these are the physical analogs of class diagrams. It shows the physical modules (packages) of code.

An example is shown in Figure 16.

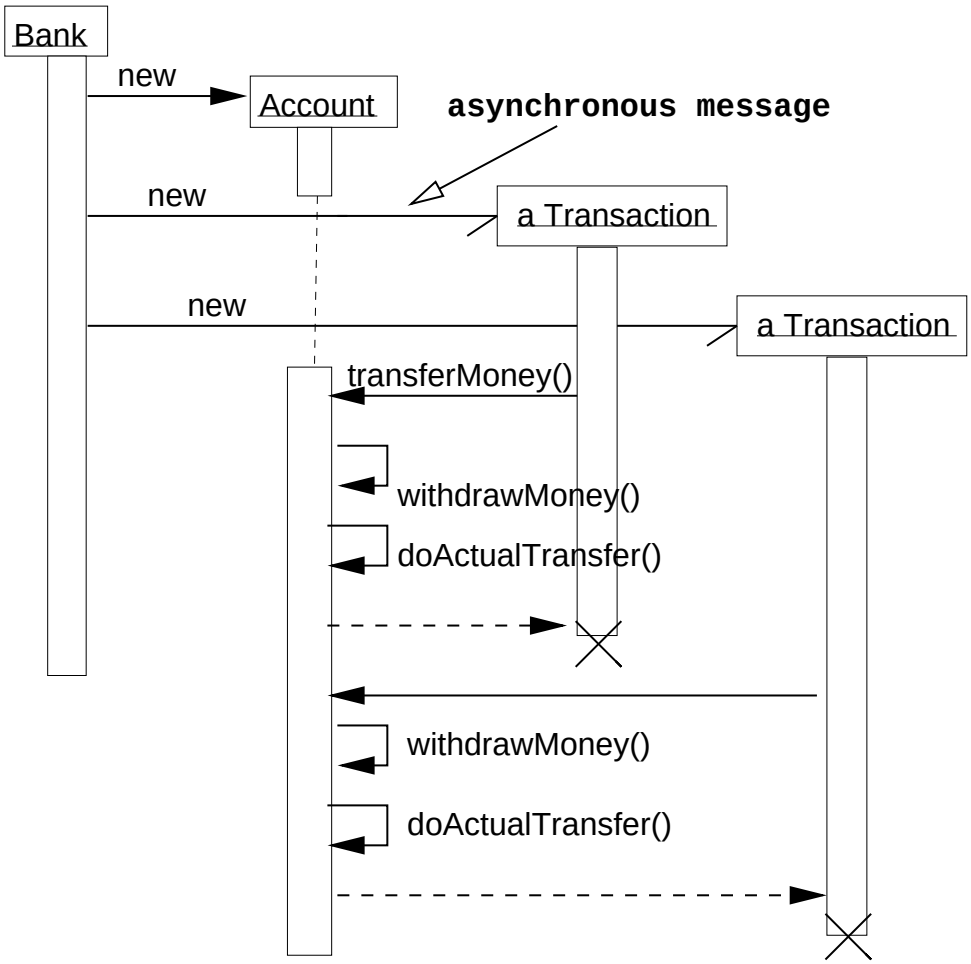

Figure 13: Concurrency in a sequence diagram.

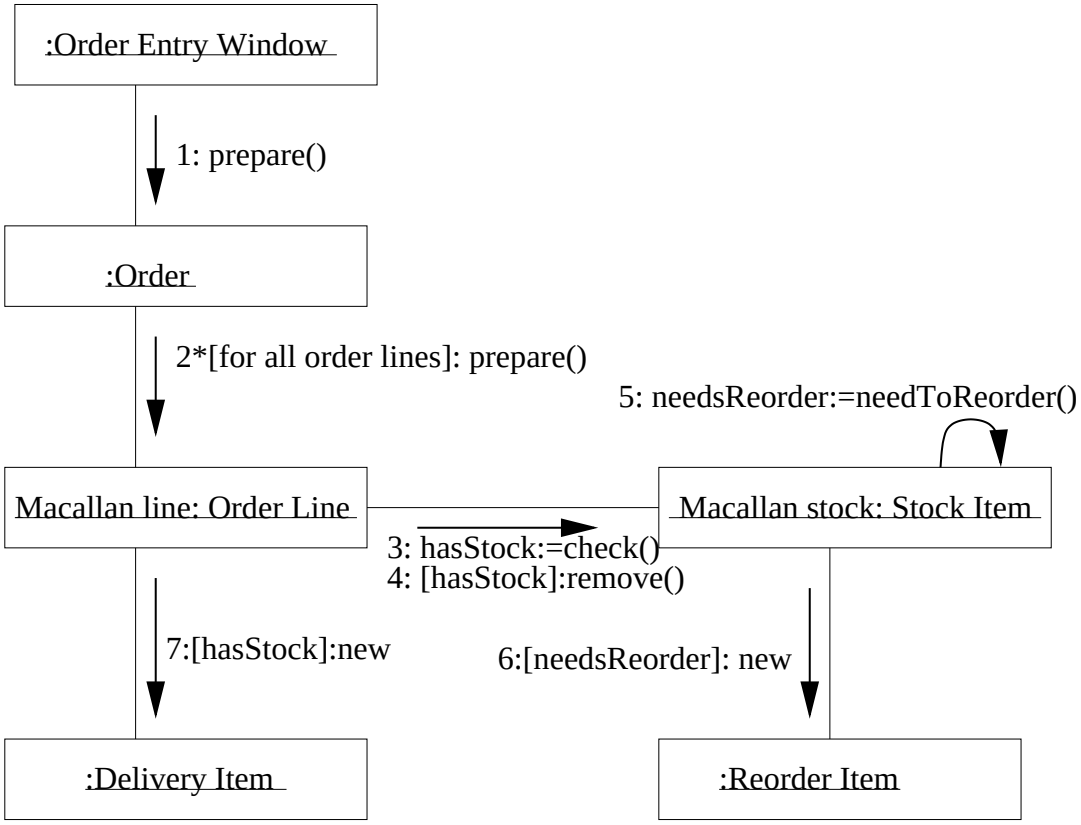

Figure 14: An example of a collaboration diagram.

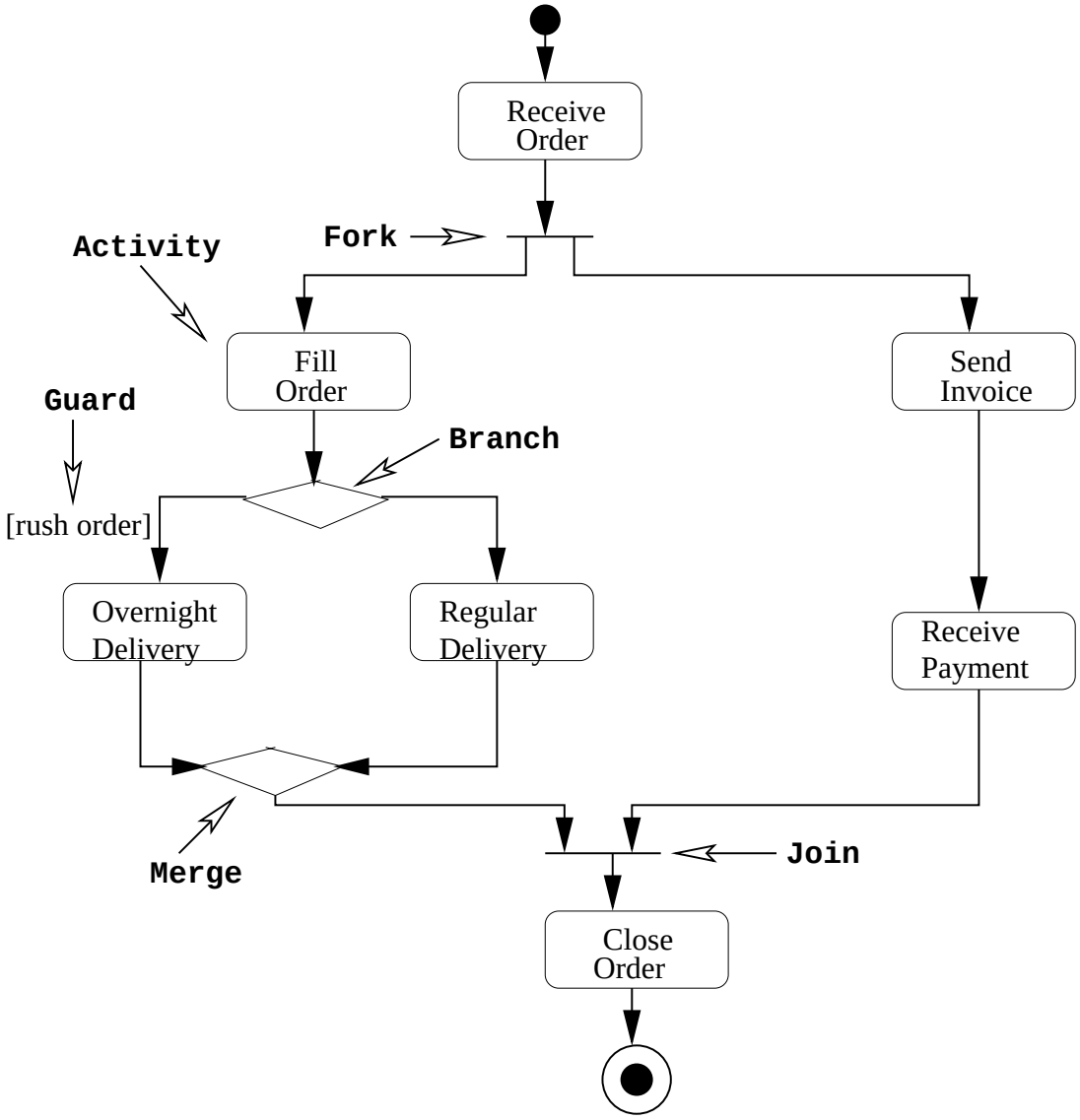

Figure 15: An example of an activity diagram.

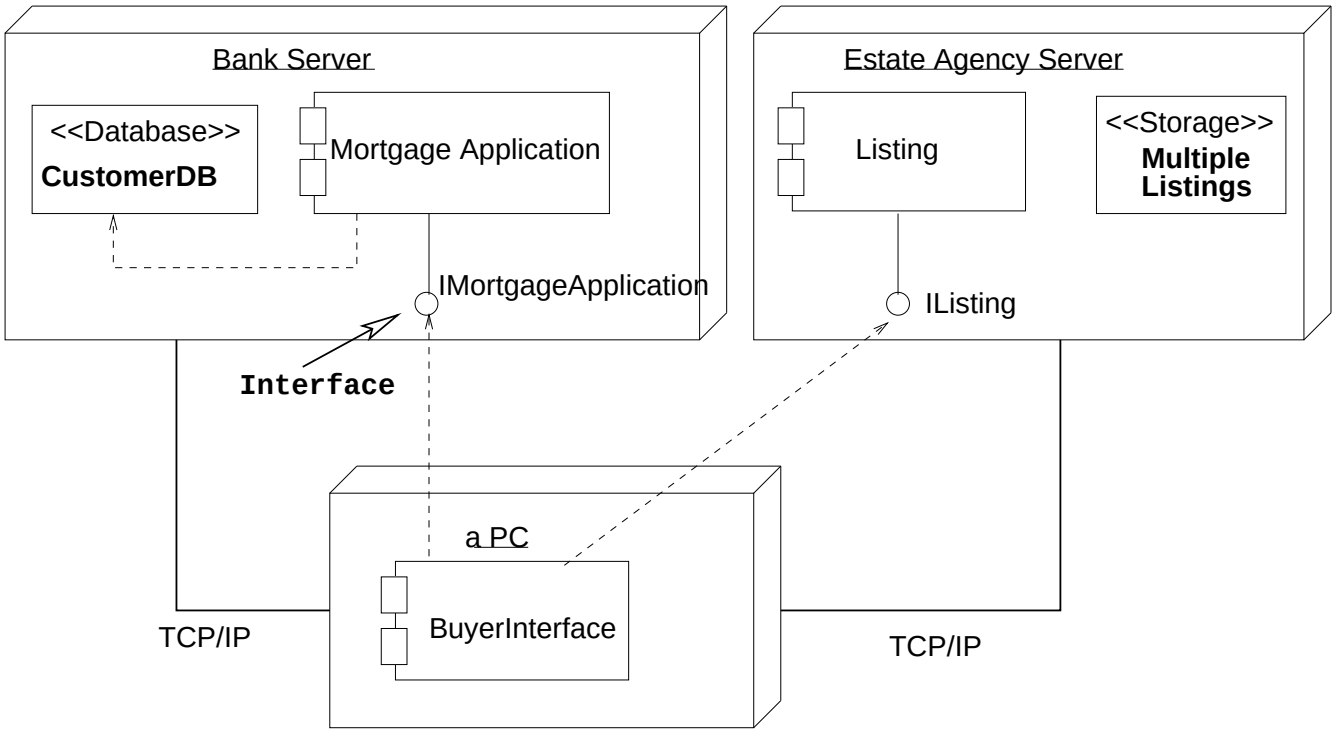

Figure 16: An example of a physical diagram, combining both a deployment and a components diagram.

21 Tools for UML (Commercial and Public Domain)

- Rational Rose
- Together Control Center (TogetherSoft Corporation)
- Inkscape
- Xfig
- Any of your favourite drawing software

Review Questions

1. What is the purpose of UML?
2. Describe the different types of diagrams used in UML?
3. What level of detail, do you think that UML diagrams should be showing? For example should we draw a sequence diagram with full detail (e.g. variable assignments, etc.) before implementing a system in code? Justify your opinion.
4. Choose the right answer and justify: Three items which are included in use case diagrams are:
   (a) Objects, activities, and communications.
   (b) Actors, messages, and activities.
   (c) Objects, use cases, and activities.
   (d) Actors, use cases, and communications.
5. Which of the following statements is correct?
   (a) Classes in class diagrams may be grouped into packages in order to illustrate the overall organization of a model.
   (b) In object diagrams, names of instances are in italics or all-caps.
   (c) If package B depends on package A, then any change in A will require a change in B.
   (d) Object diagrams and class diagrams are completely interchangeable.
6. Which of the following statements is correct?
   (a) Collaboration diagrams are dynamic models.
   (b) Sequence numbers in collaboration diagrams are optional.
   (c) Collaboration diagrams are static models.
   (d) Collaboration diagrams cannot show when an object sends itself a message.
7. What is the symbol for a component in a deployment diagram?
   (a) There is no symbol because components are not allowed in deployment diagrams.
   (b) A rounded rectangle, just like a state in a state diagram.
   (c) A 3-dimensional rectangular solid (like a box).
   (d) A rectangle with tabs on its left side.