# SIA: Software Engineering of Internet Applications

## N.B.

These notes follow the lecture slides made by Professor Stefan Edelkamp and Mr. Josh Murphy who taught SIA at King's College London in 2018 and they contains some of the diagrams included in their lecture slides.

For use cases and examples, it is recommended that you consult the slides made by Professor Stefan Elekamp, as these notes do not contain them, and the examples given in the slides can help understand some of the theory of the module.

**I do not guarantee the completeness and correctness of these notes** although to the best of my knowledge, they are correct. They contain information that I deemed to be important for my personal use, so some definitions or concepts are not included here. They are not endorsed by King's College London, the module lecturers, or any member of College staff and they have not been verified by any qualified individual.

These notes can be shared with anyone who may find these useful.

# Table of contents

# 1. Course Overview

## What is software engineering?

Specification:

- A **detailed, precise** presentation of something or a **plan** or proposal for something;

Architecture:

- The **art or science** of building; more specifically, the art or practice of **designing** and building structures which are habitable;
- The way in which the components of a computer or computer system are **organised** and **integrated**;

Engineering:

- The **application of science and mathematics** by which the properties of matter and the sources of energy in nature are made useful to people
- The **design** and **manufacture** of complex products (software engineering)

## Why bother with software engineering?

Bugs can appear when improper processes are in place, and they can be humorous (the suicidal robot at Budd Company), frustrating and expensive (people receiving 10x the requested money at a Norwegian Bank), or even deadly (the X-ray bug in Therac-25).

Between the 60s and 70s, software was simple, memory was highly restricted (punch cards) and people dealt mainly with low-level, machine-oriented programming languages (Cobol, Fortran, Algol). Even then, there were bugs which spurred interest in semantics and verification questions. The term **software crisis** was created to describe the difficulty of writing useful and efficient programs in the required time.

Between the 70s and 80s, there were improvements in hardware which allowed engineering in large projects. This brought new problems:

- Development had to be split ⇒ **decomposition** in analysis, specification, and coding;
- Unstructured programming methods don't scale: global data, non-modular construction, lack of well-defined interfaces.

The internet age brought new complexities:

- Large scale, distributed, heterogeneous systems; internet-centric computing
- Cheap microsystems leads to massive distribution:
    - Cloud computing
    - Internet of things

## Software engineering

Software engineering uses **techniques**, methods, and tools that can reduce the complexity of constructing systems.
There are also tools for building specific kinds of systems with high degrees of **reliability**.
This course focuses on **semi-formal** methods of development.

A **language** is **formal** when it has formal language (syntax) and its meaning (semantics) is described in a mathematically precise way.
A **development** method is **formal** when it is based on formal languages and there are semantically consistent transformation rules.

An example of a **semi-formal** method that is widely used is **UML** as it provides mainly syntax and a bit of semantics.

Advantages for formal methods:
- Usually more concise
- Precise and unambiguous
- Precise transformation rules ⇒ machine support possible
- Uniform framework for specification, development and testing

However, they are more **difficult** for novices.

## Structuring and abstraction in modeling

**Modeling** is used in the **early development** phases and it aims to **specify the requirements clearly and precisely** while **avoiding a premature commitment to algorithms or data structures**.

After that, there could be models of a possible implementation architecture: model sketches components, interfaces, communication, etc.

Modeling is important for overcoming the complexity of program development in large projects:
- **Structuring** servers to organise/decompose the problem/solution;
- **Abstraction** aims to eliminate insignificant details.

Classic approaches to structuring and abstraction:
- **Functional decomposition**: decomposition in independent tasks;
- **Parametrization** and **generic development**: reusability;
- **Model simplification**: to improve understand of tasks and possible solutions;
- **Information-hiding**: interfaces and property-oriented descriptions.

In all situations, interfaces must be clearly described:

- **Interface**: imagined or actual border between two functional entities with fixed rules for transfer of data;
- **Syntactic properties**: the available rules, the types of their arguments, etc.;
- **Semantic properties**: a description of the entities behaviour; the goals is to support the proper use of the entity.

Interfaces also provide the basis for communicating and explaining (sub)systems between the specifier, implementer and the user.
Correctly describing interfaces and ensuring their correct use is a central aspect of software engineering.

# 2. Model-driven development

## MDD and MDA

**Model-driven development (MDD)** considers development of software systems as a process consisting of:

- **construction** and **transformation** of models;
- **semi-automated generation** of executable code from models, instead of manual construction of low-level code.

Aims to make production of software systems more **efficient** by:

- freeing developers from **complexity of implementation** details;
- **retaining** core functionality of a system despite changes in its technology.

MDD is extremely relevant for web application because of **rapid change** in web technologies and similar processes in many web applications.

We focus on model-driven architecture (MDA) approach to MDD.

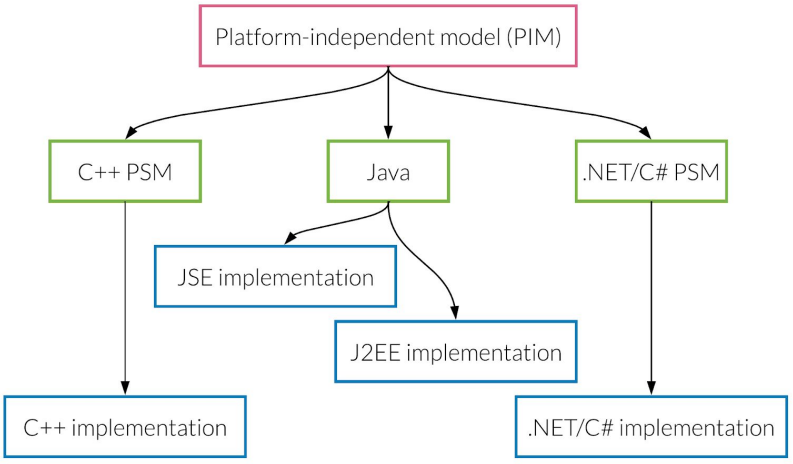
**Model-driven architecture (MDA)** aims to support cost-effective adaptation of a system to new technologies and environments by separating **technology-independent models of data and functionality of a system** from **technology-specific models**.

Key concepts of model-driven development using MDA are:

- Platform-independent models (PIM)
- Platform-specific models (PSM)
- Model transformations

There are tools that can help create executable systems from a PIM (Eclipse, UML2Web)

MDA process example:




Model transformations can be used to generate PSM from PIM, and then model transformations can be used on PSM to generate implementations.

A **platform-independent model (PIM)** is a system specification or design model that models systems in terms of domain concepts and **implementation-independent constructs**.

PIMs should:

- express "business rules" that are core definitions of **functionalities** of the system;
- be **reusable** across many different platforms via use of PIMs;
- be **flexible** to accommodate enhancements and other backwards-compatible changes.

If a PIM is defined in a way that abstracts from specific algorithms and computation procedures, then it can be considered a **computation-independent model (CIM)**.

A **platform-specific model (PSM)** is a system specification or design model that is **tailored to a specific software platform and programming language**, such as J2EE and Java, or .NET/C#.

It defines functionalities of a system in sufficient detail that they can be **directly programmed from the model**.

A **model transformation** produces a new model from an existing model to:

- either improve the quality of model (e.g. by removing redundancies, or factoring out common elements of classes or operations), or
- refine a PIM towards a PSM, or
- refine a PSM to an implementation.

Typical transformations include:

- introducing introduction of design patterns
- elimination of model elements that are not supported by a particular platform

# Unified modelling language (UML)

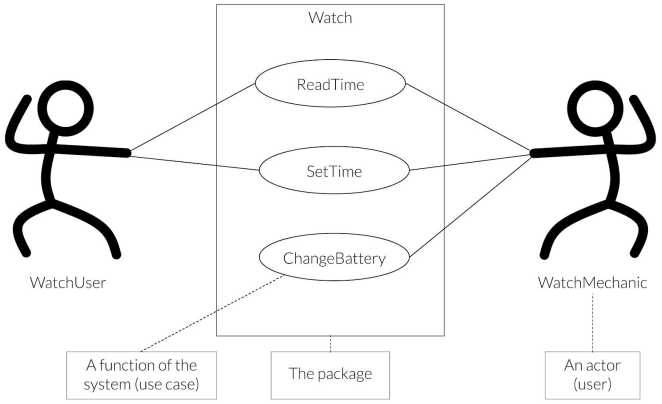
**UML** consists of several interrelated diagrams and textual notations:

- Use case diagrams
- Class diagrams and object diagrams
- State machine diagrams (i.e. statechart diagrams)
- Object constraint language (OCL)
- Interaction diagrams, which include sequence diagrams and communication diagrams
- Activity diagrams
- Development diagrams

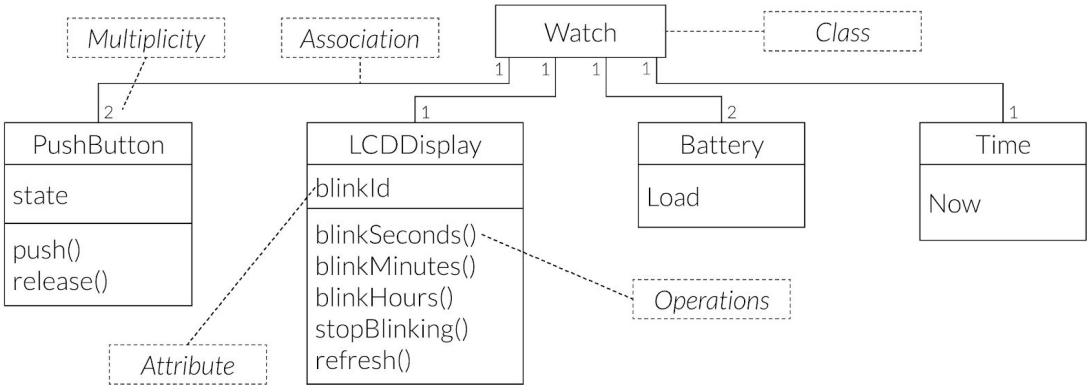
Core conventions:

- All UML diagrams denote graphs of nodes and edges
  - Nodes are entities, drawn as **rectangles** or **ovals**
  - Rectangles represent **classes** or **instances**
  - Ovals represent **functions**
- Names of **classes** are **not** underlined
- Names of **instances** are underlined
- An **edge** between two nodes denotes some **relationship** between the corresponding entities

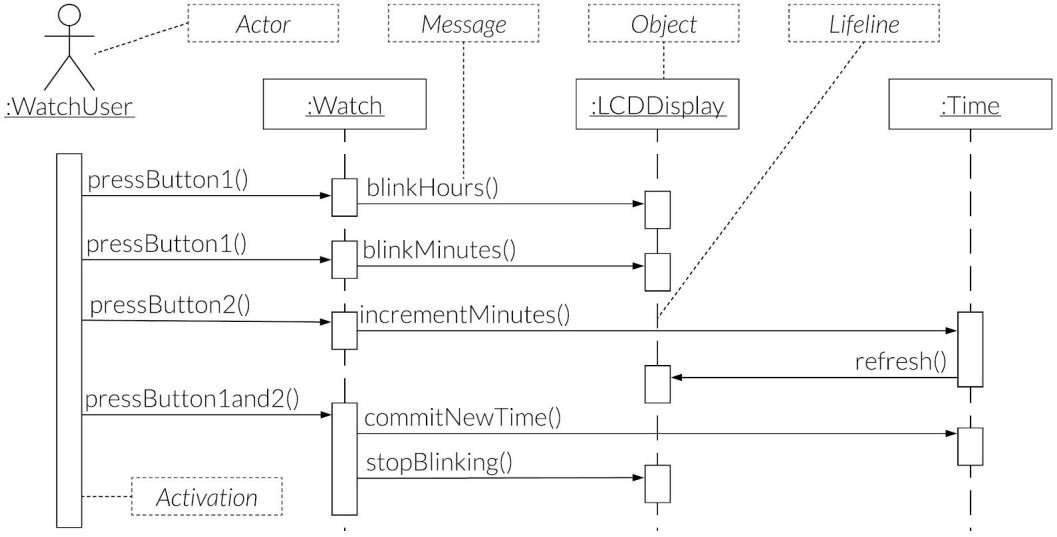
**Use case diagrams** represent the functionality of a system **from the user's point of view**.



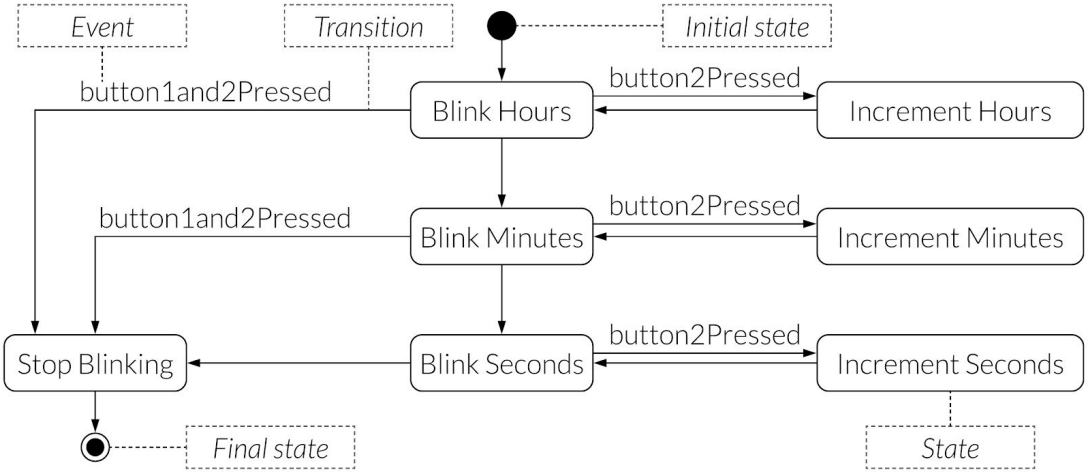

**Class diagrams** describe the **structure** of the system, the **entities** involved in it, and the **relationships** between these entities.



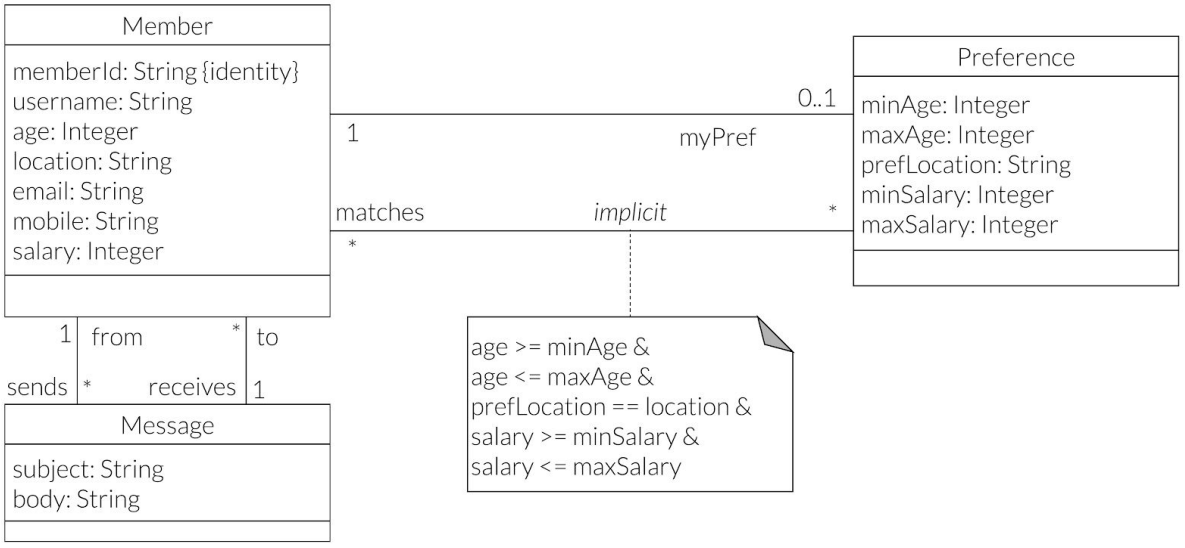
**Sequence diagrams** represent the **behaviour of a system** as messages ("interactions") between *different* objects.




**State machine diagrams** (or **statecharts**) describe the **behaviour of a single object** and the execution steps of each of its operations.




**Object constraint language (OCL)** enables developers to describe **properties** of classes, **associations** and state machines in detail, using a semi-formal language.



**Use cases** and **class diagrams** are used at the earliest development stage of a system to define the services that the system will provide to different groups of users, and the data it needs to process.
Subsequently, the **use cases** can be **refined** into more detailed descriptions of processing, using **state machines** or **interaction/sequence diagrams**; class diagrams then need to be updated to include the functionality details that the use cases include.

## Use cases

**Use cases** are a scenario-based technique in the UML, which identify the actors in an interaction and which describe the interaction itself. i.e. a use case is a set of scenarios tied together by a common user goal (that might succeed or fail).

A **set of use cases** should describe **all possible interactions** with the system.

An **actor** models an **external entity** which communicates with the system. It can be a user, an external system, the physical environment, etc.

An actor has a unique name and an optional description. Examples:
- Client: a person on the train
- GPS satellite: an external system that provides the system with GPS coordinates, etc.

A use case represents a **kind of task** provided by the system as an event flow. A use case consists of:
- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

However, there is no standard way to write a use case, and different formats work well in different cases.

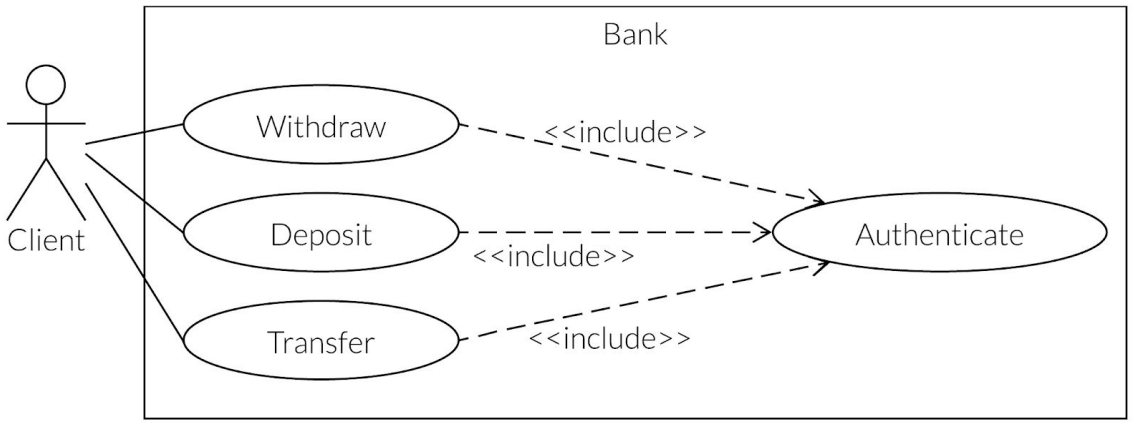
A use case contains the following elements:
- A **primary actor**;
- A **pre-condition** (or **entry condition**) describes what the system should ensure is true before it allows the use case to begin;
- A **guarantee** (or **exit condition**) describes what the system will ensure at the end of the use case; success is guaranteed after a successful scenario;
- A **trigger** specifies the event that gets the use case started.

The **primary actor** calls on the system to deliver a service. Therefore, the use case tries to satisfy the actor with the goal. Usually, but not always, the primary actor is the **initiator** of the use case.
There may be other actors (**secondary actors**) with which the system communicates while carrying out the use case.

Each step in a use case is an element of the interaction between an actor and the system. Each step should be a **simple statement**, show **who** is carrying out the step, and show the **intent** of the actor (not the mechanics of what the actor does).
The user interface is **not** described in the use case, Writing the use case usually precedes designing the user interface.

The **`<<include>>` stereotype** can be used to factorise **common behaviour** (it reuses use cases)**.**

Stereotypes are a mechanism for classifying or marking model elements and introducing new types of modeling elements; the directed dashed arrows indicate dependency:



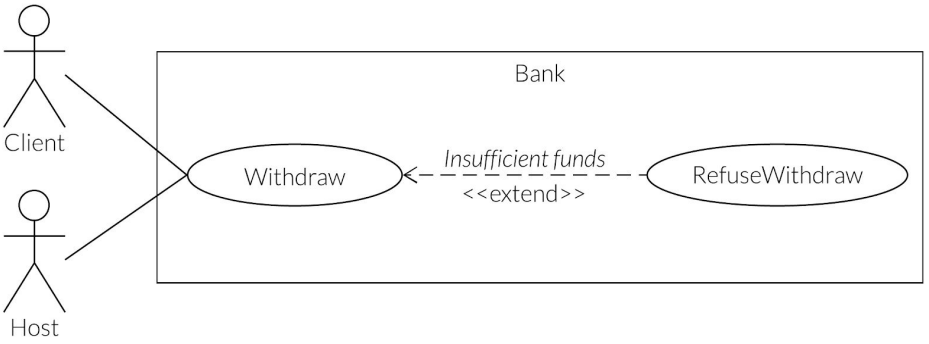

Pros of includes:

- Convenient: no duplicated information in detailed description
- Shorter descriptions
- Common functionality may lead to reusable components
- Enables integration of existing components

Cons of includes:

- May lead to functional decomposition rather than object-oriented model: included use cases can be useful for complex steps that would clutter the main scenario, or steps that are repeated in several cases, but use cases should not be broken down into sub-use cases or subsub-use cases using functional decomposition. That would be a waste of time.
- Requires greater UML knowledge

The **<<extend>> stereotype** can be used to provide **special case**. It factors different behaviour into a single scenario. It is represented using a dashed arrow from the exception to the main case.




**Use cases can also be written in text form.** Pick one of the scenarios as the main success scenarios, and write it as a sequence of numbered steps. Write one of the other scenarios as **extrensions**, describing them in terms of variations of the main success scenarios.
Extensions can be successes or failures, and writing the use case in a text form is a great way of brainstorming alternatives to the main success scenario by asking, for each step: *"What could go wrong?"*, or *"How could this go differently?"*.

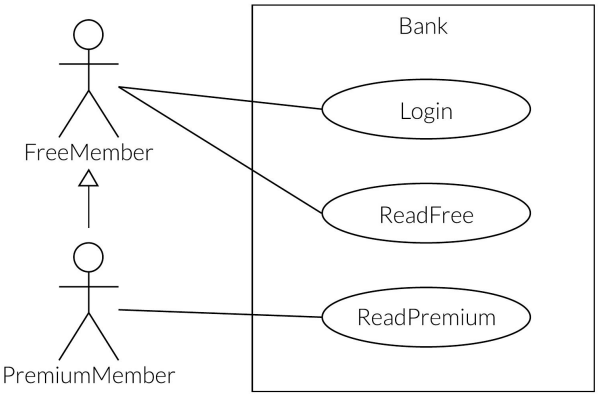
There is a scheme of levels of use cases:

- **Seal-level use cases**: represent a discrete interaction between a primary actor and the system;
- **Fish-level use cases**: are included by the sea-level ones;
- **Kite-level use cases**: show how the seal-level ones fit into wider business interactions.

**System use cases** are interactions with the software (sea- and fish-levels). And the **business use cases** discuss how a business responds to a customer or an event (kite-levels).

**Generalisation** relates two actors or two use cases to each other to factor out ***common*** but not *identical* behaviour.

Example:
*PremiumMember* can do everything *FreeMember* can do




**Steps for creating a use case diagram:**

- Identify actors
- Identify scenarios
- Identify use cases
- Refine use cases
- Identify relationships between actors and use cases
- Identify initial analysis objects
- Identify nonfunctional requirements

### Identifying actors

Actors represent **roles**: an actor is a role that a user plays with respect to the system:

- They carry out use cases: a single actor may perform many use cases, and a use case may have several actors performing it;
- A person can have several roles, and many persons can have the same role;
- In companies, roles usually exist before the system, is built.

Question to ask:

- Which user groups are supported by the system?
- Which user groups execute the systems main functions?
- Which user groups perform secondary functions (maintenance, administration)?
- With what external hardware and software will the system interact?

During the initial stages of actor identification, it is **difficult to distinguish actors from objects**. E.g. is a database system an actor (external software) or an object (part of the internal system)?

Boundaries can be defined to solve the problem: actors are **outside**, objects are **inside**.

### Identifying scenarios

Questions to ask:

- What are the tasks the actor wants the system to perform?
- What information does the actor access?
    - Who creates that data?
    - Can it be modified or removed? By whom?
- Which external changes does the actor need to inform the system about?
    - How often? When?
- Which events does the system need to inform the actor about?
    - With what latency?

Types of scenario:

- **Visionary scenario**: for describing **future** systems (greenfield engineering and reengineering projects) and usually cannot be done by the user or developer alone;
- **As-is scenario**: for describing a current situation; the user describes the system;
- Evaluation scenario: user tasks against which the system is to be evaluated;
- **Training scenario**: step-by-step instructions that guide a novice user through a system.

### Identifying use cases

Scenarios are **generalised** to high-level use cases with **name**, **initiating actor** and a **high-level description**:

- Name: a verb describing what the actor wants to accomplish
- Initiating actor: helps to clarify roles and to identify previously overlooked actors
- High-level description:
    - Entry and exit conditions (identify missing cases)

    - Event flow (define system boundary)
    - Quality requirements (elicit non-functional requirements)

Guidelines to respect when designing use cases:

- The **name** should be a verb that indicates what the actor is trying to accomplish
- The **length** of a use case should not exceed two A4 pages; if it does, use <<include>> relationships
- The **flow of events** should show an active voice, a clear causal relationship between steps and clear boundaries of the system.

# Class diagrams

## Object models

**Object models** describe the system in terms of object classes and their associations.

Modelling is usually done around objects because objects are a natural way to reflect real-world entities. However, more abstract entities are difficult to model using this approach.

Object class **identification** is recognised as a difficult process requiring deep understanding of the problem domain. Object classes reflecting domain entities are **reusable** across systems.

In object-oriented modeling, **objects** are the main unit of abstraction: they can be used for modeling requirements, design and implementation.

In an object-oriented model:

- Objects carry out **activities**;
- Interfaces to objects are **event-oriented**;
- *Example*: A robot has sensors, actuators, control units, etc.

An alternate approach is functional decomposition:

- Decomposition of problem into functions (rather than activities);
- Interface is data-oriented (input/output of functions);
- However, this is typically used only in niche domains;
- *Example*: A compiler has a parser, code generator, etc.

Objects can *interact* with users and other objects. An object has:

- **State**: encapsulated data. Consists of *attributes* or *instance variables*. Part of the state can be *mutable*.
- **Behaviour**: an object reacts to *messages* by changing its state and generating further messages.
- **Identity**: an object is more than a set of values and methods. It has an *existence* and a *name*.

An **interface** defines which messages an object can receive:

- It describes the **behaviour** without describing implementation or state;
- **Public interfaces** (all objects can use them) can often be differentiated from **private interfaces** (only the object itself can use them).

A **class** describes objects with similar structure and behaviour.

```
class Point {
	int x = 0; // attribute (state)
	int y = 0; // attribute (state)

	void move (int dx, int dy) { // method (behaviour)
		x += dx;
		y += dy;
	}
}
```

Advantages of classes:

- **Conceptual**: many objects share similarities (e.g. 10,000+ bank customers);
- Only **one implementation**;
- **Inheritance** or **overriding** methods;
- **Dynamic binding**, where method implementations are determined at run-time;

## Class diagrams

A **class diagram** describes the **kind of objects** in a system and their different **static relationships** (an example of a class diagram can be found in the previous chapter).

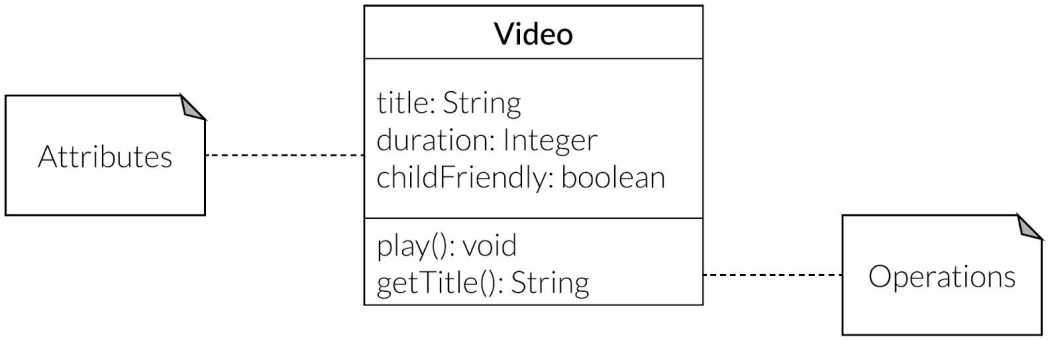
Types of relationships:

- **Associations**: "a user can *watch* videos";
- **Subtypes**: "a nurse is a *type* of person".

Class diagrams are the central model of object oriented analysis, with the largest applicability.

A **class** describes a **set of objects** with **equivalent roles** in a system. It is represented as a square, with a name (the only mandatory information), and optional **attributes** and **operations** (behaviours / methods).

Each **attribute** defines the **state** (data values) of the object and has a *type* (integer/ string). Each **operation** defines how the object affect each other and has a *signature*, stating the **return type** of the method.



An **instance** of an object has an optional name that can contain the class of the instance. If the name exists, it must be underlined, and the attributes are represented with their values.
An **abstract class** is a class that cannot be directly instantiated. Instead, an instance of a subclass is instantiated. Typically, an abstract class has one or more operations that are abstract. An abstract class or operation can be indicated by italicizing the name or by using the label `:{abstract}`.

An **interface** is a class that has no implementation, i.e. all its features are abstract. The keyword is `<<interface>>`.

### Attributes

The full syntax of an attribute is:
`visibility name : type multiplicity = default {property-string}`
E.g. `- name : String [1] = "Untitled" {readOnly}`

`name`: the only mandatory information about the attribute.
`visibility`: indicates if the attribute is public (**+**), private (**-**), package (**~**) or protected (**#**).
`type`: indicates a restriction of the kind of object that can be placed in the attribute.
`multiplicity`: see below.
`default`: the value for a newly created object if it isn't specified during creation.
`{property-string}`: used to indicate additional properties for the attribute:

- `{readOnly}` indicates clients may not modify the property
- `{frozen}` indicates that the attribute is immutable
- If it's missing, we can assume that the attribute is modifiable

The `multiplicity` of a property is an indication of how many objects may fill the property:

- An exact number (1, 2, 3, etc.)
- Arbitrary number: * (zero or more)
- A range: 1..5, 8..*

The following terms can be used to refer to the multiplicity of an attribute:

- **Optional**: implies a lower bound of 0;
- **Mandatory**: implies a lower bound of 1 or more;
- **Single-valued**: implies an upper bound of 1;
- **Multi-valued**: implies an upper bound of more than 1, usually *.

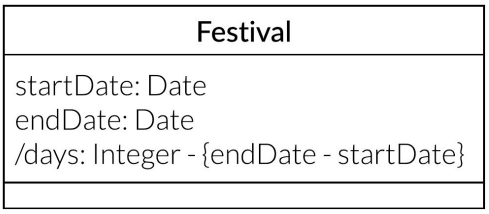
By default, the elements in a multi-valued multiplicity form a **set**, so they are **unordered and unique**:

- If **ordering** of items is important (e.g. a list of train stations), add `ordered` at the end;
- To allow **duplicates**, add `nonunique`.

**Derived properties** can be calculated based on other properties. Notation: `/name --{constraint}`

- `/` indicates that the property is derived

- `{constraint}` specifies how to calculate the value
- E.g. we can calculate the number of days an event lasts if we know the start and end dates:




### Operations

Operations are the actions that a class can carry out. They correspond to the method of a class and the ones that simply manipulate properties are often not shown because they can be inferred (e.g. `setValue()` or `incrementNumber()`).

The full syntax of an operation is:
`visibility name (parameter-list) : return-type {property-string}`
E.g.: `+ balanceOn (date: Date) : Money`

Nothing here is new, except for the `parameter-list`, which is a list of parameters for the operation. These have a name, type, and optional default value.

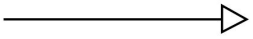
### Relations

Objects and classes can have a range of **relationships** with other objects and classes.

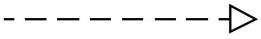
- **Association**: describes a connection between objects



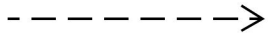
- **Generalisation** / **Inheritance**: a relationship between a more general description and a more specific variety



- **Realisation**: a relationship between a specification and its implementation (the arrow is drawn from the class that defines the functionality of the class that implements the function)

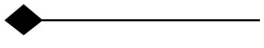


- **Dependency**: a relationship between two model elements. **Usage** is a special type of dependency and means that class A uses class B.



- **Aggregation**: a special case of association denoting a "consists of" hierarchy (e.g. the class "library" is made up of one or more "book" objects)

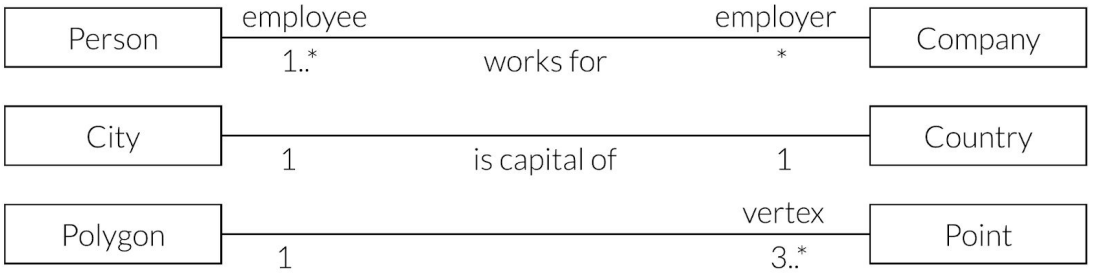
- **Composition**: a strong form of aggregation, where the lifetime of the component(s) is controlled by the aggregate



***Associations***

Associations describe the **roles** played by objects in the relationship. Annotations on them are optional, but make class diagrams **easier to understand**, without **semantic consequences** (they don't affect implementation).

**Multiplicities** of an association end denotes how many objects the source object can reference.




Associations can be **directed**, with an arrow head indicating the direction of the relationship. They can also be **bidirectional**:


- `Car` has a property `owner:Person[1]`
- `Person` has a property `cars:Car[*]`

***Generalisation***

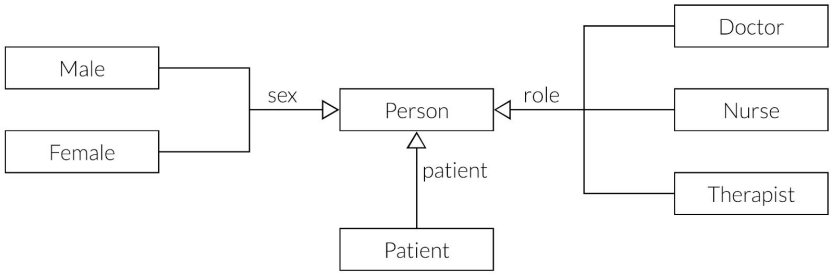
Generalisation expresses a "kind-of"/"is a" relationship, between a general entity (superclass) and a specific entity (subclass) and it simplifies the model by eliminating redundancy. It is implemented by **inheritance**. The child class inherits the attributes and operations of the parent class.

**Classification** refers to the relationship between an object and its type: it is a type assignment. It is denoted the same as a generalisation relationship.

- **Single classification**: an object belongs to a single type, which may inherit from supertypes
- **Multiple classification**: an object may be described by several types that are not necessarily connected by inheritance:
    - **Multiple inheritance**: a type may have supertypes but a single type must be defined for each object.
    - **Multiple classification**: allows multiple types for an object without defining a specific type for the purpose.
- **Dynamic classification**: allows objects to change class within the subtyping structure
    - Multiple dynamic classification is useful for conceptual modeling
    - UML diagrams usually only needs single static classification, because this can be too much.

For multiple classification, UML allows us to specify which combinations are legal by:

- Placing each generalisation relationship in a **generalisation set**
- Labeling the generalisation arrowhead with the name of the generalisation set




Generalisation sets are **disjoint**: an instance can be of only one of the subtypes within the set (a person cannot be both a doctor and nurse, but can be simultaneously a male and a nurse).

***Dependency***

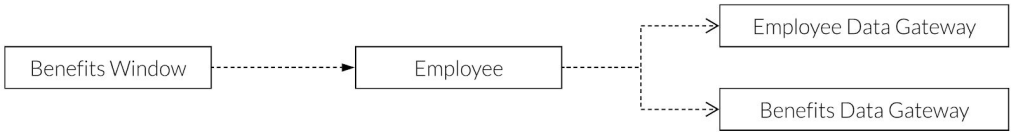
A **dependency** relation exists between two elements: the **supplier** (target) and the **client** (source).

With classes, dependencies exist for various reasons:

- one class sends a message to another,
- one class has another as part of its data,
- one class mentions another as a parameter to an operation.

UML allows one to depict dependencies between all sorts of elements, to show how changes in one element might alter some other elements.

Example:




The `Benefits Window` (a user interface, or **presentation** class) is dependent on the `Employee` class: a **domain object** that captures the essential behaviour of the system (in this case, **business** rules). This means that if the `Employee` class changes its interface, the `Benefits Window` may have to change as well.

Dependency is only in one direction, so we can freely alter the user interface without having any effect on `Employee`.

UML has many varieties of dependency, each with particular semantics and keywords. However, **dependencies should be minimised** - it's bad practice to have many cross-dependencies. **Being selective** is useful, as it means that dependencies can be shown only when they are directly relevant to a topic.

***Aggregation***

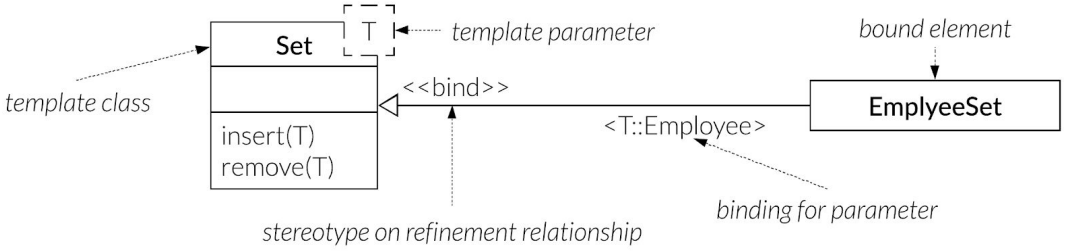
Aggregation is a special case of association denoting a "consists of" hierarchy. The **aggregate** is the *parent class*, and the **components** are the *children classes*.

***Composition***

Composition is a strong form of aggregation, which expresses that a component **cannot exist without the aggregate**. When the "whole" is deleted, so are the parts.

### Template classes

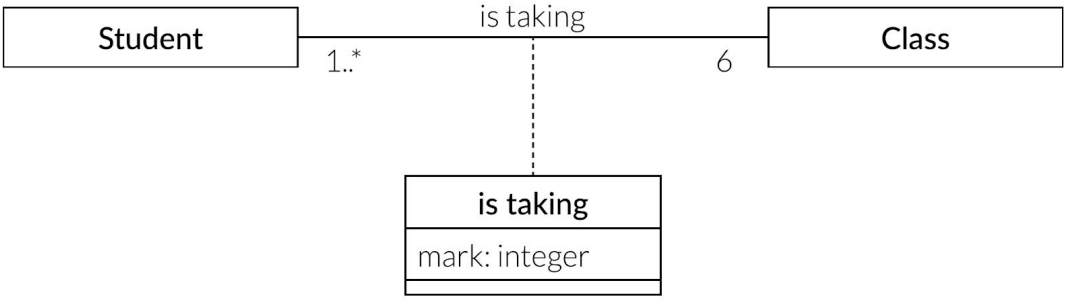
Several languages (including C++) have a notion of a **template** class, which is useful for working with collections in a strongly-typed language. E.g. a `Set` class can be used as a template by defining the behaviour for a general element type.




The `Set` template here specifies operations for a general type `T`; the `EmployeeSet` is a derivation that binds `T` to a specific type, `Employee`.

### Association classes

Association classes allow the addition of attributes, operations, and other features to associations.




It can be replaced by multiple associations, although that adds extra constraints by having multiplicities between the two participating objects.

### Constraint rules

**Constraints** can annotate any part of a class diagram to detail additional rules not captured by other parts of the diagram. These can be written in natural language, a programming language, or UML's Object Constraint Language (OCL).

### Packages

A **package** is a mechanism for organising elements into groups, typically used to increase the readability of large class diagrams. They **decompose** complex systems into subsystems, where each subsystem is a separate package

## How to develop a class diagram

The main goal when developing a class diagram is to **find useful abstractions**.

Things to do to create a class diagram (can be done in any order):

- Identify classes
- Find the attributes
- Find the operations
- Find the associations between classes
- **Iterate** to get the model right and detailed.

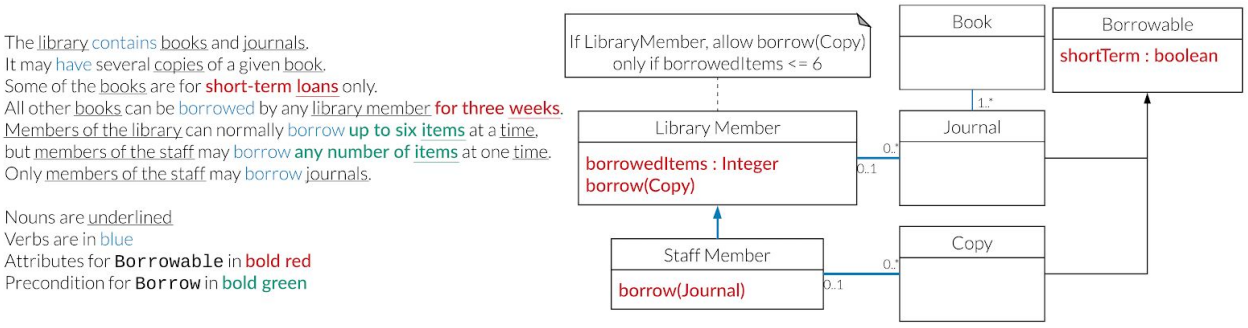
Approaches to class identification:

- **Application domain approach**: ask application domain expert to identify relevant abstractions;
- **Syntactic approach**: extract participating objects from flow of events in (already developed) use cases; use *noun-verb* analysis to identify components of the object model;
- **Design patterns approach**: use reusable design patterns;
- **Component-based approach**: identify existing solution classes.

There is a two-step procedure for identifying classes as *nouns*:

1) **Identify** possible classes. These are the nouns and noun-phrases used in the requirements analysis. Use the singular.
2) **Consolidate** the results. Delete all the ones that have a name that is:
    - **redundant** (one of many equivalent names)
    - **unclear** (alternative: further clarify)
    - an **event** or an **operation** (without state, behaviour and identity)
    - a simple **attribute**
    - **outside the scope of the system** (e.g. a library system probably doesn't require a library class; the objects **are** the system)

Example:




There are problems with the *noun-verb* analysis because natural language is **imprecise** and there can be more nouns than relevant classes. An alternative approach is to use **CRC cards**:

- **Class**: name;
- **Responsibilities**: of objects of the class;
- **Collaborators**: helpers that aid in fulfilling the responsibilities.

If there are too many responsibilities or collaborators, create a new card. Distribute the card to the developing team, and choose a use case scenario: play the scenario through, switching between the collaborators. In doing so, you can discover missing responsibilities or collaborators. Afterwards, add attributes and methods.

| LibraryMember | |
|---|---|
| Responsibilities | Collaborators |
| Maintain data about copies currently borrowed<br>Meet requests to borrow and return copies | Copy |

| Copy | |
|---|---|
| Responsibilities | Collaborators |
| Maintain data about a particular book copy<br>Inform corresponding Book when borrowed and returned | Book |

| Book | |
|---|---|
| Responsibilities | Collaborators |
| Maintain data about one book<br>Know whether there are borrowable books | |

Different kinds of objects:

- **Entity objects**:
  - Represent the persistent information tracked by the system.
  - Application domain objects, "business objects".
- **Boundary objects**
  - Represent the interaction between the user and the system.
- **Control objects**
  - Represent the control tasks performed by the system.

Distinguishing between these three kinds of objects can make models more **resilient to change**.

**Entity** objects are:

- **Recurring nouns** in the use case (e.g. card);
- **Real-world entities** or **processes** that they system must track (e.g. cash dispenser)
- **Data sources** or **sinks** (e.g. database).

**Boundary** objects:

- **Collect information** from actors (e.g. command line interface, html forms);
- **Translate information** into a format for entity and control objects (e.g. cash dispenser);
- Boundary objects **do not** model details and visual aspects (e.g. menu item, scrollbar);
- Each actor interacts with at least one boundary object.

Heuristics for identifying boundary objects:

- **User interface controls** to initiate the Use Case (e.g. bank card);
- **Forms** to enter data (e.g. option screen);
- **Messages** the system uses to respond (e.g. termination message).

**Control** objects:

- **Coordinate** boundary and entity objects;
- Usually **do not** have a concrete **real world counterpart**;
- Are usually created at the beginning of use cases and exist until its end;
- Collect information from boundary objects and dispatch it to entity objects;
- E.g. sequencing of forms, undo and history queues

## The object constraint language (OCL)

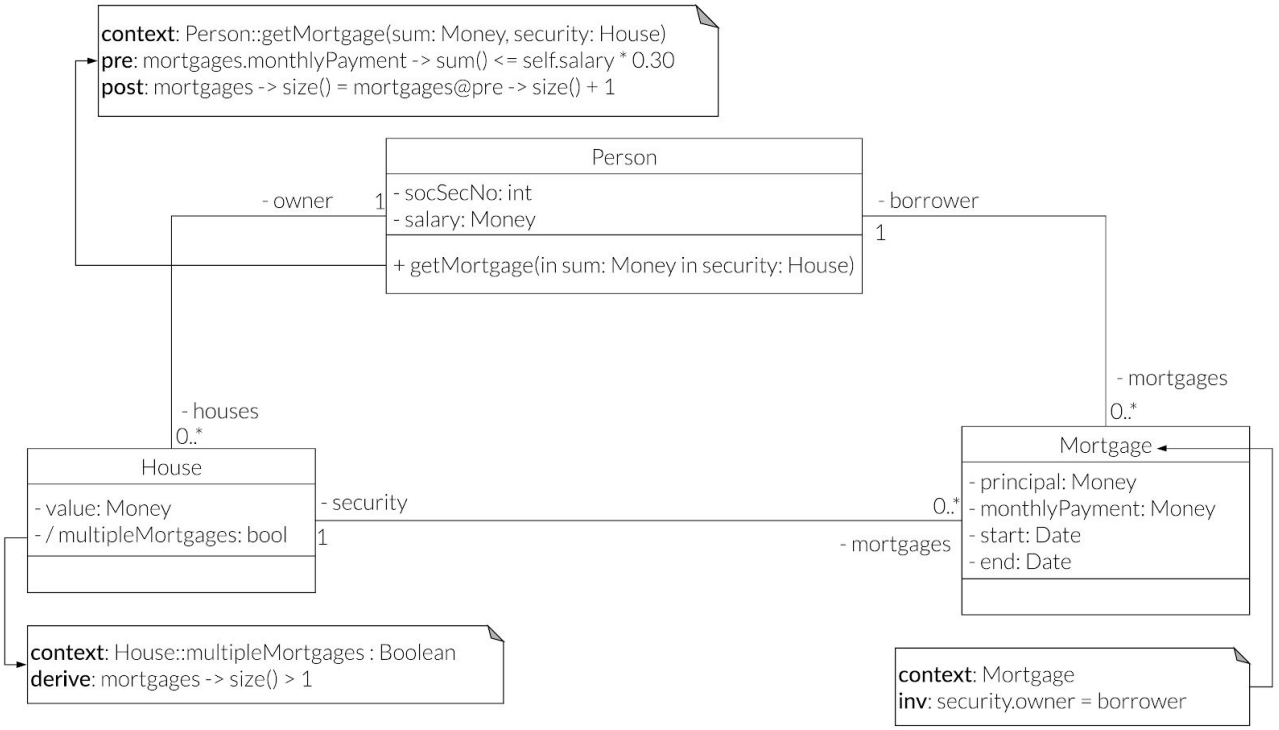
**OCL** expressions allow one to add information that often cannot be expressed in a diagram:

- **Constraints**: restriction on one or more values of an object-oriented model or system;
- **Other information**: defining queries, referencing values, stating conditions and business rules in a model, etc.

OCL expressions can be written in a clear and unambiguous manner. OCL is a constraint and query language at the same time (it has the same capabilities as SQL).

For example, we can specify that for a mortgage:

- You can only get a mortgage for a house that you own.
- The social security number of each person must be unique.
- A new mortgage requires a sufficient income.

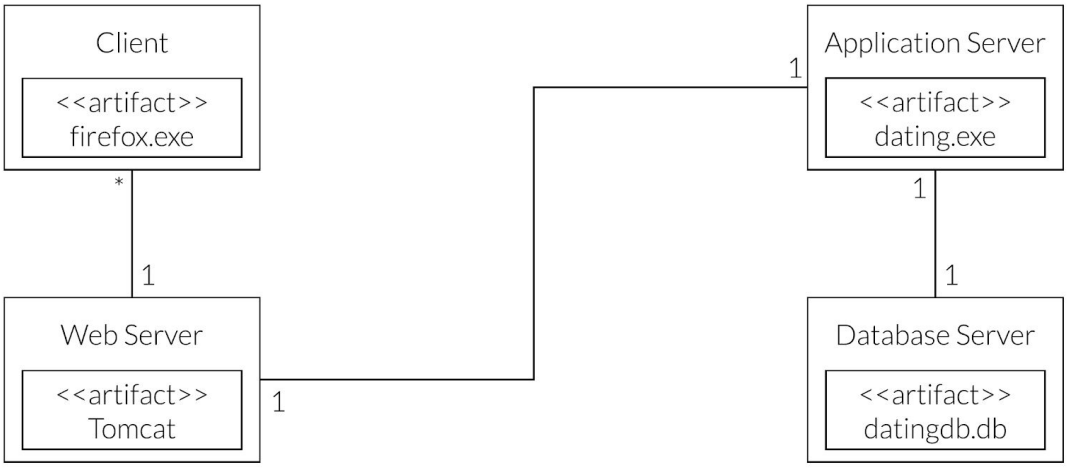


# 3. Web Application Development

**Web applications** are software systems that use the world wide web to interact with users and provide **additional functionality** to a simple web page.

Data is usually input by users filling in and submitting **HMTL forms**. The output/computation is displayed as a web page inside a **web browser** (e.g. IE, Safari, Chrome, Firefox). Increasingly, access to web applications is done through **mobile platforms**. Most web page traffic nowadays is from mobile devices.

Typical structure of a web application (in this case, a dating agency, as a **deployment diagram**):




In a web application, clients are **heterogeneous**: they are likely to be accessing the web application with different browsers, from different machines.

The **web server** hosts web software (e.g. **Apache**, **Tomcat** or **Resin**) to handle web requests, and directs them to the appropriate component of the **web application**. These components may be running on a further machine (the **application server**) and will usually process data in a database which runs on a **database server**.

It is possible for application, database and web server machines to be physically the same machine, but for large systems, they will often be distinct for **efficiency and security**.

**Model-driven development** is relevant to web application development because there are many **rapidly-evolving technologies** (such as application platforms and programming languages) involved in these applications, which create a corresponding obligation for web applications to be **flexible** and **easily upgraded** to these new or enhanced technologies.

Web applications often have **common structure** and elements (such as a relational database, and the need to generate HTML pages to present the UI of the application), which means that a systematic development process can be applied for these applications.

Development of web applications involves three forms of development:

1. Development of **software** that:
    - **receives information** from users (the clients in an internet interaction),
    - **processes information** (usually on the server side of the web application, where databases and other critical resources of the system reside), and
    - **returns information** to clients.
2. Development of **visual appearance** of web pages interfacing to clients.
3. Deciding on **information content** of web pages, choice of words to use, and what information to emphasise.

MDD and MDA apply directly to the first of these, the others require specialised development techniques based on **HCI** and **usability analysis** (not covered in this course).

**Important properties** for web applications:

- Portability
- Usability
- Accessibility

**Portability** means that **a system can be moved** to different execution environments and **provide the same functionality** in the new environment as in the old. This is an important issue for the **user interface** of the application: ideally, the web pages of the system should appear in a similar way and provide identical behaviour if **viewed in any browser**, but in practice there are often differences between the way that different browsers render web pages. This is why **developers should test** their web pages on the main browsers, as well as the respective mobile versions, and check that they look and behave as expected.

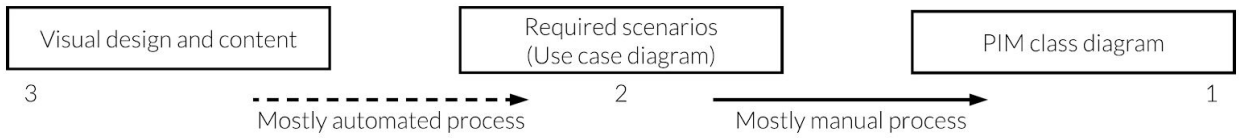
**Usability** means that **a web interface does not require unreasonable effort** to use, and that users can access provided functionality without excessive effort.
Usual principles of usability of UIs:

- **clear information** should be provided,
- **feedback** should be provided immediately after data entry,
- **related functions** should be grouped together.

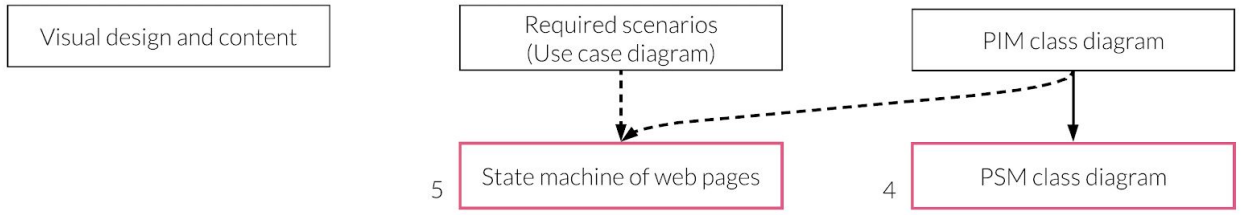
Web-specific usability guidelines include that **length of navigation paths** between parts of an interface used by the same user in same session should be minimised.

**Accessibility** means that **a web interface can be used by users of differing ability** — such as visually impaired, colour-blind, deaf, or senior citizen users — as effectively as by non-disabled, technological skilled users. Tools can be used to preview sites to show them as (for example) a colour-blind viewer would see them, thus helping developers avoid colour choices which would be unusable to colour-blind viewers.

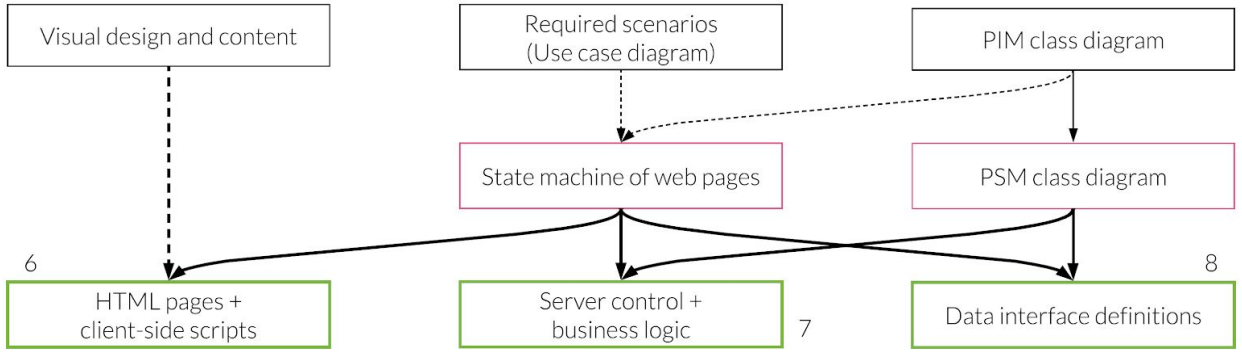
General **MDA development process** for web applications:




Begin with a **PIM** (1), and the required **scenarios** of the website (2).
The **visual design** and content should also be sketched out based on the possible scenarios (3). These should usually be consistent in style across an application, and so general design decision can be made without too much detail of the rest of the system.



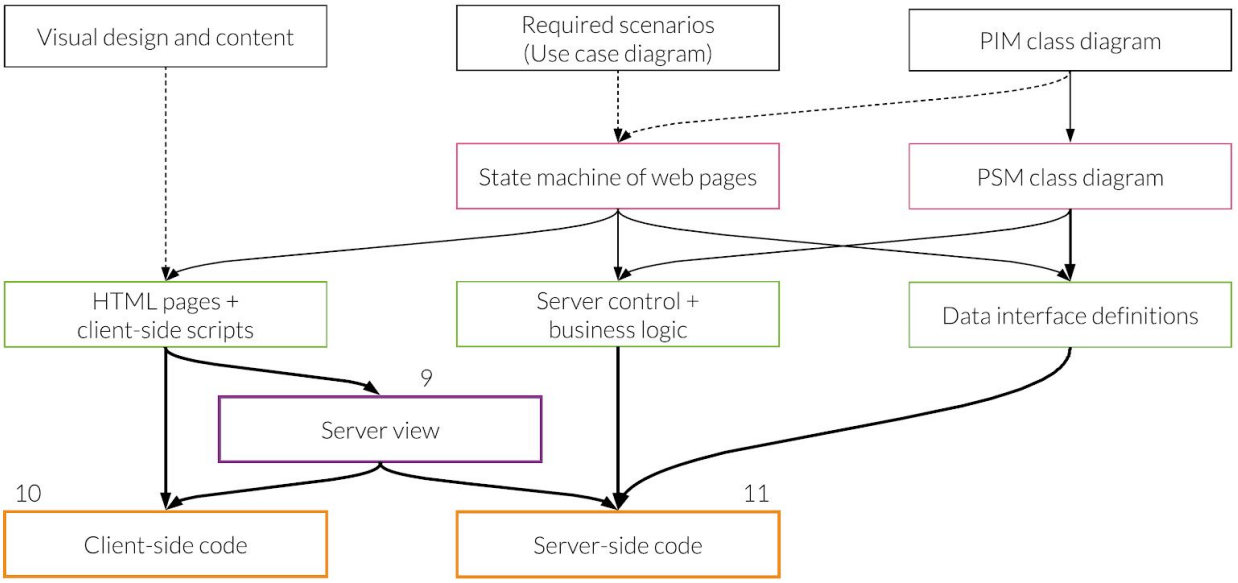

PIM class diagrams can be **refined** into **PSM** class diagrams using **model transformations** (4).
The **state machine** can be created, informed by the class and use case diagrams (5).




**HTML pages** can designed from the states of the state diagram (6), their style informed by visual design decisions.
The **internal business rules** can be decided upon based on the behaviour and structure of the system, defined in their respective UML diagrams (7).
The necessary persistent data for the system can be identified, and so **data interfaces** can be defined (8).

From the code of the HTML pages, the required **server-side components** to generate them can be designed (9).

Then, we are ready to **automatically generate** the majority of the implementation (10, 11).



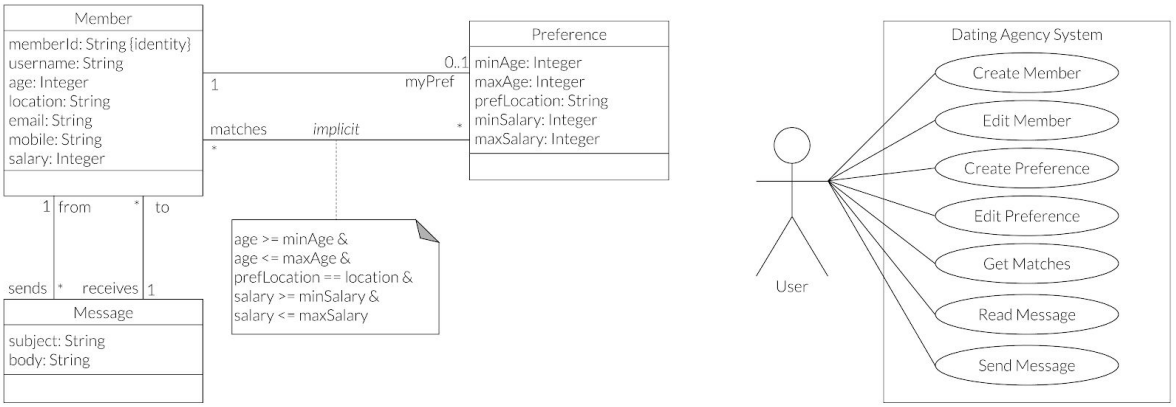

## Web application specification

The specification of the functionality of a web application typically consists of **two platform-independent models**:

1. A **class diagram**, showing the **data** to be stored and processed by the system.
2. A **use case diagram**, showing the **operations** which the user will be able to perform upon the system.

The class diagram should also include documentation of any **constraints** on the data.

**Example** for the dating agency application:




**Class diagram specification**:

The **constraint** *age ≥ minAge & age ≤ maxAge & prefLocation = location & salary ≥ minSalary & salary ≤ maxSalary* indicates the condition under which a member matches against another member: their age and salary must be in the preferred ranges of that other member, and their location must be the same as the preferred location.

This constraint defines the *Preference_Member* association: the *m : Member* and *p : Preference* objects linked by the association are exactly those for which

*m.age ≥ p.minAge & m.age ≤ p.maxAge &*

*m.location = p.prefLocation &*
*m.salary ≥ p.minSalary & m.salary ≤ p.maxSalary*

For each member, the set *myPreference.matches* is the set of other members who match the preference of the member (could actually include the member themselves).
Finally, ***<<implicit>>*** means that each association is calculated **on demand**, rather than stored persistently in some database table.

**Use case diagram specification**:
The system allows users to register and record their *details* (age, height, location, etc.) and *preferences* for dating partners (age range, location, etc.). For each user, the system can produce a list of the other users who match the preferences. Advanced features include the ability to *send messages* anonymously via the system, and the automated notification of a user when a new user who matches their requirements becomes a member.

# Web application design

The following design techniques will be covered:
- Web page design
- Interaction sequence design (using state machines)
- Transformation from analysis to design models (using class diagrams)
- Architecture diagrams.

These are independent of particular server-side programming technologies (e.g., JSPs, Servlets, PHP, ASP, etc.).

## Web page design

To design web pages, we can **sketch diagrams** of their intended structure and appearance, and review these for usability, visual consistency, etc. We need to think about how information is **displayed**, as well as how the user will **input information** — this includes navigation and the input of more complex data. The typical data input method for a web page is a **HTML form**.

Design considerations for web pages:
- In forms, use clear and simple **labels** for fields and make clear which are **mandatory**;
- Avoid exposing **internal IDs**, unless these are generally used in the domain:
    - NI numbers for adults, ISBNs for books, etc.
- Where possible, use **default** values if user does not fill in a field;
- Use an appropriate **input type** for the data required;
- **Avoid reloading an entire web page** if only part of it changes. Technologies such as AJAX can be used to achieve partial update of a page;
- A web application should provide clear and immediate **feedback** to user concerning incomplete or incorrect data;
- Think about **usability**.

Some **data validation** and dynamic user **feedback** can be provided on client side of a web application by using **client-side scripting** languages, such as Javascript.

A scripting language is a simplified programming language, designed for tasks such as checking that a string is non-empty, that a string represents a number, etc. **Javascript** code can be written as part of web pages, and executed in client browser when these pages are displayed.

**Example**: if we have the following form:

```
<html>
        <head><title>createMember form</title></head>
        <body>
                <h1>New Member</h1>
                <form action = "http://127.0.0.1:8080/servlets/createMemberServlet" method = "POST">
                        <p><strong>Username:</strong> <input type = "text" name = "username"/></p>
                        <p><strong>Age:</strong> <input type = "text" name = "age"/></p>
                        <p><strong>Location:</strong> <input type = "text" name = "location"/></p>
                        <p><strong>Email:</strong> <input type = "text" name = "email"/></p>
                        <p><strong>Mobile:</strong> <input type = "text" name = "mobile"/></p>
                        <p><strong>Salary:</strong> <input type = "text" name = "salary"/></p>
                        <input type = "submit" value = "Register"/>
                        <input type = "reset" value = "Cancel"/>
                </form>
        </body>
</html>
```

Then we can insert a Javascript code to make sure that the username field is not empty. This would be added between *</title>* and *</head>*:

```
<head><title>createMember form</title>
<script type = "text/javascript">
function checkUsername() {
        if (document.newMember.username.value.length == 0){
                window.alert("User name cannot be empty!");
        }
}
</script>
</head>
```

The *checkUsername* function is invoked whenever a new value is entered into the user name field, and opens up an alert box if the new string is empty. However, a limitation of Javascript is that **it is not necessarily executed by the browser**, because some browsers do not support Javascript or may have Javascript execution explicitly switched off. Therefore **data validation should always be done on server side** in addition to client-side checks.

Javascript can be added to web pages in the following ways:

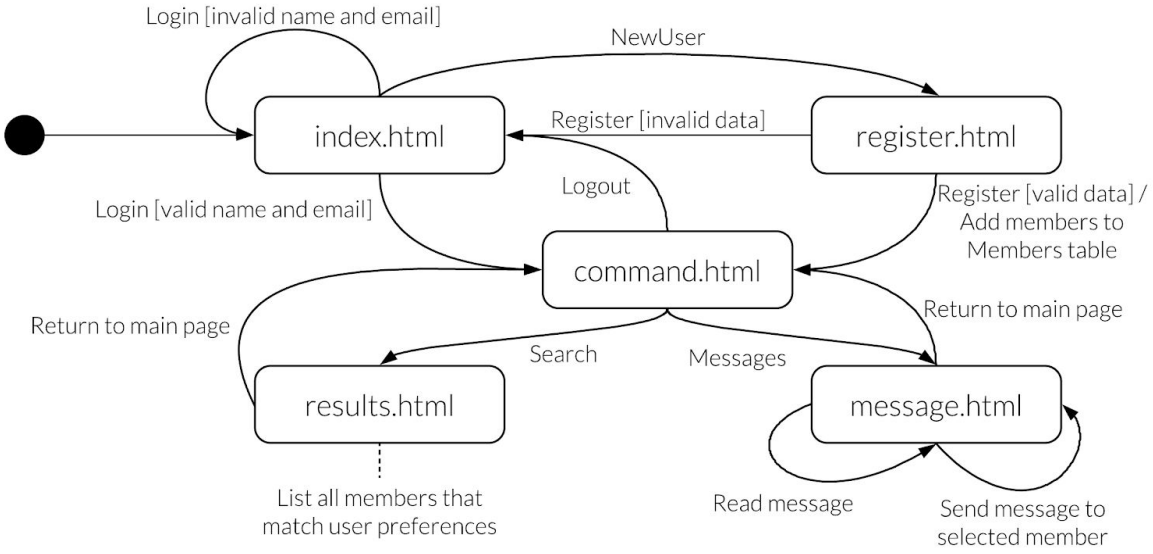
- Embedded in HTML page as *<script>* element
  - Written directly inside the *<script>* element:
    ```
    <script> alert("Hello World!") </script>
    ```
  - Linked file as src attribute of the *<script>* element:
    ```
    <script type = "text/JavaScript" src = "functions.js"></script>
    ```
- Event handler attribute:
  ```
  <a href = "http://www.yahoo.com" onmouseover = "alert('hi');"/>
  ```
- Pseudo-URL referenced by a link:
  ```
  <a href = "JavaScript: alert('You clicked');">Click me</a>
  ```

## Interaction sequence design (using state machines)

**State machine diagrams for web application development** can be used to describe the **interaction** behaviour of a web application: what sequence of pages are displayed to user, and the effect of user commands.

The **states** of a state machine correspond to web pages displayed to user: name of page is given, plus a summary of its content. The **transitions** are labelled with events that correspond to user commands or links that can be selected in source state (web page). The target state of the transition is the next web page shown to user after the command was executed.

Example of an **interaction sequence** for the dating agency:




Interaction sequences can also break down the system interface into separate subsystems / groups of related web pages that can be used at different times, and this is called **phase decomposition**.

**State machine diagrams** represent the **dynamic behaviour** of a system in terms of its internal interactions. They complement the **class diagrams**, which represent **static structure**. They can also inform the design of a class diagram, by helping to find missing objects, and helping to identify operations of classes.

State machine diagrams seek to represent:

- **States**: an abstraction of the attribute values of an object (e.g. the book is currently borrowable);
- **Events**: something that happens at a point in time (e.g. a user's birthday);
- **Actions**: operation in response to an event (e.g. notification of celebration);
- **Activity**: performed as long as an object is in some state (e.g. keep cooling the room as long as it is above the set temperature).

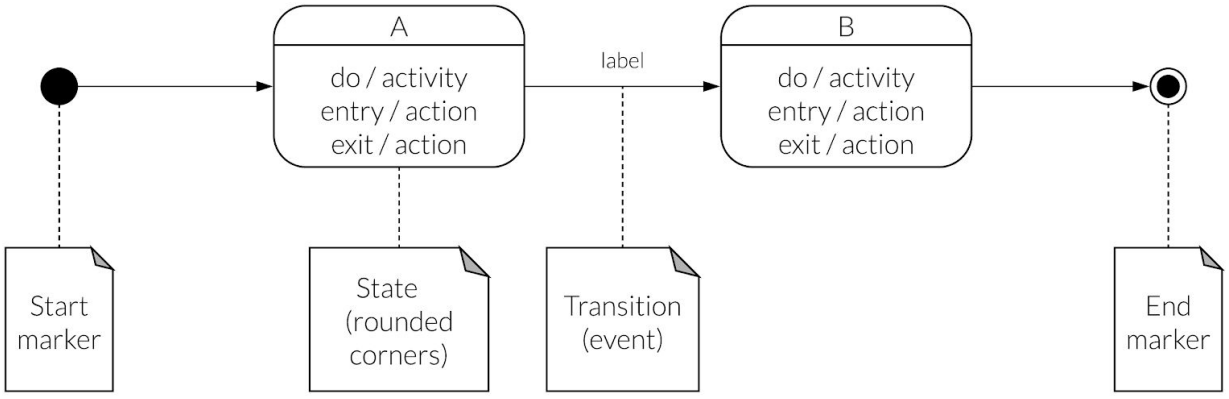
State machine diagrams model **dynamic aspects** of systems:

- What are the **states** of the system?
- Which (internal/external) **events** does the system react to?
- Which **transitions** are possible?
- When are **activities** started and stopped?

They correspond to a **labelled transition system**, $< \Sigma , \Lambda , \delta >$, where:

- $\Sigma$ is the set of all states,
- $\Lambda$ is the set of labels,
- $\delta$ are the transitions, $<s, l, s' >$, s.t. $\{s, s' \} \subseteq \Sigma$ and $l \in \Lambda$

UML state diagrams (statechart):




The **states** represent particular **values** of the objects' attributes and the transitions describe **events, conditions** under which an event occurs, and the **actions** performed in response to that event.

We can **label transitions** with the following information: **Event(par) [condition] / action**:

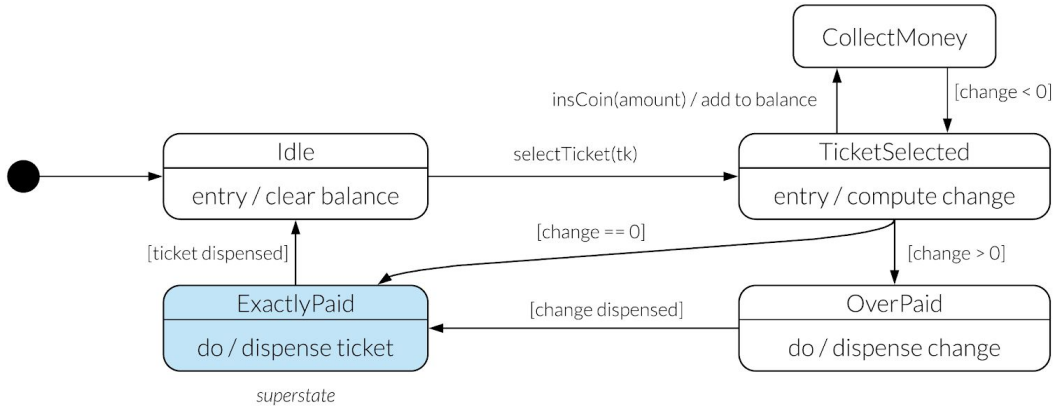
- **Event()** is a descriptive name of what has happened to cause the transition. This can have optional parameters **par**;
- **[condition]** is a description of the conditions under which this transition occur;
- **action** describes what is performed in response to the event.

All these elements are optional, but there should be at least one of them on each transition.

A **state** can contain **activities** (with the **do** keyword) and they happen while the object is in that state; **entry actions** always happen whenever the state is entered, and the **exit actions** always happen whenever the state is exited.

**Activities** in a state diagram can be **composite items** that denote other state diagrams. A set of substates in a nested state diagram can be denoted with a **superstate**.
**Transitions from** other states to the superstate **enter the first substate** of the superstate;
**Transitions to** other states from a superstate are **inherited by all the substates** (state inheritance).

"dispense ticket" as an atomic activity:

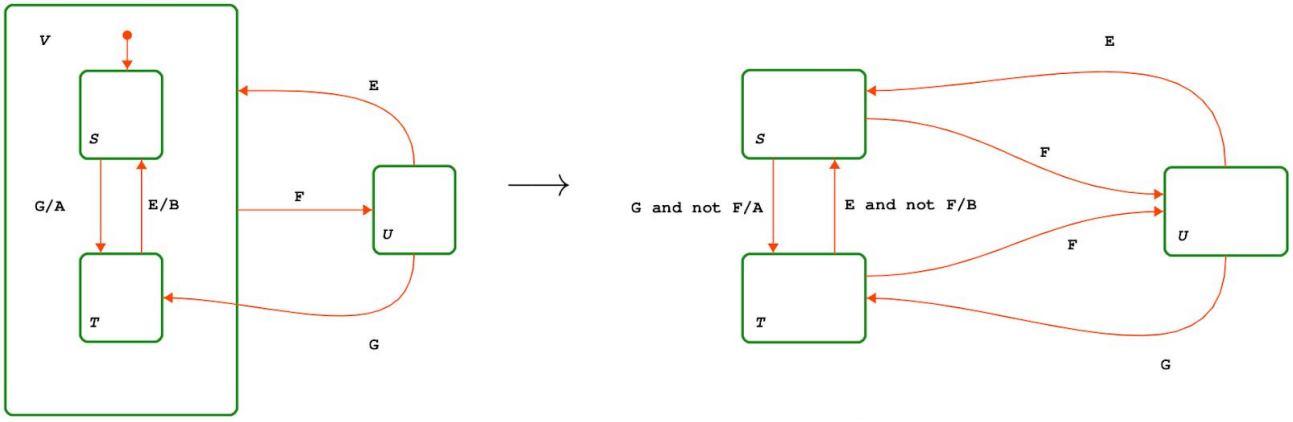



"dispense ticket" as a composite activity:


Simple state machines allow us to represent complex behaviours, but because they are usually hard to read, **Harel diagrams** (an extension to simple state machines) introduce additional notations for representing complex behaviours in a clear way:

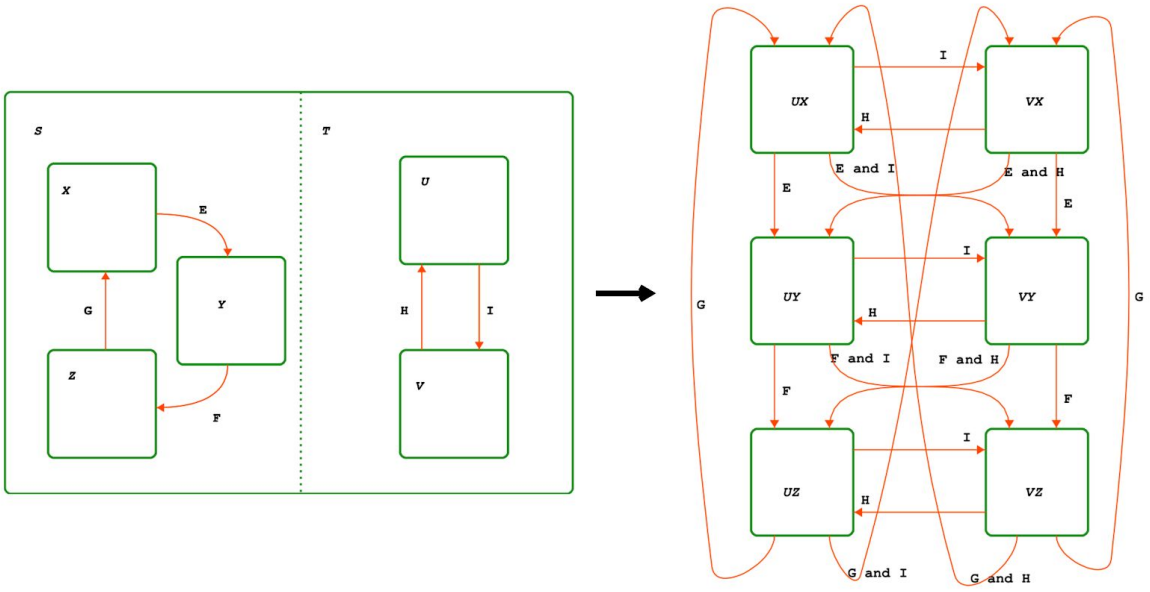
- Hierarchical states
- Parallelism
- History connectors
- Switch connectors

**Hierarchical states**: the system is a **ground state** and is in all parent states. "Higher-level" transitions have priority over "lower-level" transitions.

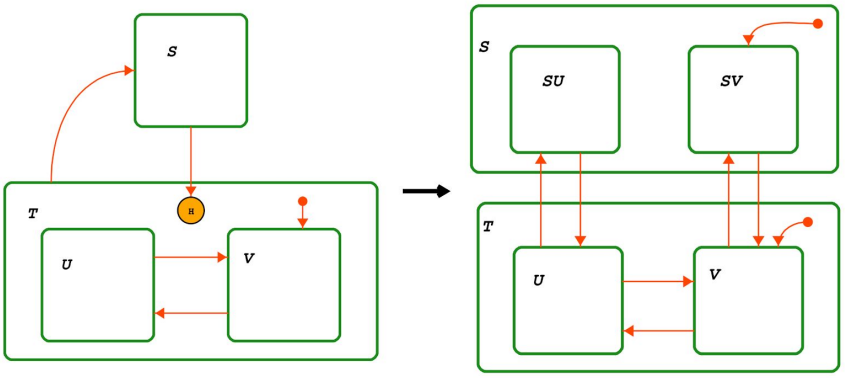



**Parallelism**:

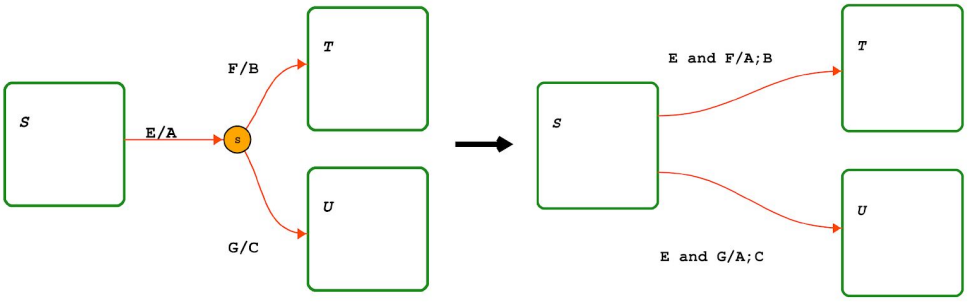
- **OR-states**: the system is in at most one state of current level;
- **AND-states**: the system is simultaneously in all states of current level ⇒ parallelism;




**History connector**: describes which state was last active



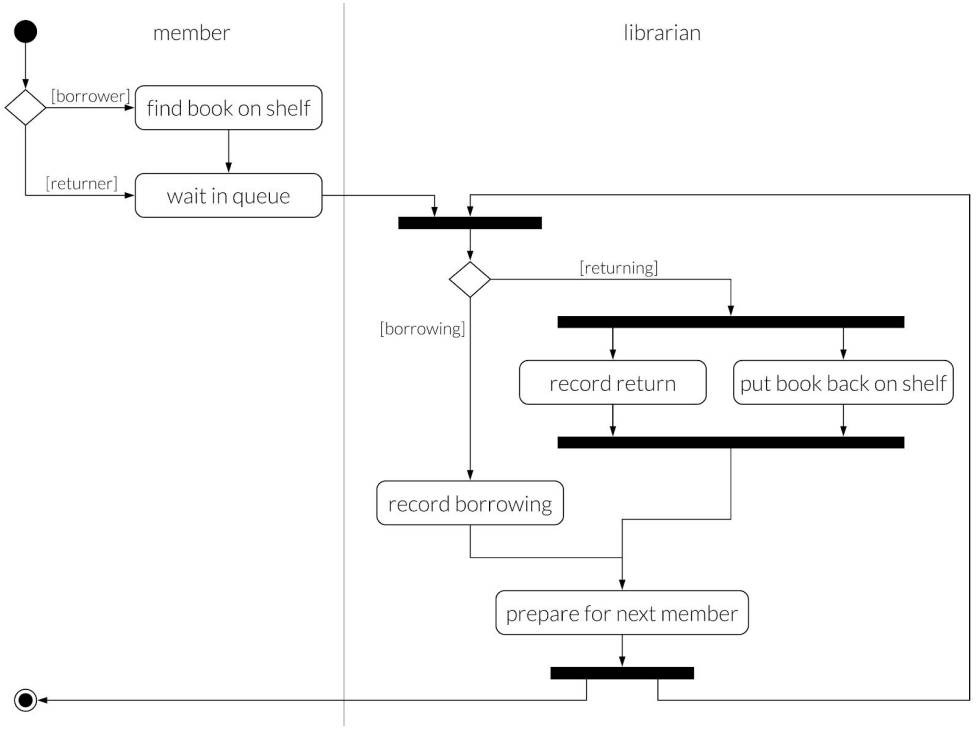

**Switch connector**:




**Activity diagrams** are a simple alternative to **statecharts**. They emphasise the synchronisation **within** and **between** objects. Syntax:

- **Activity**: a kind of state which is left when the activity is finished;
- **Transition**: normally not labelled, since the event is the end of an activity;
- **Synchronisation bars**: describes synchronisation points;
- **Decision diamonds**: shows decisions, alternative to guards;
- **Start and end markers**: like in statecharts;
- **Swim-lanes**: shows which object carries out which activity.

Example of activities in a library:




## Transformation from analysis to design models

Analysis class diagrams need to be **refined** to model for a particular implementation platform (e.g., a relational database). This is special case of PIM to PSM transformation of MDA.

For relational database implementation use transformations:

- **Remove inheritance** by **merging subclasses into their superclass**, or, if we want to represent the classes in separate tables, by **using a *-1 association from sub- to superclass**.
  This transformation is useful when refining a PIM towards a PSM for a platform that does not support inheritance (such as the relational data model). It can also be used to remove multiple inheritance for PSMs that do not support it.

- **Introduce primary keys** for all persistent entities that do not already have an **{identity}** attribute. This is an essential step for implementation of data model into a relational database.
  This refinement transformation applies to any persistent class. If the class does not already have a primary key, it introduces a new identity attribute, usually of Integer or String type, for this class, together with extensions of the constructor of the class, and a new get method to allow idealization and read access of this attribute.
- **Replace explicit many-to-many associations** by two many-to-one associations and an intermediate class.
  Explicit many-many associations **cannot be directly implemented** using foreign keys in a relational database — an intermediary table would need to be used instead. This transformation is the OO-equivalent of introducing such a table.

- **Replace explicit *-1 associations by a foreign key** from the * entity to the 1 entity.
  This refinement transformation applies to any explicit many-one association between persistent classes. It assumes that primary keys already exist for the classes linked by the association. It replaces the association by embedding values of the key of the entity at the 'one' end of the association into the entity at the 'many' end.

The resulting class can be directly translated into database tables.

## Architecture diagrams

Web applications can typically be separated into **five tiers**:

1. **Client** tier
2. **Presentation** tier
3. **Business** tier
4. **Integration** tier
5. **Resource** tier

Each tier has different **responsibilities**, and these should be kept separate where possible.

### Client tier

Consists of **web pages**: either hard coded (**static**) HTML text files or generated (**dynamic**) pages. Static HTML files are downloaded by the browser from the server to the client, while the dynamic pages are produced as a result of an HTTP request to a server-side component, which makes them more **flexible**.
These components implement the **form designs** from the design class diagram/page sketches.

### Presentation tier

Consists of **controller** and **view components**, such as **servlets** and **JSPs**.
It may also contain **helper classes** for web-page generation, and **beans** for temporary data storage and processing.
These components enforce the interaction sequencing defined in the state machine diagrams.

### Business tier

Consists of **beans**:

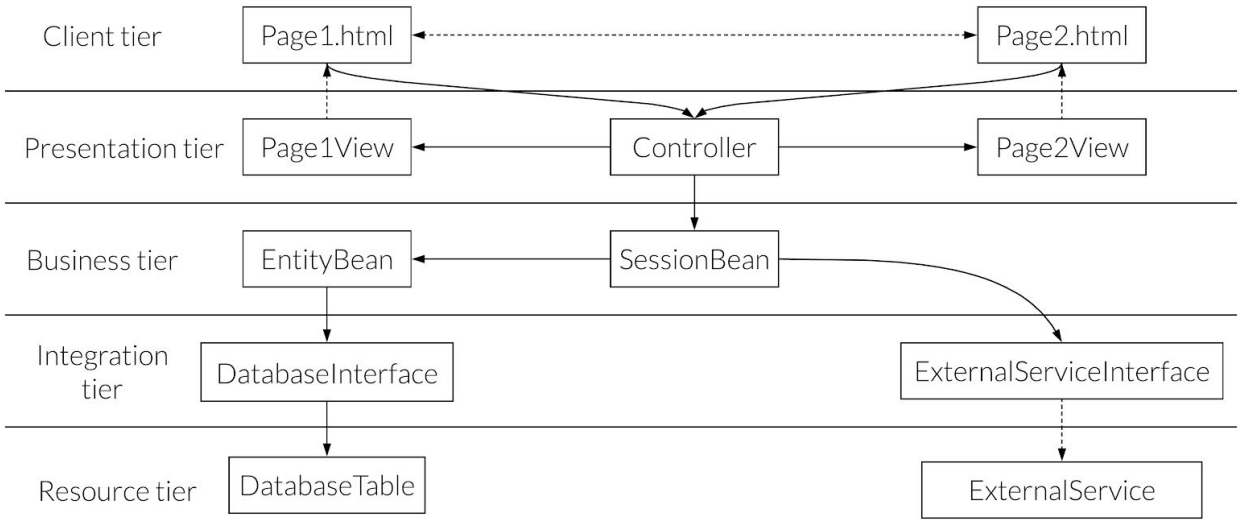
- **session beans**, which represent groups of business functions, used within a single client session; and
- **entity beans**, which represent business and conceptual entities, and persistent data.

### Integration tier

Contains database **interfaces**, interfaces to external web services, etc.

### Resource tier

Contains the **actual** databases, web services and other resources used by the system.
Typical five-tier architecture for a web application:




Meaning of arrows:

- **Solid arrow** from A to B means that A **invokes/forwards an operation/service** of B. Example:
  - A form web page invokes (via the internet) a *doGet* or *doPost* method of a servlet;
  - Servlets invoke methods of page generation classes to build result pages;
  - Methods of a database interface modify/read the database's tables.
- **Dashed arrows** are used for **HTML links** and **generation of web pages**.

**Beans** encapsulate functionality of data in a standardised way. They provide a level of abstraction between raw data and the business logic.

- A **session bean** is created within **a specific client session**. It lasts only as long as the client's session.
- An **entity bean** manages **persistent data across multiple clients**. Typically last longer than a client's session.

**Functional behaviour** of a web application can be carried out both on client side of the system (e.g., by Javascript code executing in the user's web browser) and on server side.
The usual **functional components of a web application** are:

- **Web pages**: written in HTML/XHTML, sometimes with Javascript or other scripting code. They carry out simple processing (e.g. to check validity of form data before submitting it).
- **Controller/coordinator components**: process submitted data and take actions on the server side of a system.
  In Java: **Servlets**. Servlets are Java classes that provide specific operations for receiving and responding to HTTP requests.
- **View/presentation server-side elements**: generate web pages and take actions in response to submitted data.
  In Java: **JSPs (Java Server Pages)**. JSPs are HTML pages enhanced with server-side Java code. They are compiled into servlets by a JSP compiler.
- **Resources**: databases or remote web services which the system uses.

**Server-side processing** is generally responsible for 4 main tasks:

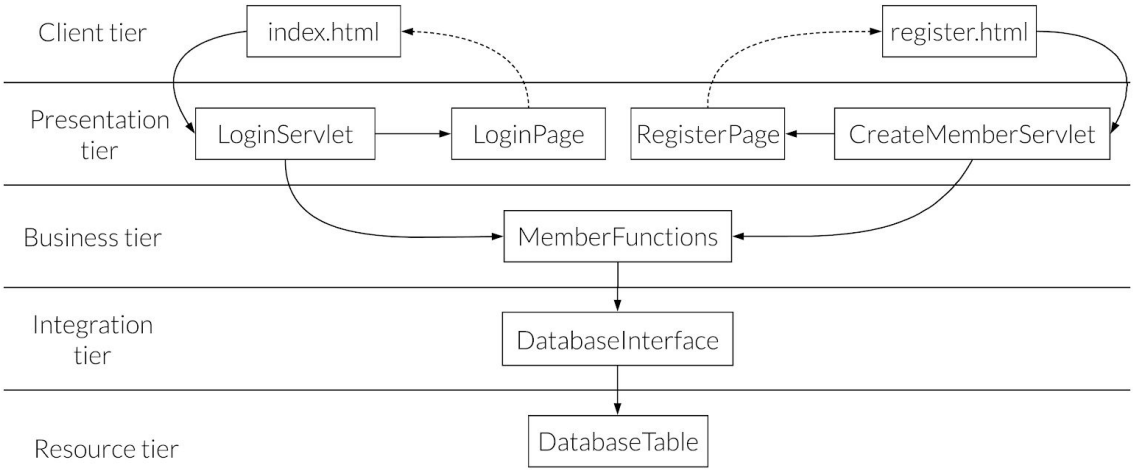
- **Processing data sent from the client side**. This can involve checks on the **correctness** of the data, or **security checks**.
- **Modifying** or **retrieving data in the lower tiers** of the server side, such as a database. Allowing clients to access this data could pose a security threat.
  This checking is most appropriately carried out in business tier components which manage entities: so-called **entity beans**.
- **Invoking operations of lower tiers**, including remote web services.
- **Generating a results web page** to be shown to the client. If not done on the server-side, it puts too much computational load on the client side.

Three alternative architectures can be used to **implement the presentation tier** of a web application **using Java standard edition (JSE)** technologies:

- **Pure Servlet**: servlets respond to requests, call database interface (DBI) or business tier, and use auxiliary classes to generate response pages. This approach has advantage that it needs no JSP compiler.
- **Pure JSP**: JSPs respond to requests, call Database interfaces/business tier and generate response pages.
- **Model-view-controller (MVC)** (Servlet/JSP hybrid): like pure servlet approach, but using JSPs to construct response pages, on redirect from servlets.

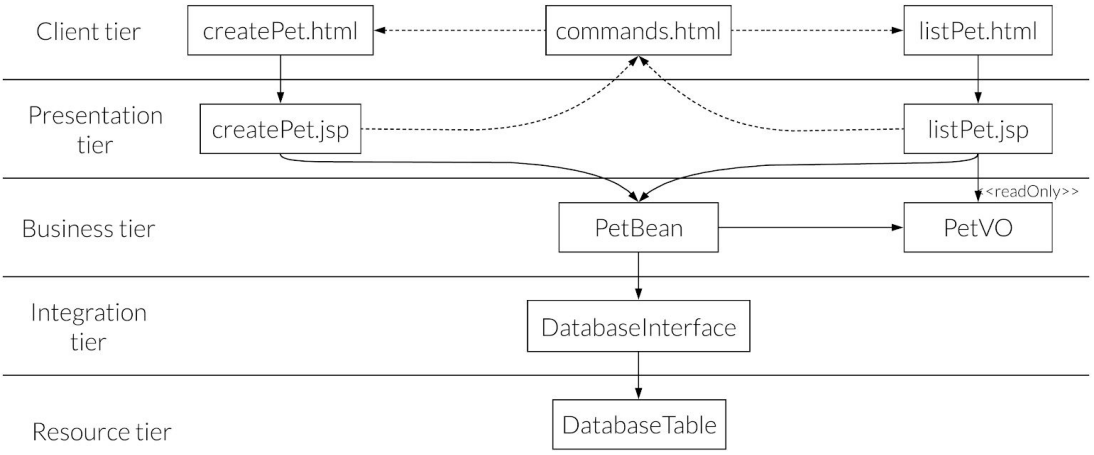
### Pure Servlet

Servlets are based on **low-level** functionality, and so they can be expensive to develop large systems with.
Servlets provide little support for building web applications or dealing with HTTP requests, so a lot of work has to been done manually to create **helper classes**.



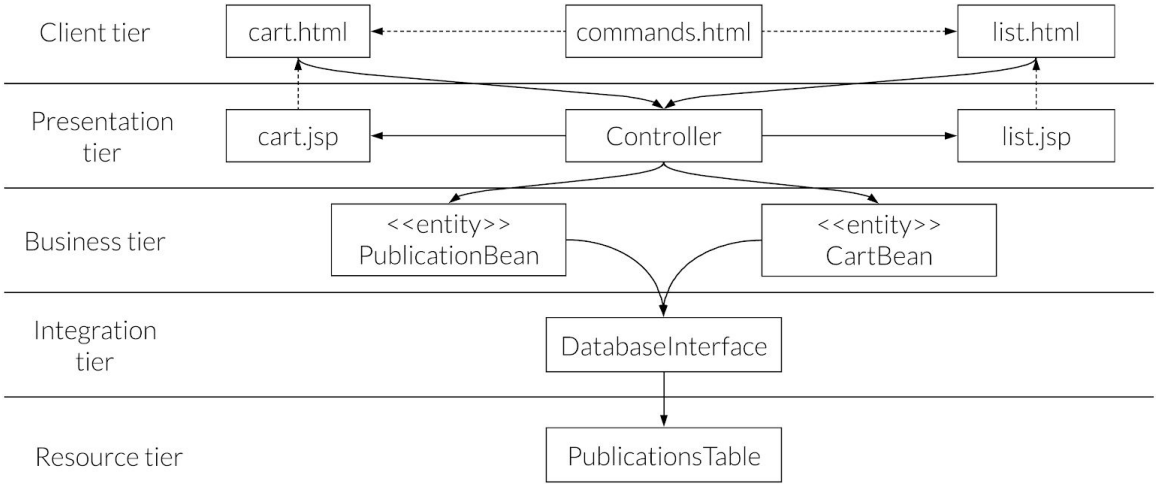
### Pure JSP

Instead of using helper classes such as *LoginPage* or *RegisterPage* to generate result web pages, we can write JSP files, which describe result pages as a mixture of fixed HTML text and dynamically generated text.
Can also separate out database update code into session beans or entity beans invoked from JSPs, manipulating/representing data (eg, instances of entities) being processed.




### MVC Approach

Pure JSP approach can lead to **complicated** programming within JSP, as scriptlets. An alternative is to use a **hybrid** approach where servlets (**controllers**) handle requests, interact with business tier beans (**models**), and create beans for use by JSPs (**views**).
Servlet controllers **decide** which JSP to forward a request to, and generally **control** what sequence of interaction is to be followed. The JSP components are only concerned with **presenting** data as web pages.
This hybrid approach is known as **MVC (Model-View-Controller)** architecture. This approach is more **flexible** and **extensible** than pure JSP or servlet approaches as **responsibilities are separated**, so it is more appropriate for larger and more complex systems.



Controller servlet identifies which command needs to be responded to:

- **Purchase cart**: it forwards to a JSP/web page that asks for credit card details, etc. Resets cart to null.
- **Add to cart**: it finds all products selected by the customer. Gets cart from this customer's session (from the session bean) and adds products to cart. It forwards to cart.jsp to display the cart.

In general, JSPs should be used for components which mainly have a **UI role**, without complex algorithms and decision making. Servlets are appropriate for **control and processing** tasks.

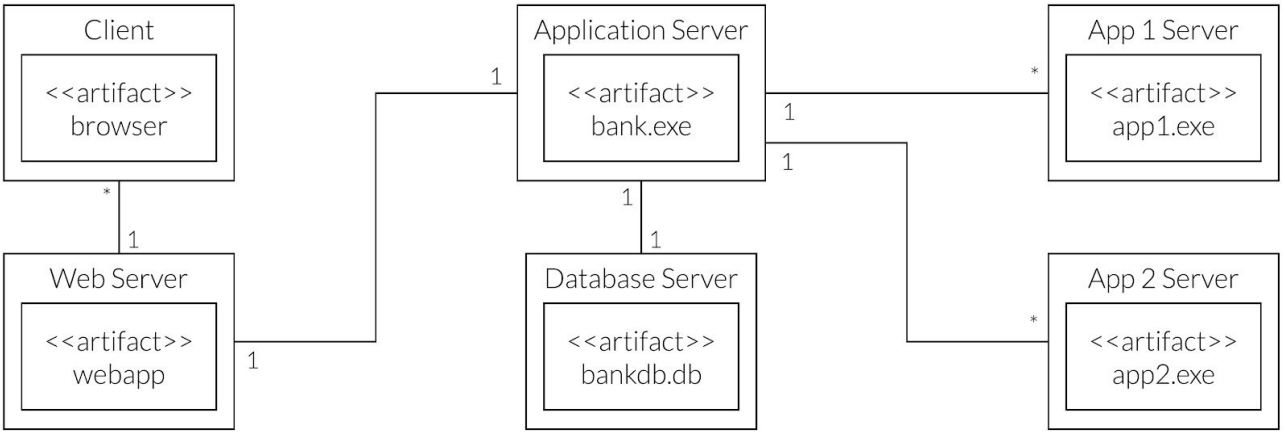
# 4. Enterprise Information Systems (EIS)

## EIS concepts

An **enterprise information system (EIS)** is a software system which holds core business data for a company or organisation and which performs operations involving this data. Usually done with **large and complex systems**, typically involving **distributed** processing and data.
In addition, an EIS will often be used by several **different applications** as a resource, including web and non-web applications.

EIS example: online banking:
An **accounts management system** for a bank will process data on thousands of customers, and may execute on computers separate from the database server machines holding the customer data. The system would be used by an **online banking application** (accessed directly by bank customers using a web browser), in addition to **internal applications** within the bank used by bank staff. All the applications may run on **different hardware** devices.

Typical EIS structure:




## EIS Specification and design

Enterprise systems can be specified in terms of the platform-independent **business rules** which define the properties of their data and operations. For example, that:

balance ≥ − limit
balance < 0 ⇒ charge = interestRate ∗ (-balance)
balance ≥ 0 ⇒ charge = 0

for a bank account system.

Platform-independent design concepts for EIS development are used where possible, although these concepts are related to those of the Java EIS platforms of **J2EE** and **Java EE.**

**EIS architecture and components**: the five-tier architecture is used to describe EIS apps.

**Client tier**

- Has responsibility to **display information** to the user, and **receive information** to transmit to presentation tier.
    - It may be a **thin client**, with minimal processing apart from visual interface functionality,
    - or a **fat client**, doing more substantial computation.
- The trend is towards thin clients, using web browsers, called **web clients**.
- Typical components: HTML pages or applets.

**Presentation tier**

- Has responsibility to **manage the presentation of information** to clients, **control interaction sequences**, and **relay user requests** to business tier.
- Typical components: controller/view components, JSPs.

**Business tier**

- Contains components **implementing** the **business rules and data** of the application.
- Typical components: session beans and entity beans.

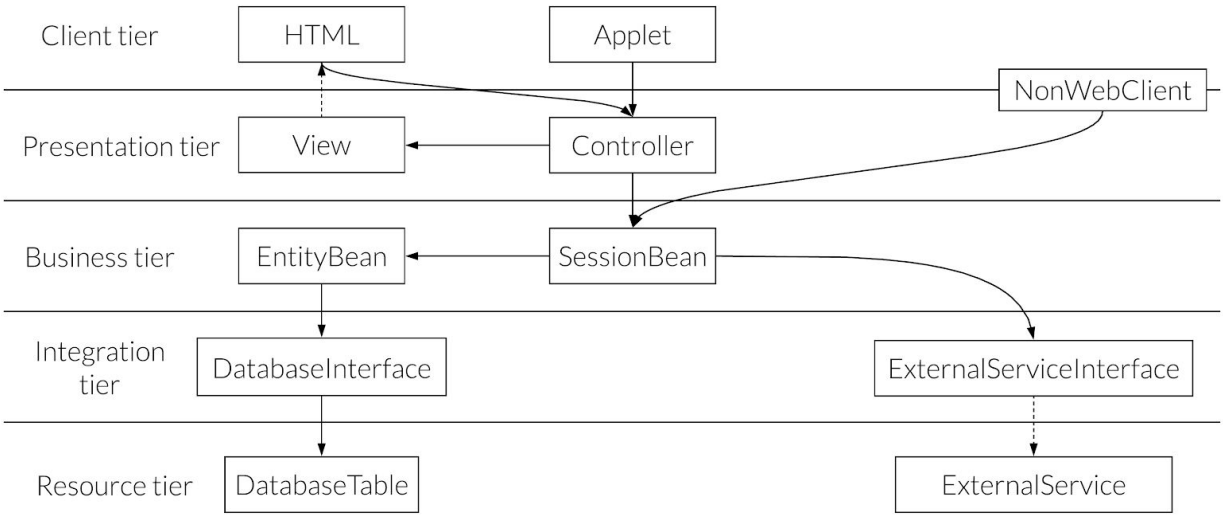
**Integration tier**

- **Mediates** between the business and resource tiers.
- Manages **data retrieval** and data conversion.
- **Insulates** higher tiers from direct knowledge of low-level data concepts.
- Typical components: Java database connectivity.

**Resource tier**

- Contains **persistent data storage** and **external services**.
- Typical components: databases, external services (e.g. credit card authorisation service).

Typical five-tier architecture for an EIS:




Not all systems need the full EIS architecture, but:

- **Session beans** should be introduced, and the **business tier should be separated from the presentation and resource tiers**, when:
    - the mix of business logic and view/control logic in the presentation tier becomes **too complex**, or
    - when business functionality needs to be made available to **multiple applications**.
- **Entity beans** should be introduced when:
    - persistent business components **become complex** and require transaction management, or
    - to make a system extensible to fully **distributed processing**.

**The business tier is a key element of an EIS**. It should be the **most stable part** of the EIS, based on **PIM specification** of business data and functionality. It should also be insulated from changes in data storage technology or resource details (through the integration tier) and from changes in UI technology of client applications (through the presentation tier). For example, the web server of an app can execute on a different machine to the app server and likewise for the database server.

The components of the business tier define the **business data and logic** of an EIS. In a Java-based system, the business tier could be **ordinary Java classes** or **specialised classes** defined in Java Enterprise Edition, known as **Enterprise Java Beans (EJBs)**.

## Session beans vs entity beans

The business tier contains both entity beans and session beans.

**Session beans**:

- Dedicated to a **single client**
- Live only for the **duration of the client's session**
- **Not persistent**
- Used to model **stateful** or **stateless** interactions between the client and business tier components
- A single session bean **encapsulates a group of operations**, usually corresponding to the use cases of the system, which operate on one or more entities of the system
- Each **use case** of the system would normally be implemented by a session bean

**Entity beans**:

- Provide an object view of **persistent data**
- **Coarse-grained** (not very detailed)
- **Multi-user** and **long-lived**
- Act as an **object-oriented facade for data** of system, which may actually be stored as relational tables, or as XML datasets
- Normally each entity from the PIM **class diagram** of the system will be managed by some entity bean

## Stateless vs stateful session beans

**Stateless session beans**:

- Provide a service by a **single method call**
- For instance, utility functions or logging services, or invocation of remote web services
- Because stateless session beans do not hold client-specific state, normally, they **can be shared between clients and reused** from one client to another

**Stateful session beans**:

- Provide a service specific to a **single client**
- **Cannot be shared**
- Involve **several related operations** (e.g., a shopping cart, or messaging operations)

Some EIS platforms, such as J2EE/Java EE, provide **automated synchronisation** of the entity bean data and the actual stored data in the resource tier. This is known as **container-managed persistence (CMP)**.
Alternatively, the synchronisation can be achieved by the programmer of the bean **explicitly providing suitable logic**. This is termed **bean-managed persistence (BMP)**.

CMP provides **greater portability**, avoiding use of platform-specific code within bean classes. It also **reduces the amount of manual coding**, since this functionality is provided by the platform.
However, with BMP there is greater scope for **customisation** of code used by the developer.

## Development process for EIS applications

### Business Tier

Group **classes** into modules which are suitable for implementation as **entity beans**, with **strong connections** between classes within each module (e.g. several invariants relating them) and **weaker connections** between classes in different modules.

- Each module is responsible for maintaining constraints which involve its contained entities.
- Modules normally have a 'master' or interface class, through which all updates to the module pass.
- Individual classes in modules enforce their local invariants, and may invoke operations of database interfaces.

Group **use cases** into **session beans**, according to entities which they operate on, and according to which use cases are frequently used together (in same session, by the same actor).
Grouping should aim to **simplify** (minimise) **dependencies** within business tier, and dependencies of presentation tier on business tier.

### Presentation Tier

**Controller** components (such as **servlets**) check the correct typing of parameters received from web pages, and pass on request data to business tier for checking of other constraints.

### Integration Tier

The **database interface** in is invoked from entity beans in the business logic tier.

## EIS design issues

Design issues for EIS cover all tiers of an EIS application from **security protection** to **database interaction approaches**.
Examples include data security, removing web-specific coding from business tier, separating presentation, business logic and data processing code, and pooling database connections.

**Data security**:

- **Passwords** should only be stored in **encrypted** format. To authenticate users, they can be compared with the (encrypted) user input but never exposed. Give the option to users to reset their password, rather than sending them the plain text of the current password - this means that you know their unencrypted password.
- **HTTPS** should be used systematically in all security-critical parts of a website.

**Removing web-specific coding from business tier**:

- Business tier code should not refer to HTTP request structures. Example of bad code:

```
public class House {
    String address;
    String style;
    public House(HttpServletRequest req) {
        address = req.getParameter("address");
        style = req.getParameter("style");
    }
}
```

- Using Using *HttpServletRequest* as input parameter type prevents non-web clients from using this business object. Instead, use data based on PIM or PSM class diagram of the system:

```
public class House {
    String address;
    String style;
    public House(String addr, String stl) {
        address = addr;
        style = stl;
    }
}
```

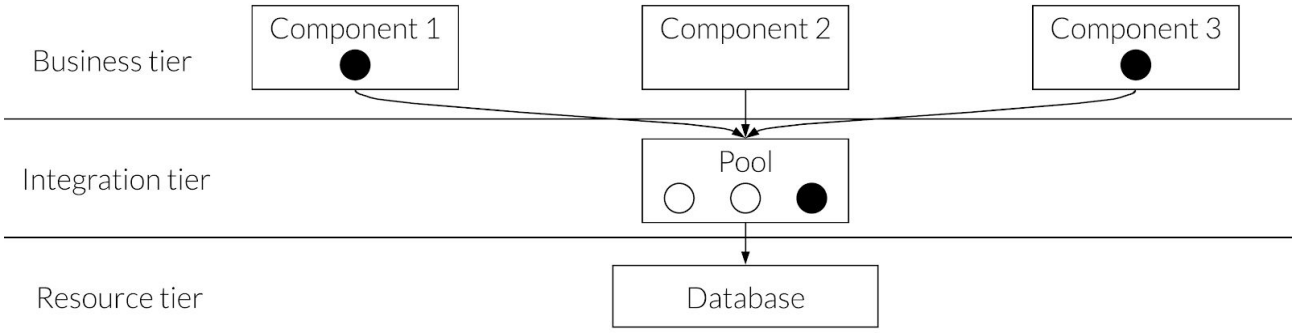
**Separation of code**:

- An important principle is to **separate** presentation, business logic and data processing.
- Code concerned with **database interaction** should be separated from presentation (UI) code and from business logic, to improve **flexibility**.
- EIS components are designed for **specific tasks** and give the basis of this separation:
    - Controller and view components for **presentation**
    - Session beans for business **processing**
    - Entity beans for complex **business data**

**Database connection pooling**:

- A particular technique useful for increasing the efficiency of a **database interface** is the introduction of a **connection pool**: creating a connection to a database is expensive, and should be minimised.
- This can be achieved by creating a **connection management component**, *ConnectionPool*, which holds a set of **pre-initialised connections**. JDBC provides pooling in *javax.sql.ConnectionPoolDataSource*.
- Components that require a connection to the database must ask the pool for a **free connection**: *con = pool.getConnection()*.
  When they have finished using it they must return it to the free set of connections that the pool maintains. However, if you close a *ConnectionPoolDataSource*, the underlying JDBC driver code decides whether to close connection or return to free set.

Example of a pool that contains three connections: two in use, and a free one:



# EIS Design Patterns

One solution to complexity of software system design is to provide **patterns** or **standard solutions** for common design problems.
Design patterns define microarchitecture within an EIS to implement a particular required **property/functionality** of a system, or to **rationalise the system structure**. They apply to **different tiers**.
If a pattern is used **incorrectly**, it can have a **detrimental** effect on the system. It is therefore important to be able to **identify** when a pattern can be applied **usefully**, and to understand how the pattern will **improve** the system.
Patterns are **independent** of any specific implementation technology.

**Presentation tier** patterns:

- **Intercepting Filter**: defines a structure of pluggable filters to add pre- and post-processing of web requests/responses, e.g.: for security checking.
- **Front Controller**: defines a single point of access for web system services, through which all requests pass. Enables centralised handling of authentication, etc.
- **View Helper**: separates presentation and business logic by taking responsibility for visual presentation (e.g., as HTML) of particular business data.
- **Composite View**: uses objects to compose a view out of parts (subviews).
- **Service to Worker**: combines *front controller* and *view helper* patterns to construct complex presentation content in response to a request.
- **Dispatcher View**: similar, but defers content retrieval to the time of view Processing..

**Business tier** patterns:

- **Business Delegate**: provides an intermediary between presentation tier and business services, to reduce dependence of the presentation tier on details of business service implementation.
- **Value Object**: an object which contains attribute values of a business entity (entity bean), this object can be passed to the presentation tier as a single item, to avoid the cost of multiple getAttribute calls on the entity bean.
- **Session Facade**: uses a session bean as facade to hide complex interactions between business objects in one workflow/use case.
- **Composite Entity**: uses an entity bean to represent and manage a group of interrelated persistent objects, to avoid costs of representing group elements in individual fine-grained entity beans (e.g., group a master object with its dependents).
- **Value List Handler**: provides an efficient interface to examine a list of value objects (e.g., result of a database query).

**Integration tier** patterns:

- **Data Access Object**: provides an abstraction of persistent data source access.
- **Service Activator**: implements asynchronous processing of business service components.

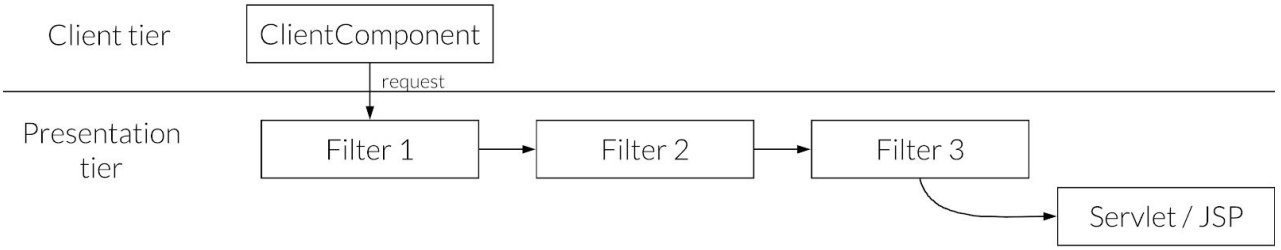
## Intercepting filter

This pattern has the purpose to provide a **flexible** and **configurable** means to add **filtering**, pre- and post processing, to presentation-tier **request handling**.
When a client request enters a web application, it may need to be checked before being processed, e.g.:

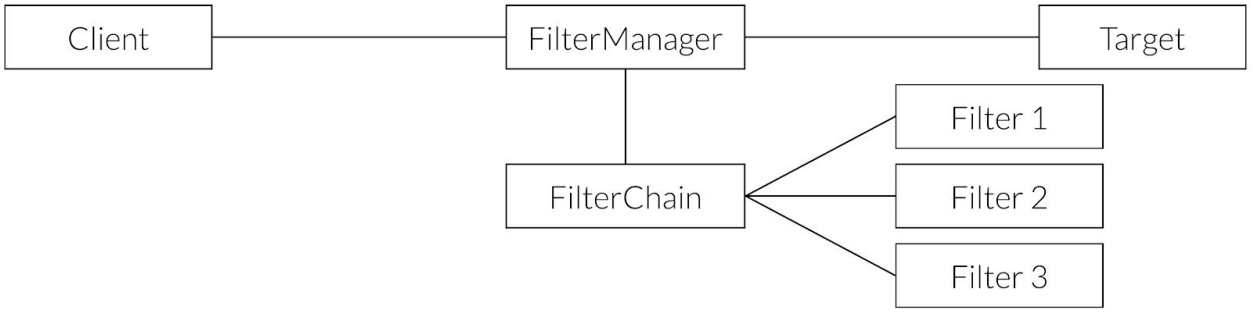
- Is the client's IP address from a trusted network?
- Does the client have a valid session?
- Is the client's browser supported by the application?

It would be possible to code these as nested conditionals, but is more **flexible** to use separate objects in a chain to carry out successive tests.

**Architecture**:




**Class diagram**:




**Elements**:

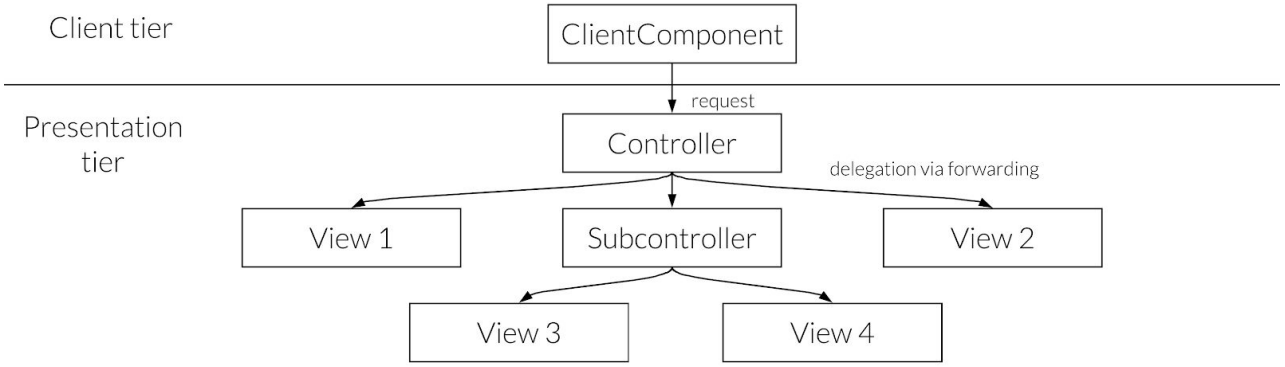
- **Filter Manager**: sets up filter chain with filters in correct order. Initiates processing.
- **Filter One, Filter Two, etc**: individual filters, which each carry out a single pre/post processing task.
- **Target**: the main application entry point for the resource requested by the client. It is the end of the filter chain.

Filters can be added and removed independently of other functionality. This is a useful pattern when entry conditions are **likely to change** in the future.

## Front controller

This pattern has the purpose to provide a **central entry point** for an application that controls and manages web request handling. The controller component can control **navigation** and **dispatching**. It **factors out** similar request processing code that is **duplicated** in many views (e.g., the same authentication checks in several JSPs).
It makes it easier to impose **consistent** security, data, etc, checks on requests. However, if the controller is not efficient, it can cause widespread **performance issues** for the rest of the application.

**Architecture**:



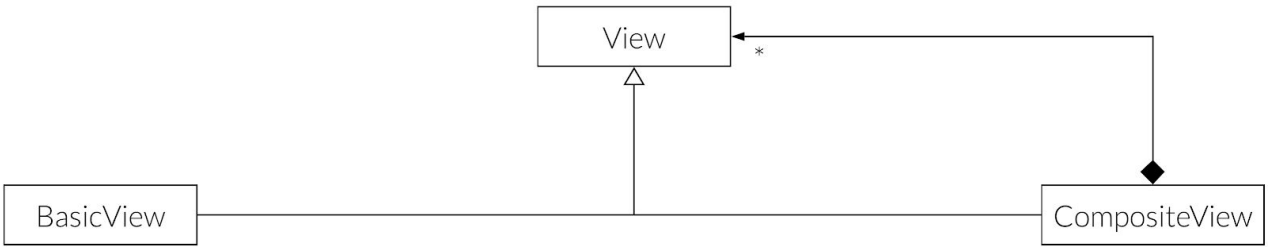

**Elements**:

- **Controller**: initial point for handling all requests to the system. It forwards requests to subcontrollers and views.
- **Subcontroller**: responsible for handling a certain set of requests, e.g., all those concerning entities in a particular subsystem of the application.
- **View1, View2, etc.**: components which process specific requests, forwarded to them by the controller.

## Composite view

This pattern has the purpose of **managing views** which are composed from **multiple subviews**. Complex web pages are often built out of **multiple parts**, e.g., navigation section, news section, etc. **Hard-coding** page layout and content provides **poor flexibility**. The pattern allows views to be **flexibly composed** as structures of objects.

**Class diagram**:




**Elements**:

- **View**: a general view, either atomic or composite.
- **View Manager**[1]: organises inclusions of parts of views into a composite view.
- **Composite View**: a view that is an aggregate of multiple views. Its parts can themselves be composite.

An example of this pattern in Java could be the `<jsp:include page = "subview.jsp">` tag in JSP, used to include subviews within a composite JSP page.
Other approaches include custom JSP tags and XSLT (if data is stored as XML).

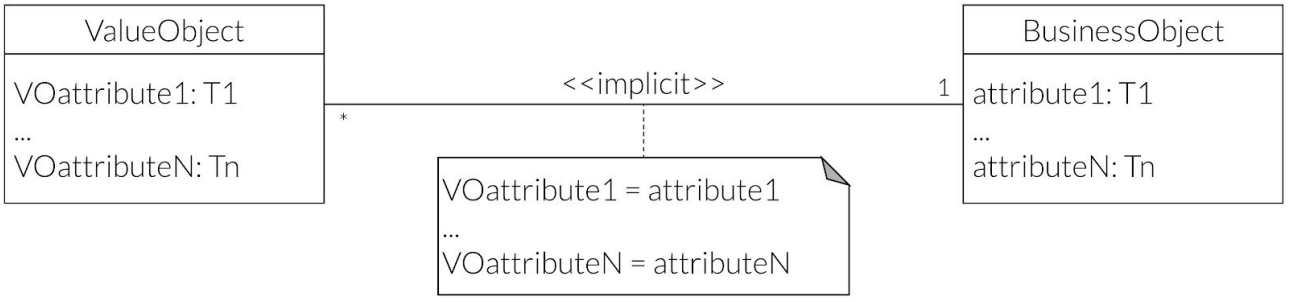
[1] The class diagram shows Basic View, but it is listed in the elements as a View Manager. The slides provided by both lecturers, plus the old slides from Dr. Kevin Lano, have the same issue, so we will never know if the element item should be Basic View or if the class diagram should have a View Manager instead of Basic View.

## Value object

This pattern has the purpose to improve the **efficiency** of **accessing persistent data** (e.g., in entity beans) by **grouping data** and transferring data as a group of attribute values of each object.
It is inefficient to get attribute values of a bean one-by-one by multiple *getAttribute()* calls, since these calls are potentially remote. This pattern **reduces data transfer cost** by transferring data as packets of values of several attributes. It can **transfer** data between presentation and business tiers, and between integration and business tiers. **However**, value objects are not helpful if calls for data are not **remote** or **expensive** (e.g. if the web service is located on a **single server** then this pattern is not appropriate).

**Architecture**:


**Class diagram**:

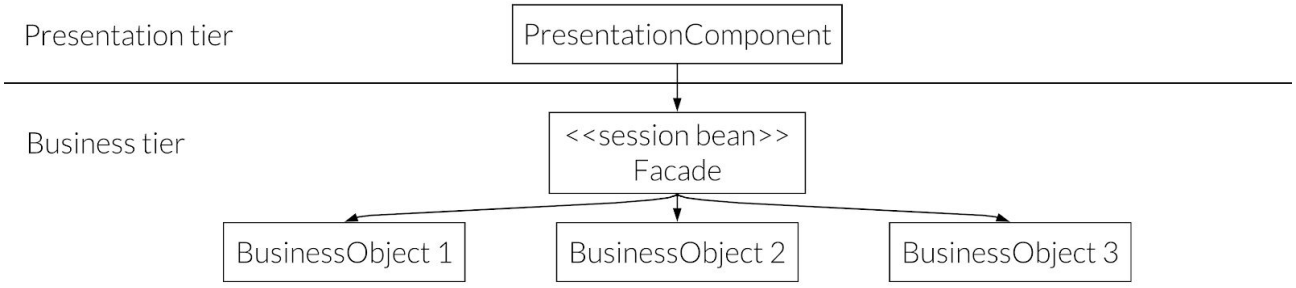



**Elements**:

- **Business Object**: can be a session or entity bean.
  - Holds **business data/logic**
  - It is responsible for **creating** & **returning** the value object to clients on request
- **Value Object**: holds **copies** of values of attributes of business object.
  - It has a **constructor** to initialise these.
  - Its own attributes are normally **public**.

## Session facade

This pattern aims to **encapsulate** the details of **complex interactions** between business objects. A session facade for a group of business objects manages these objects and provides a simplified **coarse-grain** set of operations to clients.
Interaction between a client and multiple business objects may become **complex**, with code for many use cases written in the same class; instead, this pattern **groups related use cases**.
**Architecture**:




**Elements**:

- **Presentation component**: client of session facade, which needs access to the business service.
- **Session facade**: implemented as a session bean. It manages business objects and provides a simple interface for clients.
- **Business objects**: can be session beans or entity beans or data. For the facade to be useful, they should have some relationship or common behaviour.

## Composite entity

This pattern uses entity beans to manage a set of **interrelated persistent objects**, to improve efficiency.
If entity beans are used to represent individual persistent objects (e.g., rows of a relational database table), this can cause inefficiency in access due to the (potentially) remote nature of all entity bean method calls. Instead, this pattern groups related objects into **single entity beans**.

**Class diagram**:


**Elements**:

- **Composite Entity**: coarse-grained entity bean.
  - It may itself be the **master object** of a group of entities, or hold a **reference** to this.
  - All **accesses** to the master and its dependants go via this bean.
- **Master Object**: main object of a set of related objects, e.g., a 'Bill' object has subordinate 'Bill Item' and 'Payment' objects.
- **Dependent Object**: subordinate objects of set.
  - Each can have its own **dependants**.
  - Dependent objects **cannot be shared** with other object sets.

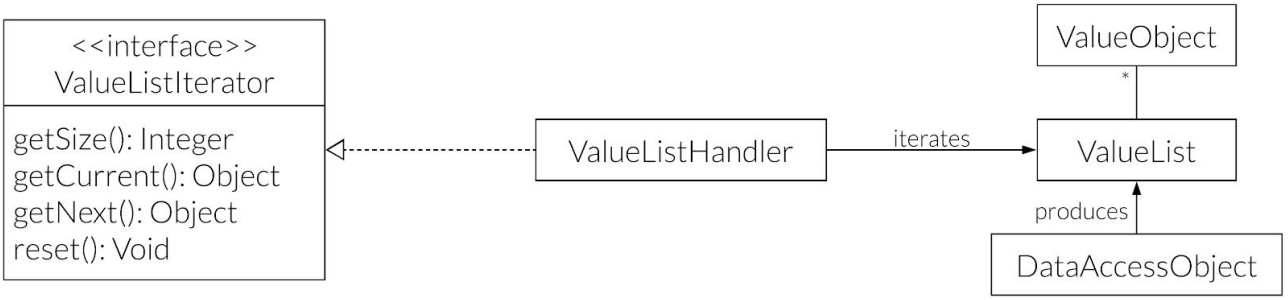
**Guidelines** for composite objects:

- If there is association E → D and no other association to D, put E and D in same entity bean.
- Put subclasses of a class in same entity bean as it.
- Put aggregate part classes of class in same entity bean as it.
- If D is a target of several associations E → D, F → D, etc., choose the association through which most accesses/use cases will be carried out, and make D part of the same entity bean as the class at the other end of that association.

## Value list handler

This pattern has the purpose to **manage a list of data items/objects** to be presented to client components.
It provides an **iterator-style** interface allowing **navigation** of such lists. The result data lists produced by database searches can be very large, so it is **impractical** to represent the whole set **in memory** at once. This pattern provides a means to access result lists **element by element**.

**Class diagram**:



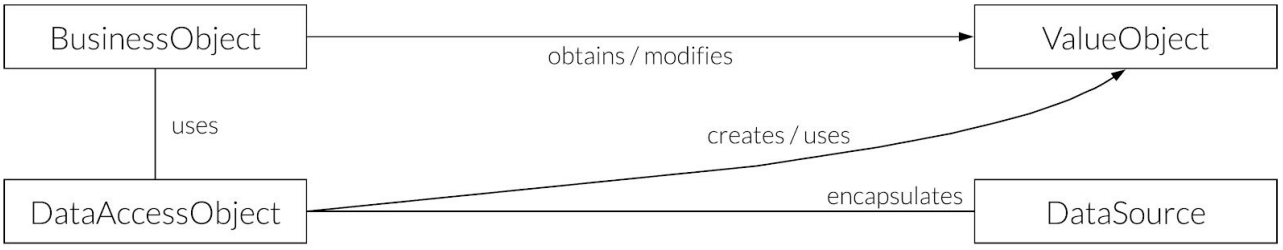

**Elements**:

- **Value List Iterator**: an interface with operations to navigate along the data list.
- **Value List Handler**: implements *ValueListIterator*.
- **Data Access Object**: implements the database/other data access.
- **Value List**: the actual results of a query. Can be cached.

## Data access object

This pattern **abstracts** from details of particular **persistent data storage** mechanisms, hiding these details from the business layer.
The **variety** of different APIs used for persistent data storage makes it difficult to **migrate** a system if these operations are invoked directly from business objects.
This pattern **decouples** the business layer from specific data storage technologies, using the *Data Access Object* to interact with a data source instead.

**Class diagram**:




**Elements**:

- **Business Object**: requires access to some data source. It could be a session bean, entity bean, etc.
- **Data Access Object**: allows simplified access to the data source. Hides details of data source API from business objects.
- **Data Source**: actual data. Could be a relational or object-oriented database, or XML dataset, etc.
- **Value Object**: represents data transmitted as a group between the business and data access objects.

**Factory Method** or **Abstract Factory** patterns can be used to implement this pattern, to generate data access objects with the same interfaces, for different databases.

# 5. EIS implementations

This chapter will show how EIS applications can be implemented in the Java EIS platforms of J2EE and Java EE. It will also cover discussions on web services and patterns for web services.

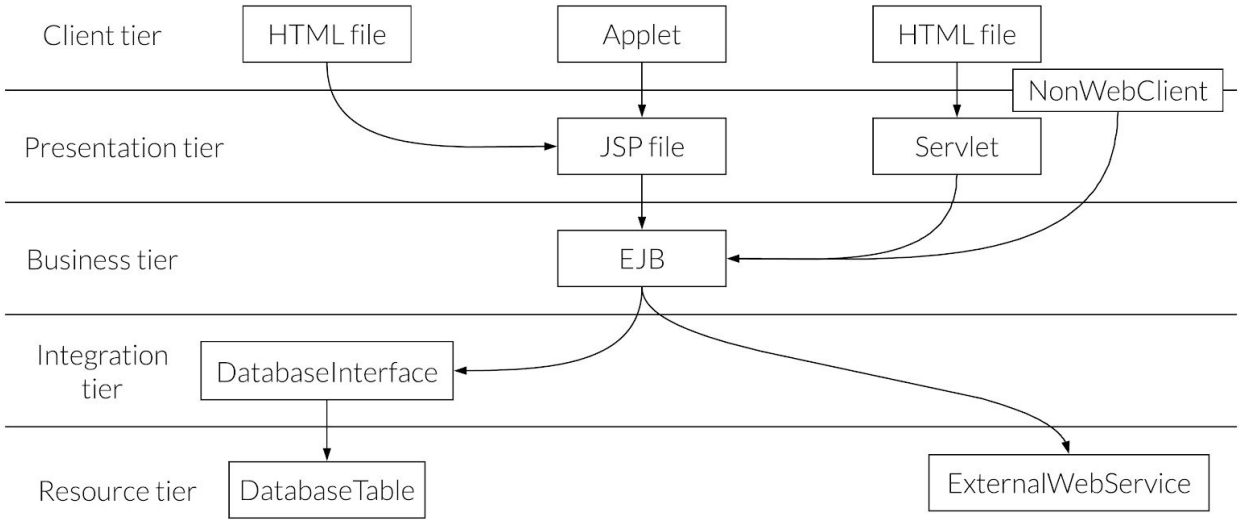
## Java 2 Enterprise Edition (J2EE)

**J2EE** is a Java framework for distributed enterprise systems, which includes:

- Servlets, JDBC and JSP.
- **Enterprise Java Beans (EJB)**: represents distributed business components, possibly with persistent data.
- **Java Message Service (JMS)**: an API to communicate with **message-oriented middleware (MOM)** to provide messaging services between systems.
- **Java Naming and Directory Interface (JNDI)**: an interface to support naming and directory services, such as the Java RMI registry for locating remote methods.
- **JavaMail**: an API for platform-independent mailing and messaging in Java.

J2EE uses the five-tier architecture defined in the previous chapter.

Typical J2EE system structure:




### Enterprise Java Beans (EJB)

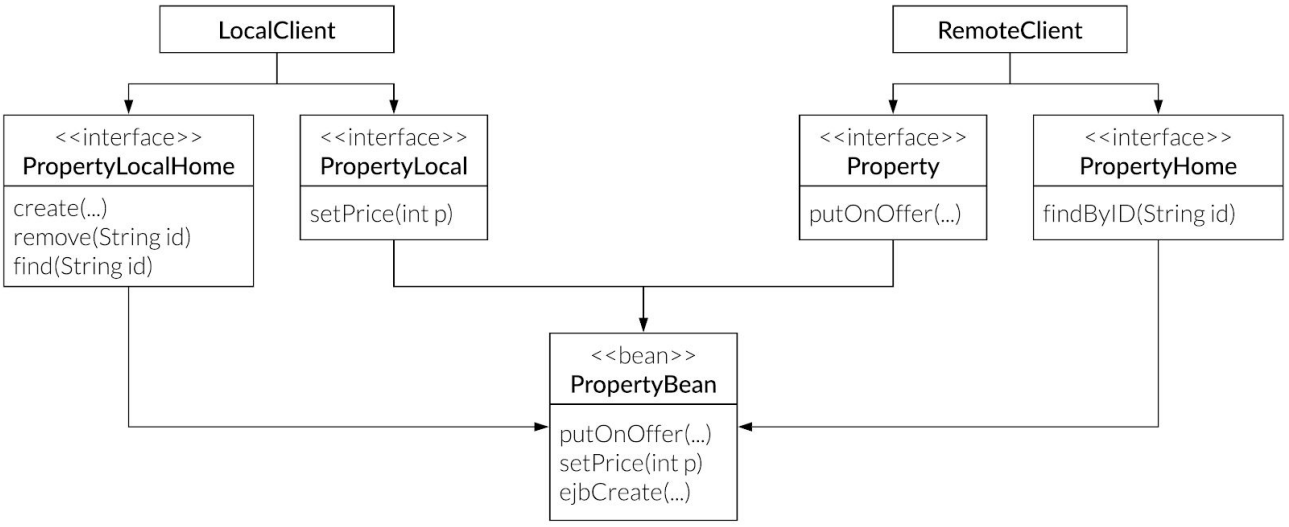
EJBs are a core mechanism for carrying out business logic on the server side of a J2EE-based system. There are two forms of EJB: **entity beans** and **session beans**.

Each EJB has a number of different **interfaces** which provide alternative ways that the bean can be used by different clients (**remote clients** on separate servers, or **local clients** on the same server):

- **Remote business interface** lists business operations specific to the bean.

    - In the dating agency system, an operation to determine matching members for a user would be listed in remote interface of *Member* EJB.
- **Remote home interface** lists life-cycle operations (creation, deletion) and utility methods (such as *findByPrimaryKey*) to return particular bean objects.
- **Local business interface** lists business operations that can be accessed by local clients, i.e., those executing in the same JVM as the EJB.
- **Local home interface** lists life-cycle and finder methods for local clients. These can offer different operations to home interfaces to differentiate the abilities of local and remote clients.

EJB interfaces for a Property entity:




Recall that J2EE provides an **automated synchronisation** of the entity bean data and the actual stored data in the resource tier. This is known as **container-managed persistence (CMP)**. The alternative is to define the behaviour manually, known as **bean-managed persistence (BMP)**.
EJB containers maintain an **instance pool** of entity and stateless session beans ready to use. When a client requests for a specific entity, an instance is chosen from the pool, populated with data from the database and assigned to service the client. When an entity is assigned to service a client, the instance is "wrapped" by an EJBObject, which provides the **remote reference** or **stub**.

J2EE provides a **sophisticated environment** for distributed and internet system construction, and for definition of web services. However its **complexity** can lead to poor design practices, and a substantial amount of **experience** and **familiarity** with J2EE seems necessary to take full advantage of its features. Solutions to this are the definition of **J2EE design patterns** to express good design structures for J2EE in a reusable way, or to encode expert knowledge of J2EE into a code generation tool for J2EE applications.

**Tool support**: the **UML2Web** toolset provides facilities for creating a PIM specification of an EIS, applying transformations to the PIM class diagram to produce a PSM, and code generation facilities to produce J2SE and J2EE implementations of a system.
Other tools which adopt MDA approach for **generation of J2EE applications** are **OptimalJ**, and **Codagen Architect**. These support the PIM specification of enterprise systems, and the generation of executable implementations. However, in contrast to our approach, there is no use of constraints to guide architectural choices, or to ensure the correctness of generated code.

## Web services

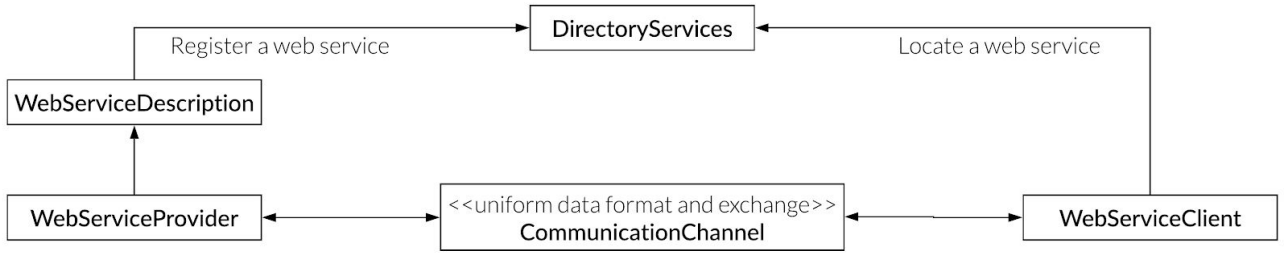
Web services are **software functions** that can be invoked across the **internet**. They support integration of applications at **different network locations**, enabling these applications to function as if they are part of a single large software system.
Web services are example of services in service-oriented architecture, and can be used to provide functionalities offered by public or private clouds in **cloud computing**.

A **functionality/task** may be made into a web service if:

- It involves access to **remote data**, or other **business-to-business interaction**
- It represents a **common subtask** in several business processes
- If it does not require **fine-grain interchange of data**
- If it is not **performance-critical**

Web service invocation is relatively **slow** because it uses data transmission over the internet, and packaging of call data.

Web service architecture:



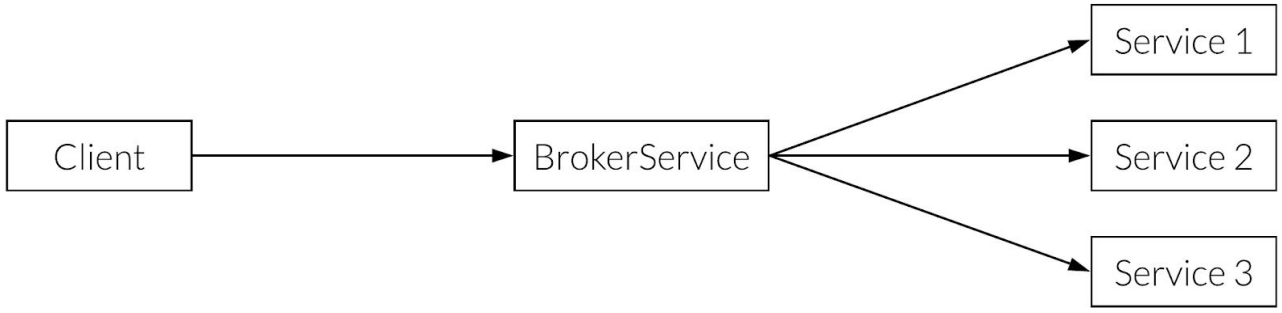

There are several ways in which an application can **communicate data and services** across the internet.

- **Raw HTML**: the most basic way a client program can extract data from server is by downloading web pages and then parsing them.
    - **Advantage**: does not depend on software at server, beyond support of HTTP.
    - **Disadvantage**: analysis of data depends on format of the web pages, which can change at any time.
- **CSV**: a server may make its data available as **comma-separated value files**, a text format for database tables.
- **FTP**: **file transfer protocol** provides a means to access files stored on a remote computer connected to internet.

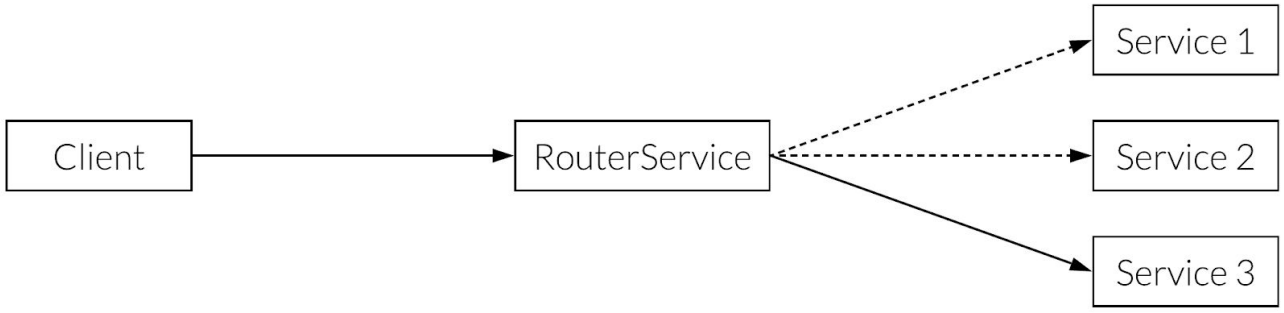
- **SOAP**: a more sophisticated approach is to use a protocol designed for application to-application communication across the web, such as SOAP (**Simple Object Access Protocol**), an XML-based protocol for exchanging messages, including descriptions of remote procedure calls.
  A SOAP message is an **XML document**, and an **envelope** that describes the method call that message concerns. The **body** of message can either be a request or a response.
- **WSDL**: **Web Services Definition Language** is also XML-based. It supports description of network **services** operating on messages with document or procedural content.

## Web service design patterns

**Broker design pattern**: the source application needs to call **multiple target services** (e.g., to find price of an item supplied by alternative suppliers). This pattern introduces the **broker service** to **perform this distributed call**.




**Router design pattern**: the source application needs to call **one specific** (external) **service**, depending on **various criteria/rules**. This pattern introduces the **router service** which applies these rules to **select the correct target service**.




## Implementing web services using J2EE

J2EE provides the **JAX-RPC** API to program web services that communicate using an **XML-based protocol** such as SOAP. JAX-RPC **hides details** of SOAP message formats and construction, and is **similar to the Java RMI** (Remote Method Invocation) interface.
Unlike RMI, web clients and services **do not have to run on Java platforms**, since HTTP, SOAP and WSDL are used to support client-server communication, independent of particular programming languages.

J2EE also provides means to directly construct SOAP messages and interact with web services by sending such messages. The **SAAJ (SOAP with Attachments API for Java)** API supports **construction of SOAP messages**, and **transmission** of these over a *SOAPConnection*.

## Java Enterprise Edition Platform

**Java EE** is successor to J2EE. It incorporates all J2EE technologies, but **simplifies their use**, via defaults and annotations in source code to indicate what role a class plays in system and what interfaces (remote or local) are required for it (e.g. entity beans are now incorporated as Entity components in Java). However. the developer needs to include code to **maintain mutually inverse relationships**.
Java EE also provides **simpler specification of web services**, using **annotations** to declare that a component offers a web service.

Many other web and EIS platforms are based on J2EE/Java EE, e.g., Struts, Spring, Hibernate.