Software Architecture and Patterns (CS2SAP)

- Software Design
- Design Techniques
- Database Design with Java
- Design Patterns
- Internet-based Application Design

Essential book!: “Software Design in Java 2”, K. Lano et al, Palgrave/Macmillen, 2002.

Part 1: Software Design

- Role of design in software development process
- Properties of software designs.
- Introduction to UML.

Lectures: Fridays at 10am and 11am.

Coursework: Design exercise, web exercise.

What is Software Design?

- For small programs, under 100 lines, with one or two classes, can get away with hacking code at terminal and ending up with something which, most of the time, works as intended.
- But this is far from industrial software production, where new systems have to be planned with care to ensure that they work correctly with existing systems, that they are reliable, and can be extended and maintained by people other than original programmer.
- Software design is essential for any non-trivial program: unless care is taken to structure code into sensible classes and methods, a program can quickly become very difficult to debug or extend.

Software Design: Definition

> *Software design is the organisation of a software system into modules/subsystems/components/classes or other units; the definition of behaviour and data storage responsibilities for these units, and the definition of interactions and collaborations between them which together meet the required functionalities of the system.*

Therefore design involves:

- Breaking up a required service or group of services (from those identified during analysis) into methods of classes and compositions of these methods, or services provided by external software.
- Selecting algorithms for services.
- Representing data which is required to support the system services, as program data within classes, or external data (eg,

held in a file or database).

- Organising connections and dependencies between modules to support the identified functionalities.
- Addressing non-functional requirements of system (performance, portability, maintainability, evolvability, etc) as appropriate at design level.

Design establishes *how* a system functions, as opposed to analysis, which describes *what* services system should provide.

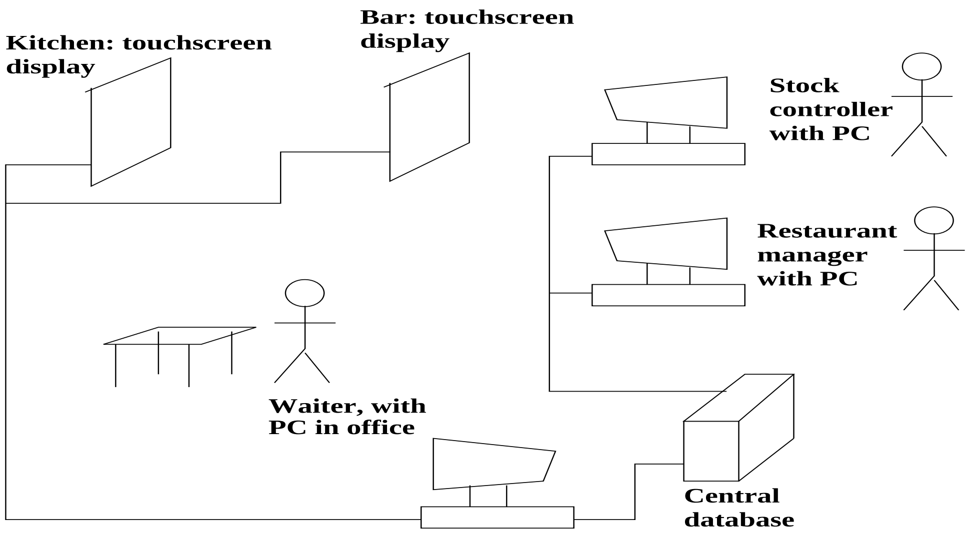
Design Case Study 1: Restaurant Management System

In a popular London restaurant, the following computerised system is required to speed up preparation of meals, manage stock and collect statistics on staff performance:

- Waiters enter orders (list of dishes and drinks ordered by a diner/group of diners) into system at their PC. Dishes to be prepared are sent to kitchen, drink orders to bar.
- Kitchen staff see dish orders on their screen, prepare them in an appropriate sequence and confirm preparation to system when complete, similarly with the bar.
- When waiter sees completion indications on his terminal he collects items and takes them to the table.
- The system prints a bill at end of the meal, and waiter enters details of payments for it. The manager can give discounts.

- System keeps track of number of customers served by each waiter, and amount of money taken by each waiter. The manager can view these statistics, and keep track of staff details such as shift hours and timing.
- Stock management part of system keeps a record of stock level of all ingredients which may be used in dishes of the restaurant, and of all drinks. When stock level of an item goes below a set level, a warning is given to stock manager, and a reorder can be set up.

Restaurant Management System



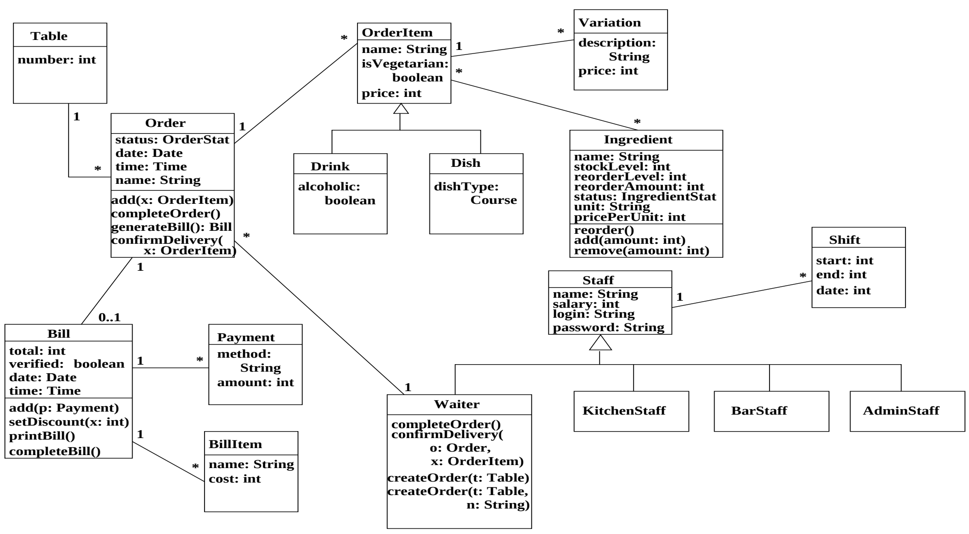
Restaurant Management System

We have been given the task to develop this system, using a standard PC platform where possible.

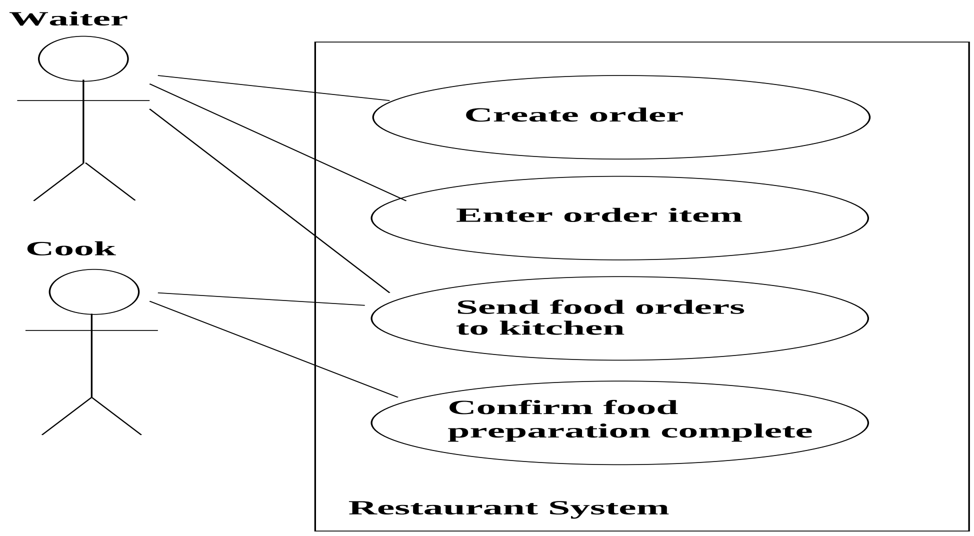
The first development steps are to define the entities and relationships in the system. An analysis model of the data of the system, expressed in a UML class diagram, is given below.

Also, *use cases*, describing the typical interactions and functions that different users will require from the system, are defined.

Class Diagram of Restaurant System



Selected Use Cases of Restaurant System



Waiter and Cook Use Cases

Create an order Waiter creates a new order instance, setting table number and order name (optional).

Enter order item A dish or drink ordered is added to current order, together with number of copies required and any variations (eg: no mayo).

Complete order Waiter closes the order.

Send food orders to kitchen System sends a list of the required dishes to kitchen, displayed on kitchen's terminal in the sequence in which they should be prepared (ie, all starters first, then all mains).

Send drink orders to bar System sends a list of required drinks to bar, where they are displayed on terminal there.

Confirm preparation complete A cook confirms that preparation of a particular dish is complete and it is ready for collection. Similarly for the bar.

Confirm delivery Waiter confirms that an order item has been delivered to table. It is removed from kitchen or bar display.

Bill order System computes charge for the order from data on dish and drink prices, and discounts, and prints a bill.

Complete payment Waiter enters details of how the bill is to be paid (eg, a mixture of cash and cards). Once payment is confirmed, order ceases to be a current order and is stored in archive of past orders.

Design Process for Restaurant System

What does design process consist of for this system? Firstly some decisions about implementation platform have to be made to guide choice of design.

Assume in this case that MS Access has already been mandated as a database, and that Java has been selected as implementation language.

So design has to include development of a relational database schema and definition of modules which store and retrieve data from this database, using JDBC package of Java (http://www.java.sun.com/jdbc) which enables information to be moved between database and a Java program using SQL statements.

The design must also:

- define components which provide the different GUI's required:

for waiter, kitchen, bar, manager and stock controller;

- define components which carry out computational tasks such as performance calculations and bill processing;
- define components which support transmission of data between different terminals, eg, waiters console and the kitchen.

Defining System Architecture

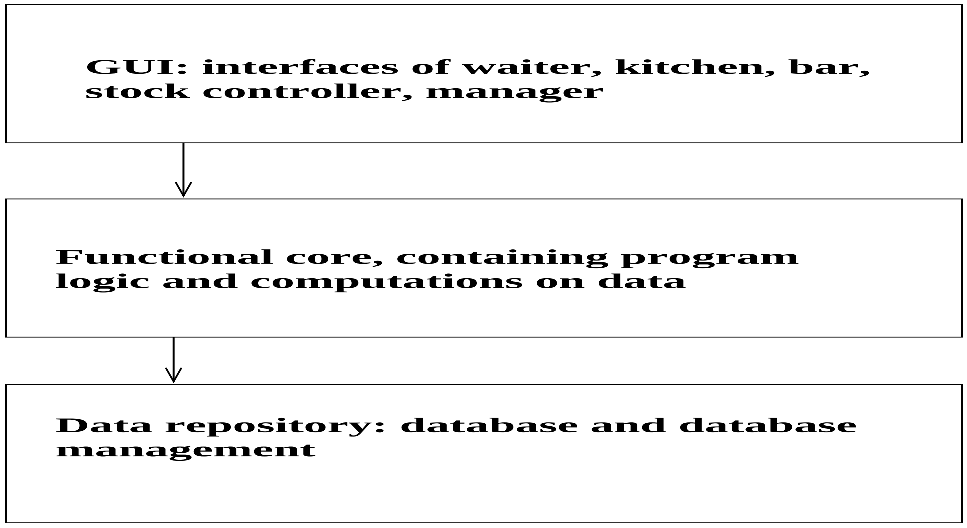
First step in design is to map out global *architecture* of system: what major subsystems exist and what their responsibilities and connections are. We could draw an outline architecture at this stage.

Once main subsystems are decided on, we can break down functions identified by analysis into operations provided by one or more of these subsystems, so identifying *their* requirements more precisely.

Finally, subsystem and module design repeats process with each subsystem, until we obtain individual class and method definitions.

In restaurant system specialised forms of design, such as GUI design and database design, will be needed in corresponding subsystems.

Restaurant Management System Architecture



What is a Good Design?

Some essential and desirable attributes of a design include:

- That it supports the given functional and non-functional requirements.
- That it is flexible and extensible – it can be effectively modified and extended to deal with changes to requirements and future enhancements.
- That it is comprehensible and verifiable – system structure and components should be easy to understand and to test that they satisfy the requirements.
- It should be as simple as possible, given need to fulfil the above properties.

Functional and Non-functional Requirements

- This is the most important property: a design which fails to meet functional requirements in particular will not be acceptable.
- By considering each use case and required service throughout development we can ensure that stated functional requirements are met.
- By considering each non-functional requirement at all relevant stages of design, we can also select design choices which meet these requirements.

Design for Flexibility and Extension

Most real-world software applications are not simply written once and then stay frozen in place unchanged for the rest of their lives. Instead, to remain useful, must be extensible to new requirements and operating environments due to business and technological evolution. Therefore software design has to consider such change.

At most basic level should avoid features of a language or external software which are *deprecated*: (due to be phased out in next upgrade of language/software) or likely to be deprecated in foreseeable future.

At higher level, design structure should support maintenance: changes to one module in system should not affect other modules in uncontrolled manner.

Comprehensibility and Verifiability

One guideline for improving comprehensibility is to ensure that each component has a clear purpose and meaning, so there is no ambiguity about what services it provides and how it fits into a larger component/system.

Eg, in restaurant system, well-defined subsystem of GUI could be waiters GUI. This would contain operations for creating and updating this display and dealing with waiters input actions.

Methods should have meaningful names which indicate what they do. Java has several naming conventions for methods which should be used in most cases: if class C has attribute att, then method to return value of att is called $getAtt$, unless att is boolean, when the method is called $isAtt$. Method to set att to new value is called $setAtt$.

Simplicity

The design should minimise dependencies between components: component M *depends on* component N if M invokes methods of N or uses other features of N.

Dependency of M on N is indicated by an arrow from M to N on a module dependency diagram, and this also means that if N is modified, M may need to be modified (because its behaviour may alter due to possibly changed effect of N's code). Therefore minimising dependencies will, in principle, reduce 'knock-on' effect of a modification in a module.

Simplicity should also be aim for individual modules: a module may need to be rewritten during development to make it more simple and robust, if original design idea is leading to more and more complexity, need for special cases and flag variables, etc.

Design Principles

The following properties contribute to a good design:

- *Separation of responsibilities within an application*: for example, dividing application into 'tiers' which are relatively independent large subsystems such as GUI and interface to a database. These tiers do not depend on internal organisation or implementation of each other.

  This has advantage that tiers can be independently developed (once their interface is agreed) and are resilient to changes in other tiers.

- *Modularity*: system (or its subsystems) are divided up into modules, which are coherent groups of operations and data. A module may correspond to a class, or to a group of closely related classes.

- *Low coupling and high cohesion*: modules will depend on other

modules, for example GUI module may invoke operations from another module to make a request for information or enter data into application. GUI is therefore dependent on module whose services it uses. A design exhibits low coupling if there are relatively few dependencies between modules: only minimum needed to carry out the functions in system should exist, system should be organised so that most method calls occur within modules instead of between them. High cohesion means modules are internally strongly related, they are not random collections of data and operations but have single clear meaning and purpose.

- *Coding standards, tools and notations such as UML*: where possible an application should be developed with uniform set of coding standards, such as placement and style of comments, naming conventions for methods, classes and variables, layout of classes, use of exceptions, etc.

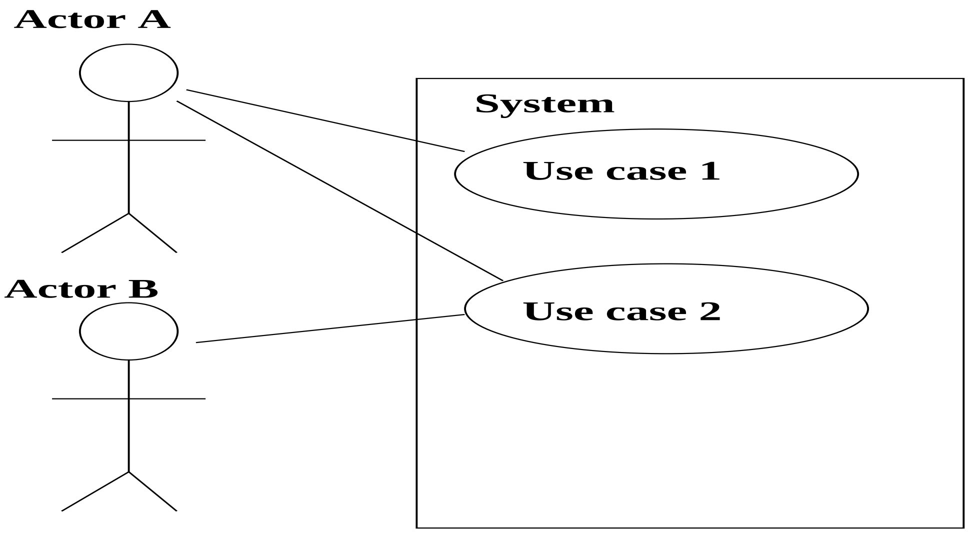
Notations for Software Design

- use cases, class diagrams, statecharts and interaction diagrams from UML;
- operation specifications.

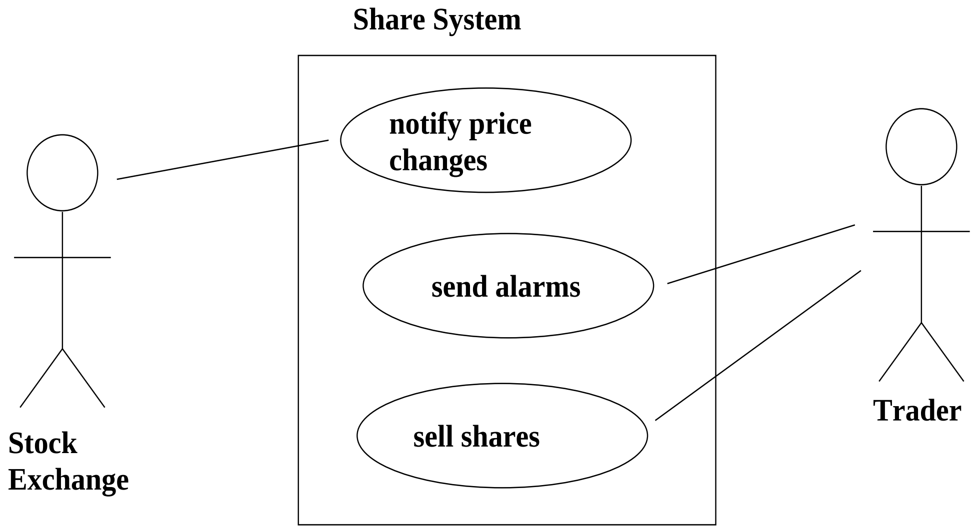
Use Cases

A use case model describes (1) system to be constructed, (2) the *actors* – representing a role played by person or other entity that interacts with system, and (3) *use cases* – families of usage scenarios of application, grouped into coherent cases of functionality.

Simple Use Case Model



Share System Use Cases



Above we give use cases for a share management system, price changes are informed to system from a stock exchange, and trader using system may receive alerts from it when share price changes pass some preset limits.

Trader can also use the system to buy shares.

Use cases may have additional textual descriptions. A complete textual description of a use case includes:

1. *Start of use case* – the trigger event: "the use case begins when e occurs".
2. *End of use case* – the termination event: "when f occurs the use case terminates".
3. *Interaction between use case and actors* – this identifies what activities should be inside system and which outside.
4. *Exchanges of information* – what data items are passed

between system and actors.

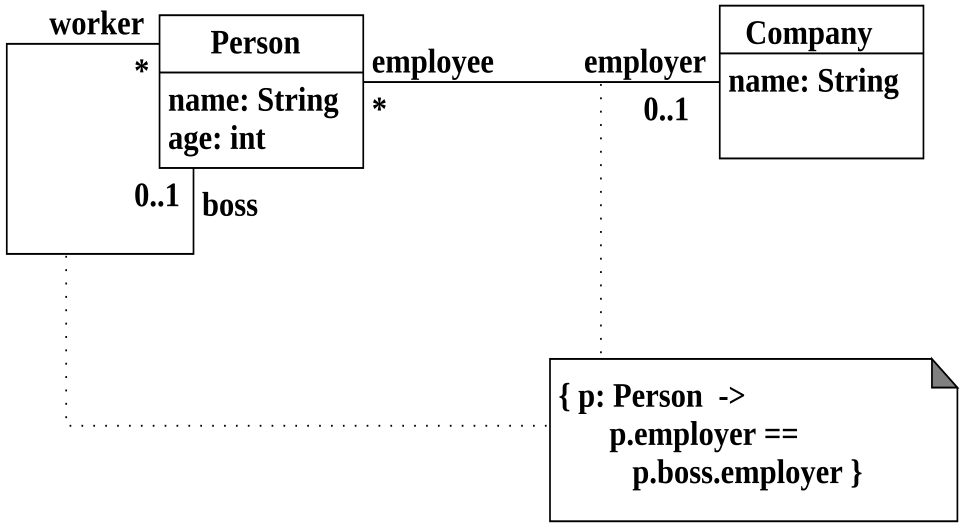
5. *Chronology and origin of information* – identifies when system requires internal or external information and when it records it.
6. *Repetitions of behaviour.*
7. *Optional situations* – points where actor or system may choose different behaviours within use case.

UML Class Diagrams

UML class diagrams define data which an application deals with, as set of entities (called classes) and relationships (called associations) between them.

Following is simple UML class diagram, describing two entities and two relationships.

Simple UML Class Diagram



Basic elements of class diagrams are:

- *Classes*: displayed as rectangles with three components: name (italic font for name denotes an abstract class: one without direct instances); attributes; operations.
- *Associations*: displayed as lines. Cardinality constraints are written on end that they constrain: $*$ means many (zero or more), default (if no annotation is present) is 1.

  Associations can be 'read' from one entity to another using role names and cardinalities: each *Company* has *many employee*s (*Person*s); each *Person* has zero or one *employer* (a *Company*).
- *Constraints*: invariants which relate several elements in diagram. Written in 'dog-eared' boxes, and linked by dashed or dotted lines to elements they relate. In constraints can use mathematical notation such as *OCL* (Object Constraint Language), or Java-like expressions.

Constraint in example above relates two associations in diagram, and expresses that boss of each employee works for same company as themselves:

$$\forall\, p : Person \cdot p.employer = p.boss.employer$$

- Attributes are written in form *name* or *name* : *Type*, where *Type* is type of the attribute *name*. Visibility constraints $-$ (private), $+$ (public) and # (protected) can also be used to annotate attributes.
- Operations are written in form *op*(*params*) for operations that do not return a result, and *op*(*params*) : *Type* for operations returning a result of type *Type*. *params* is a list of parameter declarations $p_1 : T_1, \ldots, p_n : T_n$ which may be empty.
- Associations can be directed: if there is arrow at one end of an association it means that we need to navigate from entity at base of arrowed association to entity at end with the

arrowhead, in design and implementation of system. There is therefore a dependency in direction of arrow.

- Eg, if *Company* needed to call an operation on *Person* to change the boss of a person, there would need to be a navigation arrow from *Company* to *Person*. *Person* in such a situation would be called a *supplier* to *Company*, because it supplies operations and functionality to it. *Company* is called a *client* of *Person*.

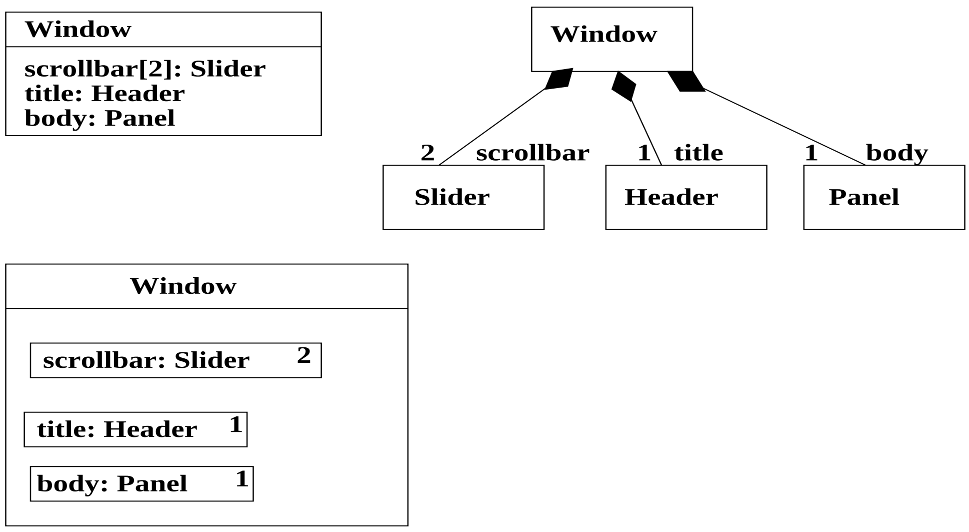
Composition

An association can express a strong form of binding/ownership between objects of one class and objects of another: this is termed *composition* and represented by a filled diamond at ‘whole’ (owner) end. Eg: a company owns a number of departments: lifetime of each department is contained in that of company. A composition association from A (whole) to B (part) should be:

1. transitive (a part of a part is also a part of the whole)
2. irreflexive (an object can’t be a part of itself)
3. one-many (no sharing of parts between different wholes)
4. deletion propagating from A to B: deleting a whole deletes all its parts.

Parts are similar to attributes, hence the alternative notations in following figure.

Alternative Notations for Composition



Inheritance

Inheritance of class A by class B is represented by a line from B to A with an open-headed triangle at A end (as with directed associations this indicates that B depends on A). B is called a *descendent* or *subclass* of A. If A represents an interface (a class which merely defines signature of methods, not their implementations), then line may be dashed, indicating an *implements* or *realisation* relationship.

UML *object diagrams* use a variant of class diagram notation to show typical arrangements of objects (instances of classes) and links (pairs of objects, ie, instances of associations). Objects are represented as rectangles, but their names are underlined and their class is also usually indicated: $\underline{wayne : Student}$. Links are represented as lines between objects they relate.

Statecharts

A class C may have a statechart describing life history of its objects. Statecharts show all possible states or 'modes' of object, as rounded boxes, connected by transition arrows:
$Source \longrightarrow Target$. States have names, and effectively represent (disjoint) subclasses of C.

Transitions are labelled
$event(params)[Condition]/action_1/action_2/\ldots/action_n$ where

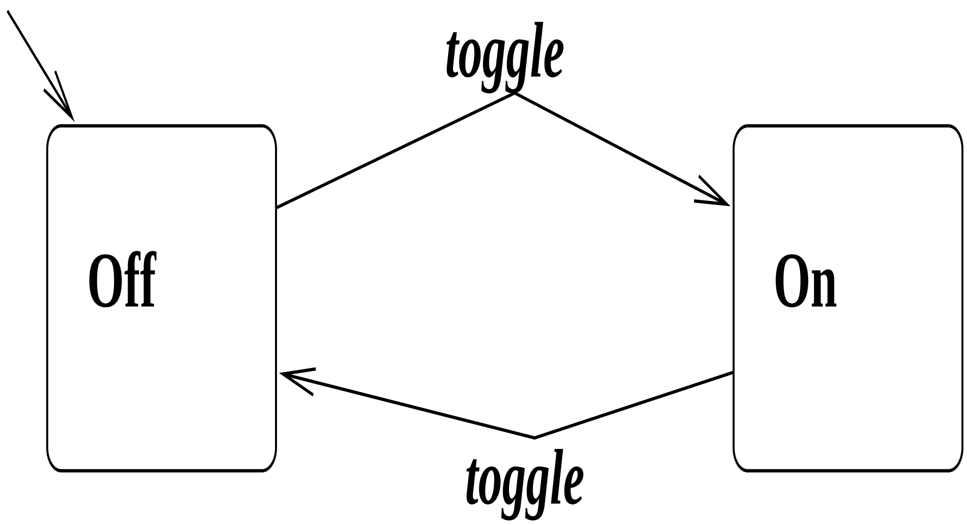
- $event(params)$ is name of event which triggers transition (eg, name of method which has been called on object)
- $Condition$ is a boolean test on $params$ and attributes of C: if condition is true when trigger occurs then transition is taken, otherwise it is not
- $action_i$ are actions, which can be assignments to attributes of C or method calls on its supplier objects

- states may contain activities, given by writing *do activity* in their box. These activities can include program code or specification of maximum residence time t in state: $wait(t)$.

A transition is only taken if object is in source state, and trigger event happens with the condition true. The object then moves into target state.

Simple example (light switch):

Statechart of Light Switch



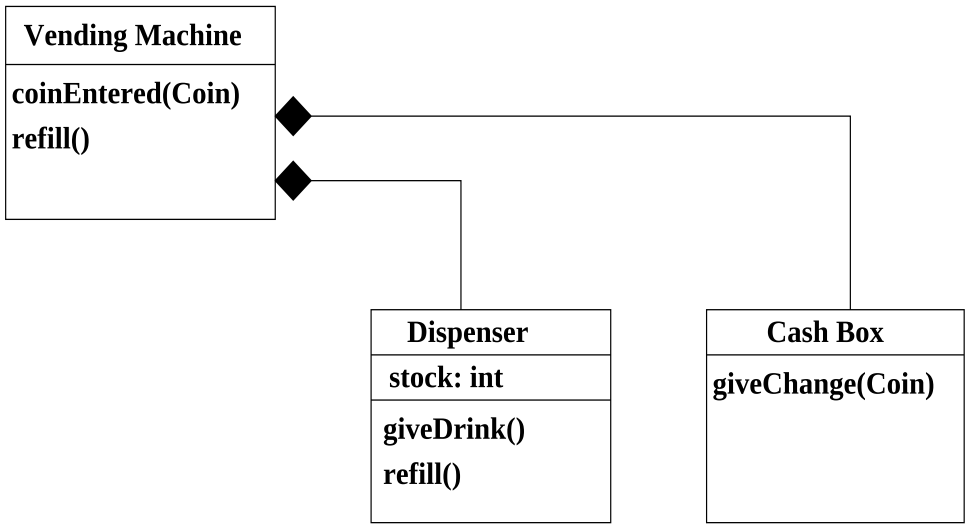
Example: Vending Machine Controller

System to be controlled is a vending machine for drinks costing 15p each (this is an old example!).

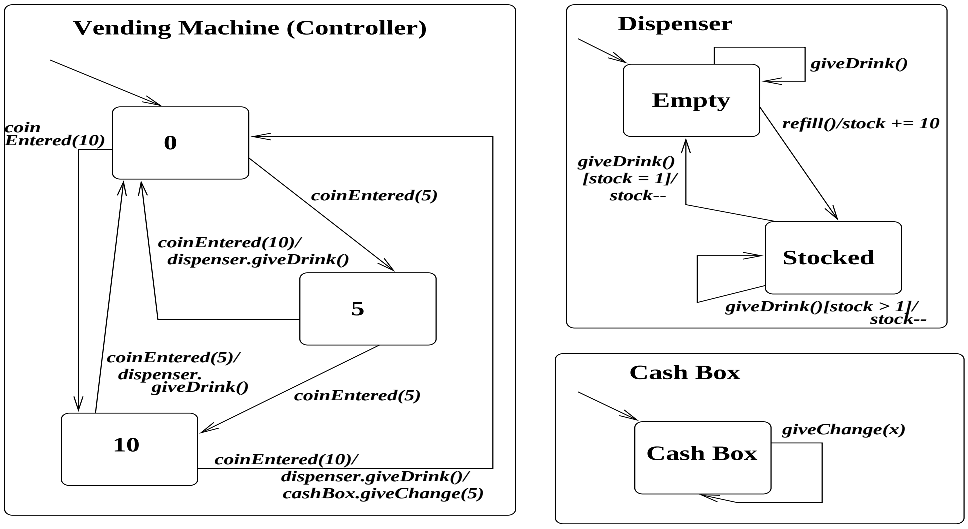
A customer can enter 5p and 10p coins only, machine should dispense a drink once sufficient money has been entered, returning any excess change. The machine can also be refilled with 10 drink cans at any time.

Notice use of default role names *dispenser* and *cashBox* to denote the objects at opposite ends of 2 associations from vending machine.

Class Diagram of Vending Machine System



Statecharts of Vending Machine System



Statecharts

When we use statecharts to describe system behaviour:

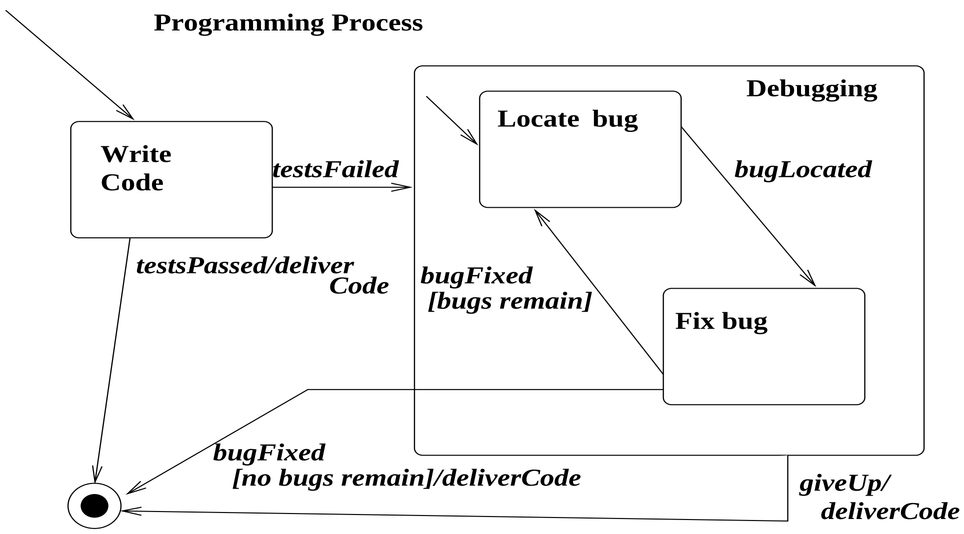
- initial states are shown as target of an arrow with no source. They represent state system is in when it starts up;
- trivial statecharts (eg: *CashBox*) are usually not shown;
- a good design is one with clearly identified and separated responsibilities, and a pure client/supplier hierarchy: *Controller* sends commands to *Dispenser* and *CashBox*, no commands go in opposite direction;
- Java notation is used for actions: we could alternatively use pseudocode or other kinds of notation: $stock := stock + 10$, $stock' = stock + 10$, etc. provided we are consistent.

Nested States

States may have other states nested within them, to describe further substates/subclasses.

- states can be nested in others;
- an initial substate is given: a transition to the enclosing state actually goes to this substate (eg, *testsFailed* goes to *Locate bug*);
- a transition whose source encloses other states represents a family of duplicate transitions from each enclosed state (unless these enclosed states have a more specific transition for same event) to target state: *giveUp* exits whole debugging process and delivers code, regardless of its state . . .;
- a 'bullseye' denotes final state: end of life history of the object.

Example of Nested States



Interaction Diagrams

Interaction diagrams show examples (scenarios) of particular use cases. There are two forms of interaction diagram in UML: sequence and collaboration diagrams.

Sequence diagrams emphasise timing and history of objects. Sequence diagrams relate an interaction to statechart models of classes involved, and can be used to give examples of interactions involved in a particular use case scenario.

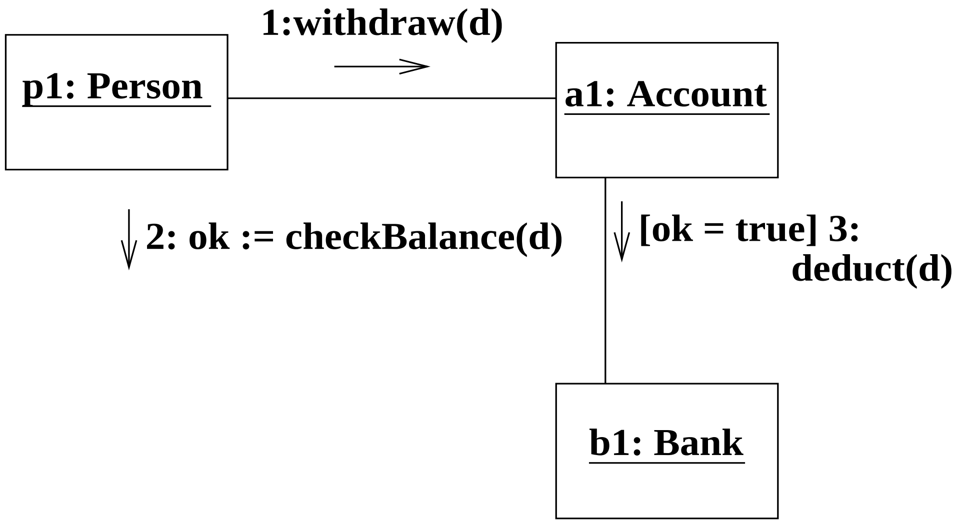
Collaboration diagrams in contrast emphasise connection of interaction to the class diagram

Collaboration Diagrams

These show an interaction between objects as a set of messages sent from one object to another to achieve some required unit of functionality (eg, a module operation execution).

Diagrams consist of objects, links (instances of associations) and messages, they enhance UML object diagrams by showing explicitly what messages are passed between objects and the order of these messages.

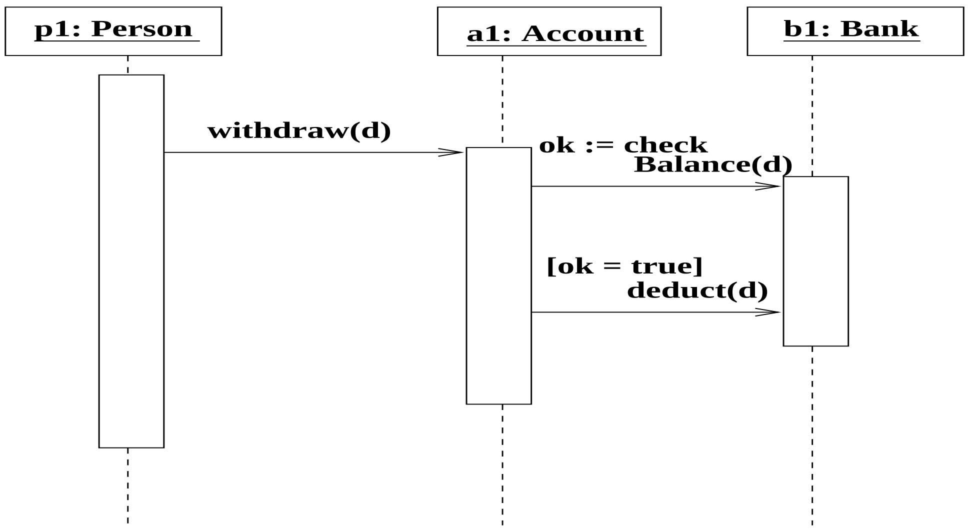
Example Collaboration Diagram



The features of collaboration diagrams are:

- Objects are represented by rectangles containing name/identifier of object, and name of its class, all underlined: <u>object: Class</u>.
- Objects created during operation execution are noted as $\{new\}$.
- Links (instances of associations) often have arrows indicating their navigability direction: if a message is sent from object a of class A to object b of class B then a must have access to b, for example by means of an association from A to B which is navigable in that direction.
- Messages are numbered consecutively starting at 1, nesting of messages is indicated by suffixes: eg: 3.1.4 follows 3.1.3 within subprocedure 3.1.
- Conditions are given in square brackets: associated message is only sent if condition is true.

Example Sequence Diagram



Sequence Diagrams

In this diagram there are three objects, $p1$, $a1$ and $b1$, which exist throughout interaction (their lifelines are the vertical dashed lines).

Vertical rectangles indicate an activity of objects (usually an execution of a method on the object) and the duration of this activity.

Arrows from one object lifeline to another represent messages, usually method invocations. Parameters, results and conditions are shown as in collaboration diagrams.

Method and Operation Specifications

These are used to describe abstractly what a method or operation does, without giving details of an algorithm or code. They consist of:

- *REQUIRES* – what conditions are needed on input parameters and starting state to ensure that operation or method executes without error, and to ensure that EFFECTS are true at termination. REQUIRES conditions are also called *preconditions* or 'Assumes' conditions.
- *EFFECTS* – what is true of outputs of method/operation, and the final state at normal termination of method/operation, if precondition is true at its start. This is also called *postcondition* of method or operation.

Callers must ensure precondition, will then be guaranteed that postcondition holds when call of operation completes.

If an operation parameter is modified within method, we should explicitly declare it as such in a *MODIFIES* clause of specification:

```
public class VectorUtil
{ // OVERVIEW: Provides useful utility procedures for
  // manipulating vectors

  public static void removeDups(Vector v)
  { // REQUIRES: All elements of v are not null
    // MODIFIES: v
    // EFFECTS: Removes all duplicate elements
    //          (under equals) from v. Order of
    //          remaining elements may be changed.
    if (v == null)
    { return; }
    for (int i = 0; i < v.size(); i++)
    { Object x = v.get(i);
      int j = i+1;
```

```
      // remove all duplicates of x from rest of v
      while (j < v.size())
      { if (x.equals(v.get(j)))
        { v.set(j, v.lastElement());
          v.remove(v.size() - 1);
        }
        else
        { j++; }
      }
    }
  }
}
```

We don't require $v \ != \ null$ in precondition, since this case is dealt with normally in method body: it is valid to call method with null argument. Since *VectorUtil* is a utility package, it should work in widest range of possible circumstances. Indeed it should ideally tolerate null elements inside vector as well.

Part 2: Design Techniques

This part describes processes and techniques that can be used in object-oriented design.

Design Steps

Following steps in design can be distinguished, independent of notation and application:

1. *Architectural design*: define global architecture of system, as a set of major subsystems, and indicate dependencies between them.

   For example, partitioning a system into a GUI, functional core, and data repository. The GUI depends on core because it invokes operations of the core, and core may depend on GUI (if it invokes GUI operations to display result of computations, for example). The core depends on repository. There may be no other connections.