Software Engineering of Internet Applications

Lecture 1: Software Engineering

Definitions

Specification: Detailed precise presentation of something or of a plan for something / A written description of an invention for which a patent is sought.

Architecture: The **art or science of building** / The manner in which a computer system is organised and integrated.

Engineering (Software E): The design and manufacture of complex products

Why bother with software engineering?

System failures occur often and result in costly and dangerous situations. This bugs cost the US economy $312 billion per year.

- Typical software projects require 1-2 years and at least 500,000 lines of code
- Only between 70-80% of all projects are successfully completed.
- Over the entire development cycle, each person produces on average less than 10 lines of code per day
- During development on average 50-60 errors are found per 1000 lines of source code. Typically this drops to 4 after system release.

What is software engineering?

Software engineering is the practical application of scientific methods to the specification, design, and implementation of programs and systems

Semi-Formal Methods

A language is **formal** when it has a formal language (**syntax**) whose meaning (**semantics**) is described in a mathematically precise way.

A development method is **formal** when it is based on a formal language and there are semantically consistent transformation/proof rules

Semi-formal methods are widely used, for example **UML**, and are generally made of syntax with a mere hint of semantics.

The advantages and disadvantages of formal methods are:
+ Typically more concise
+ Precise and unambiguous
+ Precise transformation rules
+ Uniform framework for specification, development and testing
- They are more difficult for novices

Structuring and abstraction in modeling

Modeling

Models are built during the early development phases to specify requirements clearly and precisely without committing to specific algorithms or data-structures.

Later models can be made to reflect specific implementation architecture (sketches, interfaces, communication etc). Modeling style depends on the notion of “component” (it can be a function, procedure, class, module…)

To overcome the complexity of program development in the large:

- **Problem Structuring** serves to organise or decompose the problem or solution
- **Abstraction** aims to eliminate insignificant details

Classical approaches to structuring and simplification:

- **Functional decomposition**: decomposition in independent tasks
- **Parameterization and generic development**: focus on reusability.
- **Model simplification**: to improve the understanding of tasks and possible solutions
- **Information-hiding**: Interfaces and property-oriented descriptions

In all cases **interfaces** must be clearly describe, taking into account:
- **Syntactic Properties**: available rules, the types of their arguments etc…
- **Semantic Properties:** description of the entities’ behaviour

Lecture 2 MDD to UML Use Cases

Module-Driven Development (MDD)

The idea is that it is Language and Platform independent.

Model Driven Development (MDD):

Development of software by:
- construction and transformation of models
- semi-automated generation of executable code from models

rather than focusing on manual construction of low-level code

MDD aims to make production of software more reliable and efficient by:
- freeing developers from complexity of implementation details on particular platforms (it should be more or less the same for Java, C implementations)
- retaining core functionality of a system despite changes in its technology

We focus on **Module-Driven Architecture (MDA)** approach to MDD
It is relevant for web applications because of rapid exchange in web technologies in many web applications.

MDA Concepts

MDA aims to support cost-effective adaptation of a system to new technologies by separating technology independent models of data and functionality from technology specific models.

Key concepts of model-driven development using MDA are
- Platform-Independent Models (PIMs)
- Platform-Specific Models (PSMs)
- Model Transformations.

Platform Independent Models (PIM)

The idea of PIM is that by not having to consider the specific lower-level implications of working with a certain language it “frees our mind”.

System specification that models systems in terms of domain concepts and implementation-independent constructs

- Should express “business rules” that are core definition of functionality of system
- Should be reusable across many different platforms via the use of PSMs, and be flexible to accommodate enhancements.

If PIM is defined in a manner that abstracts from particular algorithms and computation procedures, then it can be referred to as **Computation Independent Model (CIM)**

Platform-Specific Model (PSM)

System specification of a system tailored to specific software platform and programming language.

Defines functionalities of a system in sufficient detail that they can be directly programmed by a model

Model Transformations

Produce a new model from an existing model

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

- Typical transformations include
- introduction of design patterns or
- elimination of model elements that are not supported by a particular platform (such as multiple inheritance, or many-many associations)

We use UML as language for PIMs and PSMs

Unified Modelling Language UML Notation

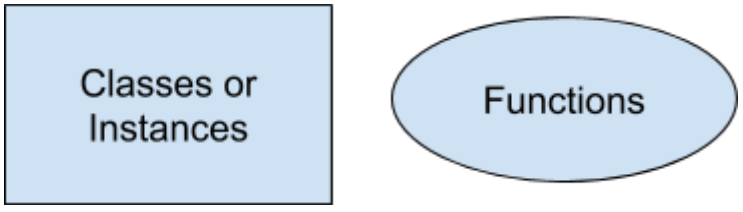
Unified Modelling Language, consists of several (seven) inter-related diagrammatic and textual notations including:

- **Use case diagrams**: shows services provided by a system and which agents these interact with
- **Class diagrams and object diagrams:** describe structure of entities involved in a system and the relationships between them
- **State machine diagrams (a.k.a statechart diagram):** describe behaviour of objects and execution steps of individual operations of objects
- **Object constraint language (OCL)**
- **Interaction diagrams**
- **Activity diagrams**
- **Deployment diagrams**

UML Conventions

All UML Diagrams denote graphs of nodes and edges

- Nodes are **entities** drawn as rectangles or ovals

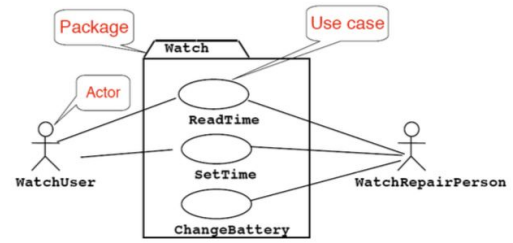


- Names of **classes are not underlined**

- SimpleWatch
- FireFighter
- Names of **instances are underlined**
- myWatch:SimpleWatch
- Joe:FireFighter
- An edge between two nodes denotes a relationship between corresponding entities.

Use Cases

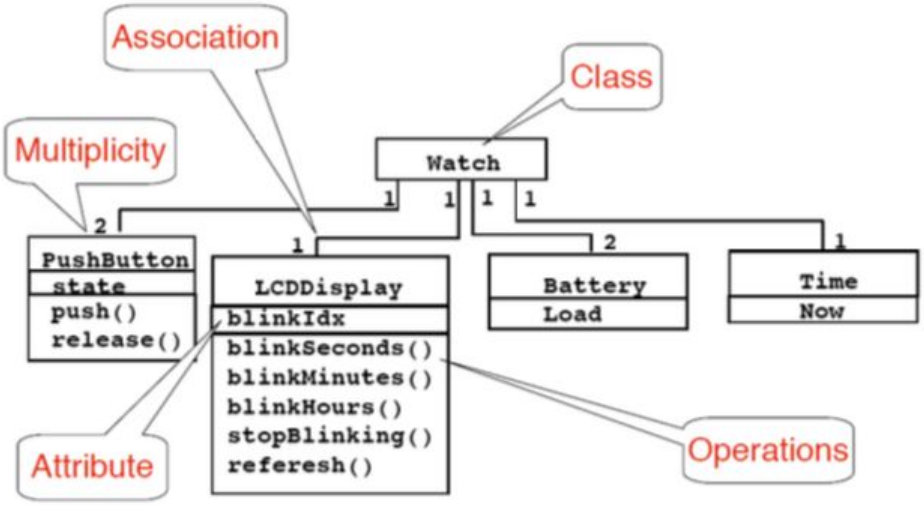
Scenario based technique in the UML which identify the actors in an interaction an which describe the interaction itself. The set of all Use Cases specify the complete functionality of a system and its environments.



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

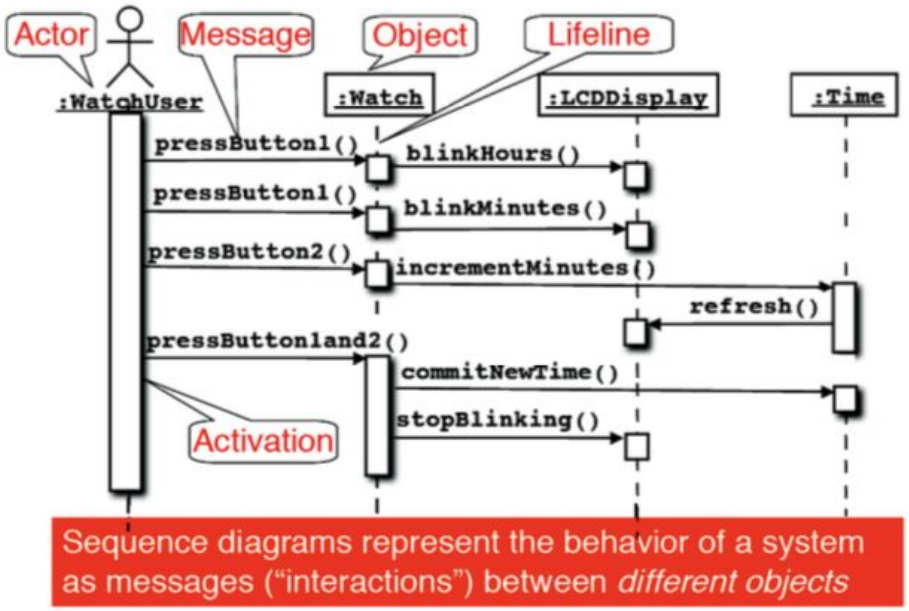
Class Diagrams

Class diagrams represent the structure of the system.



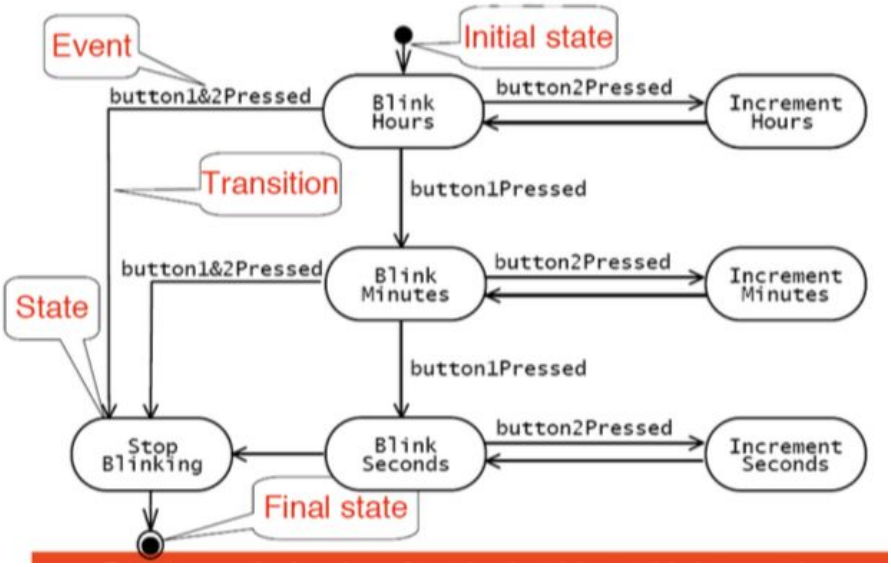
Sequence Diagrams

Sequence Diagrams show how the data/information flows as time progresses in our application.



Statecharts Diagrams

These show a state machine and these always have a final state. They represent the behavior of a single object with dynamic behavior.



Object Constraint Language (OCL)

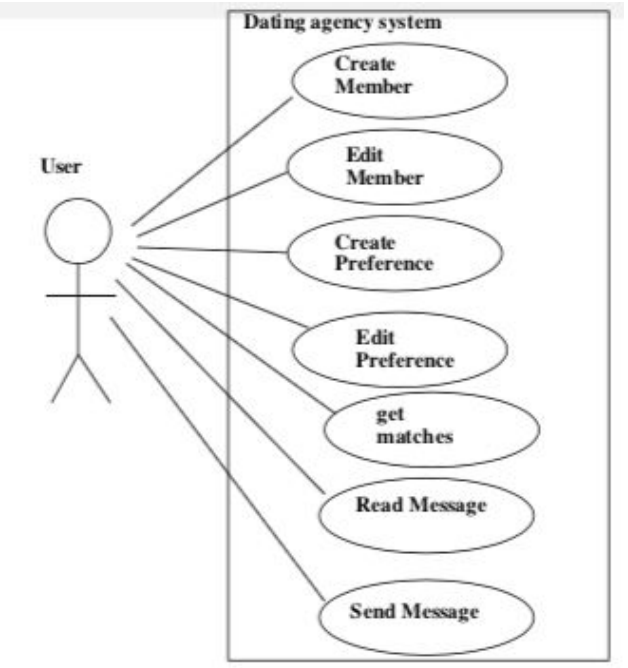
OCL enables developers to describe properties of classes, associations and state machines in detail, using a textual language.

Use cases and class diagrams are used at earliest development stage of a system, to define services that system will provide to different groups of users, and data that it needs to process.

Manuel Martinez Chamorro
Alex Franch Tapia

Case Study: Online Dating Agency

System allows users to register and record their details (age, height, location, etc.) and their preferences for dating partners (age range, location, etc.).



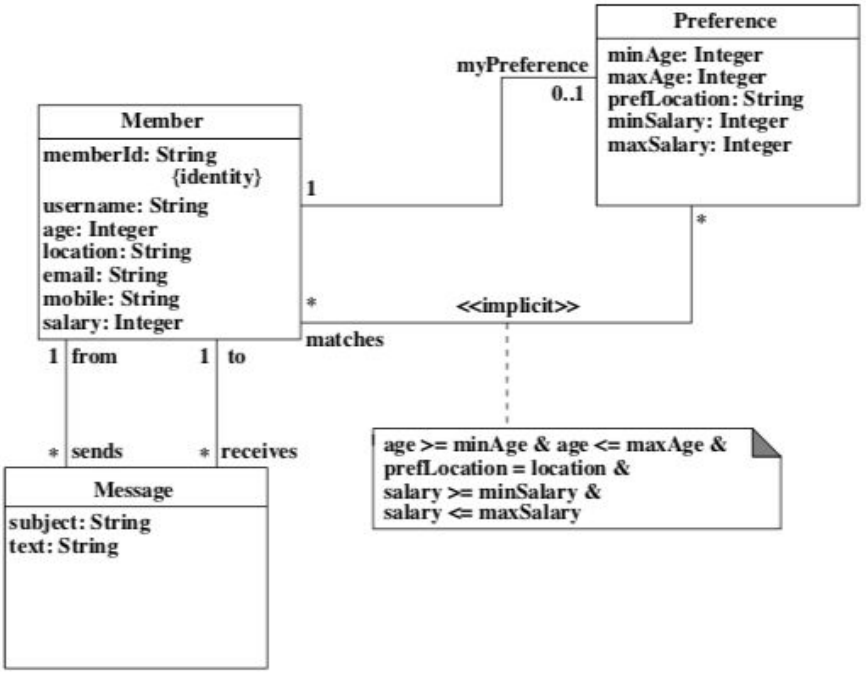
For each user, system can produce list of other users who match the preferences.

The constraint:
age ≥ *minAge* & *age* ≤ *maxAge* & *location* = *prefLocation* & *salary* ≥ *minSalary* & *salary* ≤ *maxSalary*

defines when a member matches against another member's preference — age and salary are in preferred ranges of other member, and location is same as preferred location.

For each member, the set
myPreference.matches
is set of other members who match preference of the member.

After we have the use case we will come up with the class diagram



Use Cases

Use Cases are scenario-based technique in the UML which identify the actors/roles in an interaction and which describe the interaction itself. Therefore, a Use Case is a set of scenarios tied together by a common user goal (which might succeed or fail).

Note: sequence diagrams may be used to add detail to Use Cases by showing the sequence of events processing in the system.

Actor/Role: models an external entity which communicates with the system (user, external system, environment). It does not have to be a person.

A Use Case: represents a kind of task provided by the system as an event flow. Consists of: Unique name, participating actors, entry conditions, flow of events, exit conditions and special requirements.

Each use case has a

- **Primary actor**: which calls on the system to deliver a service
  - Who calls on the system to deliver a service. *Client in the initiating factor of withdraw.*
  - Secondary actors are those with which the system interacts when carrying out the use case.
- **Pre-condition**: what the system should ensure is true before the systems allows the use case to continue
  - Entry condition of Withdraw: Client has opened a bank account with the bank, and has received a bank card and PIN.
- **Guarantee (or exit condition)**: what the system will ensure at the end of the use case.
  - Exit condition of Withdraw: Client has requested cash or receives an explanation from ATM about why cash could not be dispensed.
- **Trigger**: specifies the event that gets the use case started

i.e

- For example, the withdraw event flow:

Actor steps	**System Steps**
1. Authenticate	
	2. ATM displays options
3. Client selects “Withdraw”	
	4. ATM queries amount
5. Client enters amount	
	6. ATM returns bank card
	7. ATM outputs specified amount

- Anything missing?
  - Exceptional cases.
  - Details of authentication.
- Some of these issues can be specified with extensions or with other kinds of models.

Note that:

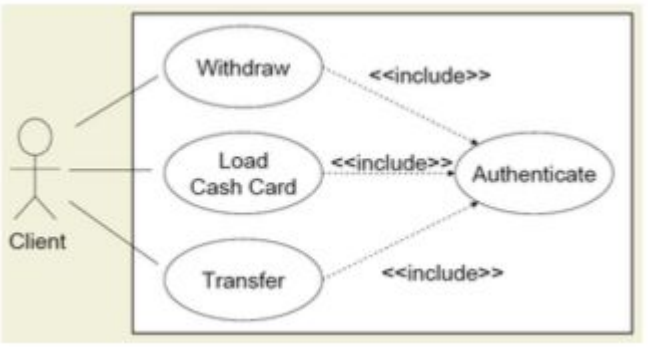
- Each step should be a simple statement and should clearly show who is carrying out the step.
- The step should show the intent of the actor, not the mechanics of what the actor does.

- Consequently, one does not describe the user interface in the Use Case (writing the Use Case usually precedes designing the user interface).

<<Include>>

Used to re-use Use Cases
Common behaviour can be factored by using: **<<include>> stereotype** to include use cases

In this case we see that **dependencies are marked with dashed arrows**, meaning that in order for a Client to Withdraw, Load Cash Card or Transfer it must be authenticated.



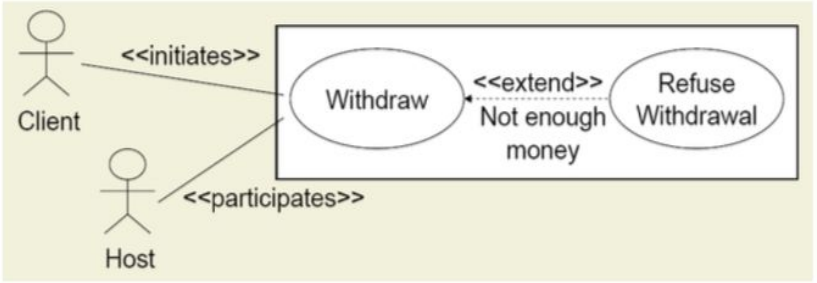
This allows us to create smaller models and spot the possibility of reusing existing functionality.

Is Reusing Use Cases good or bad?
+ Convenient
+ Shorter descriptions
+ Common functionality might lead to reusable components
+ Enables integration of existing components
- May lead to object decomposition instead of OO model, this decomposition would be a waste of time
- Requires more UML skills

<<Extend>>

Used to specify a special case. It can be used to factor different behavior into a single scenario.
Syntax: dashed arrow from exception to main case.

- It specifies a point at which the behavior may diverge (extension point)
- Extending case specifies the condition under which the special case applies (as entry condition)



Actor steps	**System Steps**
1. Authenticate (Use Case `Authenticate`)	
	2. ATM displays options
3. Client selects “Withdraw”	
	4. ATM queries amount
5. Client enters amount	
	6. ATM returns bank card
	listed as extension point
	7. ATM outputs specified amount

Note that use cases can also be written in text. There are different types of use cases. You may present use cases to business people, which means you may want to expose the functionality of a system in words not in diagrams.

Buy a product
Goal level: Sea level

Main Success Scenario:
1. Customer browses catalogue and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; 1-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming e-mail to customer

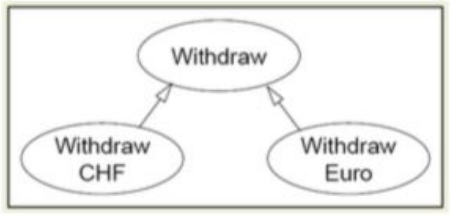
Extensions:
3a. Customer is a regular customer
  3a.1. System displays current shipping, pricing, and billing information
  3a.2. Customer may accept or override these defaults, returns to main success scenario at step 6
6a. System fails to authorize credit purchase
  6a.1. Customer may reenter credit card information or may cancel

Generalization and Specification

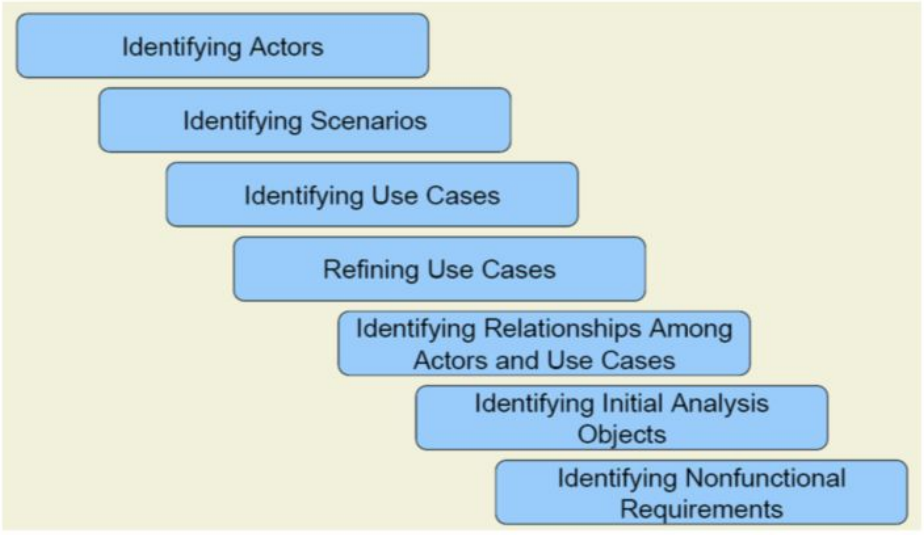
Generalization relates two actors or two use cases to each other - factoring out **common** (but not identical) **behaviour.**

- Example actor generalization: *each Journal Borrower is a Book Borrower. Semantics: every Book borrower Use Case holds for each Journal Borrower.*

We can also inherit behaviour from a parent, meaning that we add or override some behavior.



Requirements elicitation to identify Use Cases



Identifying Actors

Actors represent roles

In order to distinguish between an actor (role) and an object (part of the system) we must remember an actor is in the outside while an object is in the inside

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

Actors are not the same as objects, rather they are like roles, these are outside the Use Case, 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?

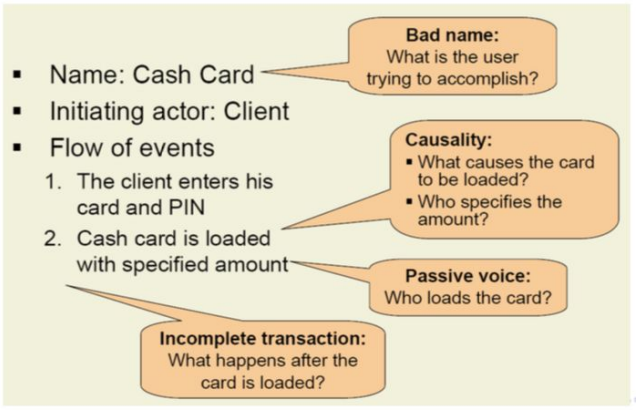
Elicitation: the process of gathering information to understand the requirements.
- Note that requirements evolved over time

Types of scenarios

Use cases can be different with respect to the scenario we are in.
- **Visionary scenario:**
  - To describe a future system
  - Usually used in greenfield engineering or reengineering projects
  - Can often not be done by the user or developer alone
- **As-is scenario**:
  - Used to describe a current situation
  - Usually used in reengineering projects
  - 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

NOTE: Scenario's name should be a verb describing what the actor wants to accomplish

Guidelines for Use Cases



Name:

- Use a verb phrase to name the Use Case.
- The name indicates what the user is trying to accomplish. Examples: “Withdraw”, “Authenticate”, “Load Cash Card”.

Length:

- A Use Case should not exceed two A4 pages.
- If longer, use «include» relationships.
- A Use Case describes a complete set of interactions

Flow of Events:

- Use active voice.
- Steps start either with “The Actor...” or “The System...”. The causal relationship between steps is clear.
- All flow of events are described (not only main flow). The boundaries of the system are clear.
- Important terms are defined in the glossary.

Bottom line for Use Case diagrams:

- The title has to be specific. “Website” -> Fundraising Web Application / Website
- Actors must have names not descriptions of what they do. “User that is donating” -> Donator
- An Arrow with a white head means **inheritance** not the <<extends>> tag, that means that we add an exception.
- If were is only one use case that makes use of a <<include>> tag then we do not need it as we are decomposing a sub-use case for no reason, rather we remove it. *Only one use case makes use of an authenticate inclusion.*

Tutorial Framework

- Python with Flask (Web Framework) and SQLAlchemy Database
- Java (spring, tomcat and gradle)
- Ruby-on-Rails
- Haskell with Scotty
- JavaScript with Node.JS

Lecture 3 UML Case Diagrams

For bigger software projects OOP can be very helpful but when starting a project you must consider whether its actually needed.

A class is a part in an OOP that tells what it has and what it does. You start by making an **abstraction** of what the client wants and a **modeling method** turns the idea of an object with data and operations into a **UML Class Diagram**

Object Models

Describe the system in terms of object classes and their associations

An **Object Class** (depicted as a Rectangle) is an abstraction over a set of objects with common attributes and each object's operations which is reusable across different systems.

This is a natural way of reflecting on real-world entities, but it makes it difficult to model abstract entities.

Object class identification is a difficult process and requires a deep understanding of the application domain.

Basic Definitions and Concepts

Objects

In object oriented modeling (OOM), **objects** are the main unit of abstraction

- Objects carry out **activities**
- Interface to objects is **event oriented**

I.e. a robot has sensors, actuators, control units etc…
In functional decomposition on the other side:
- Decomposition of problems into functions (not activities)
- Interface is data-oriented (I/O of functions)

I.e. a compiler has a parser, code generator etc…

In OOM one interacts with objects which have:

- **State**: encapsulated data consisting of **attributes** or **instance variables** part of which can be mutable
- **Behaviour**: An object reacts to **messages** by changing its state and generating further messages
- **Identity**: An object exists and has a name

Interfaces

An **interface** defines which messages an object can receive
- Describes behaviour without describing implementation or state
- There is difference between **Public Interfaces** (all objects can use) and **Private Interfaces** (only the object itself can use)

Public interfaces are often denoted with the symbol +

Classes

A **Class** describes objects with similar structure and behaviour

A class has a fixed interface and defines attributes and methods.

Advantages of classes:

- **Conceptual**: many objects share similarities
- **Implementation**: only one implementation
- **Inheritance** or overriding of methods
- **Dynamic binding** (method implementations are determined at run-time)

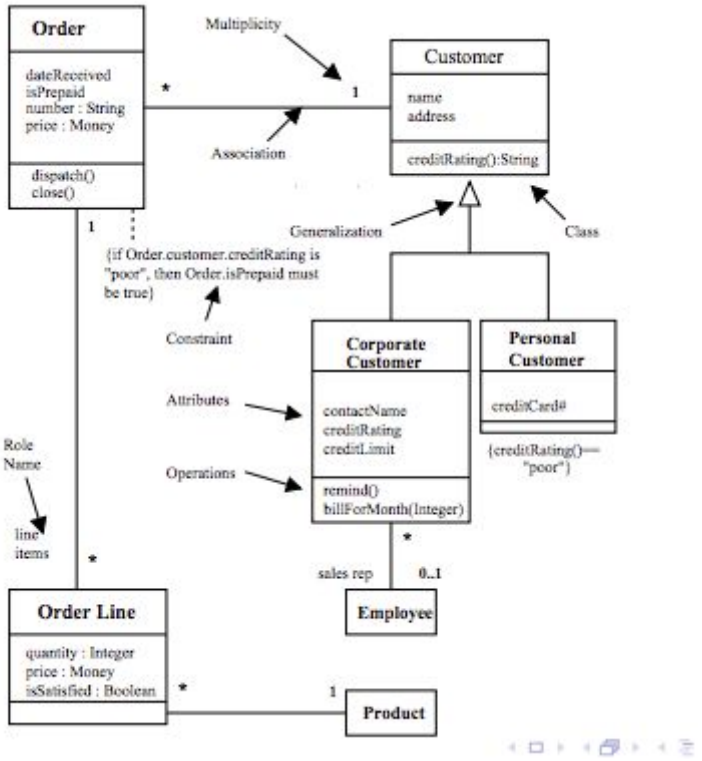
Class Diagrams

Language to model the **static view** of a system

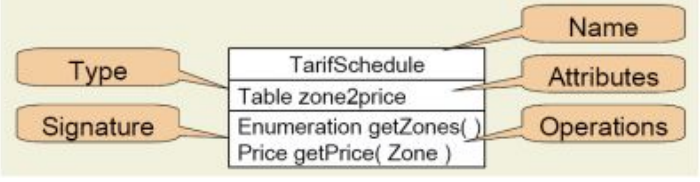
A **class diagram** describes the **kind of object** in a system and their different **static relationships** (Associations & Subtypes)

Class diagrams represent the structure of the system and are used:

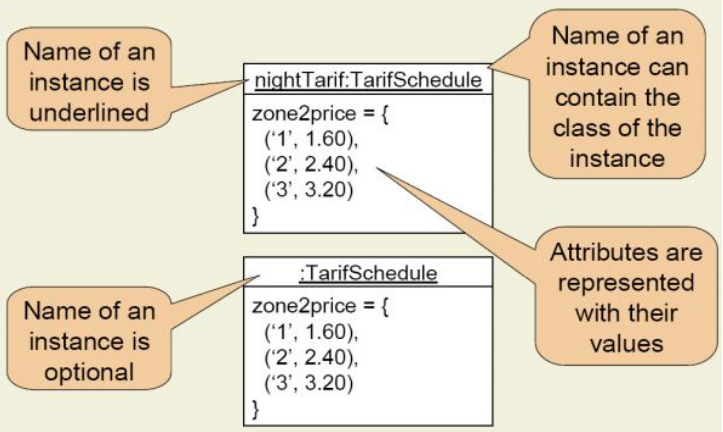
- During requirement analysis to model domain concepts
- During system design to model subsystems
- During object design to specify the detailed behaviour and attributes of classes



A class is represented as a square with a name and a set of optional attributes and operations (name is mandatory)



Instances (Objects)



Actor vs Class vs Object

Actor:

- An entity outside of the system interacting with the system (e.g. passenger)

Class:

- An abstraction modeling an entity in the application or solution domain
- The class is part of the system model (e.g. ticket distributor, server)

Object:

- Specific instance of a class (e.g. Joe, the passenger who is purchasing a ticket from the ticket distributor")

Attributes

The full syntax for an attribute is:

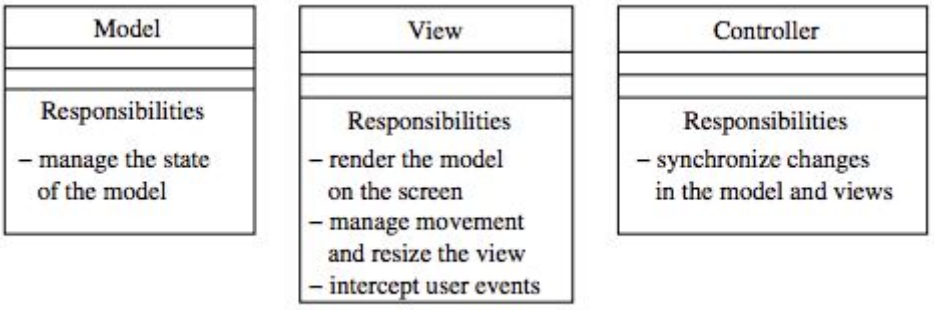
```
visibility name : type multiplicity = default {property-string}
```

e.g.: - `name: String [1] = "Untitled" {readOnly}`

- Only the `name` is necessary.
- The `visibility` marker indicates whether the attribute is public (+), private (–), package (~), protected (#).
- The `name` of the attribute (how the class refers to the attribute) roughly corresponds to the name of a field in a programming language.
- `type` indicates a restriction on what kind of object may be placed in the attribute; roughly corresponds to type of a field in a programming language.
- For the `multiplicity`, see discussion in the following slides.
- The `default value` is the value for a newly created object if the attribute isn't specified during creation.
- The `{property-string}` allows one to indicate additional properties for the attribute.
  - E.g. `{readOnly}` to indicate that clients may not modify the property, or `{frozen}` to indicate that property is immutable.
  - If it is missing, one can usually assume that the attribute is modifiable.

Responsibilities

One can also specify responsibilities like in this UML Class Diagram of the MVC model.



Relations

Relationship	*Function*	*Notation*
Association	Describes connection between instances of classes	———
Generalization	A relationship between a more general description and a more specific variety	——▷
Dependency	A relationship between two model elements	- - - -➢
Realization	Relationship between a specification and its implementation	- - -▷
Usage	Situation where one element requires another for proper functioning	- - - -➢

Associations describe relationships between objects in a class, the others describe relationships between classes or instances

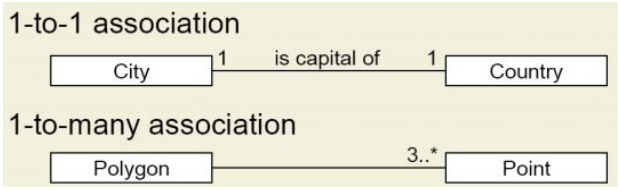
Associations

Denotes relationship between classes, and can have optional labels providing additional information like “is capital of” below.

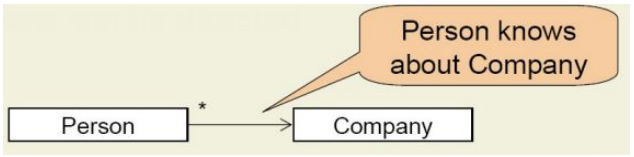
The **multiplicity** of an association and denotes how many objects the source object can reference

- Exact number: 1, 2, etc. (1 is the default).
- Arbitrary number: ∗ (zero or more).
- Range: 1..3, 1..∗

Semantics constrains the relation. (the relation has to make sense)

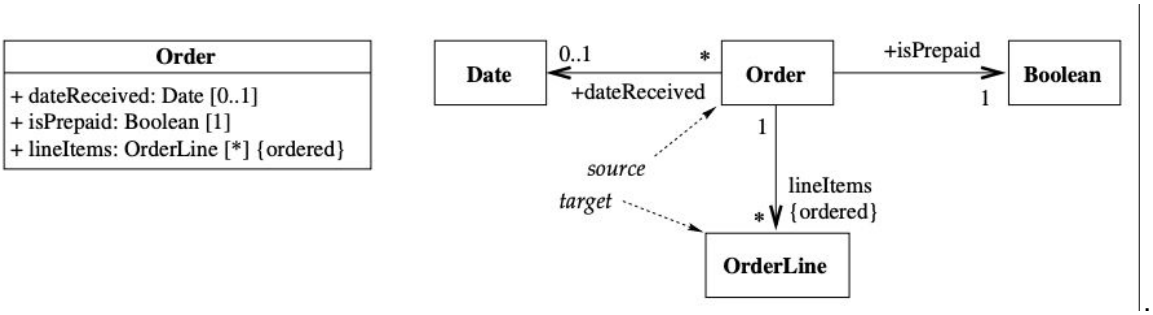


Navigability of associations mean they can be directed (although this is not common)



Associations vs attributes (or properties)

Much of the same information that one can show on an attribute can be shown on association



In order to choose which one to use:

- Use Attributes for small things such as dates or Booleans
- Associations for more significant classes, such as customers and order.

Multiplicity of Attributes

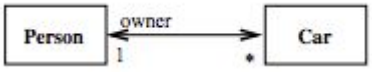
Property of how many objects may fill the property

- **1**: an order must have exactly one customer
- **0...1**: a corporate customer may or may not have a single sales representative
- ***** : a customer need not place an order and there is no limit on how many might be placed

- The following terms refer to the multiplicity of an attribute:
  Optional: implies a lower bound of 0.
  Mandatory: implies a lower bound of 1 or possibly more.
  Single-valued: implies an upper bound of 1.
  Multivalued: implies an upper bound of more than 1, usually ∗.
  Advise: use a plural form for the name of a multivalued property.
- By default, the elements in a multivalued multiplicity form a *set*, so if you ask a customer for its `Orders`, they do not come back in any order.
  - If the ordering of the `Orders` has meaning, then add `ordered` at the end.
  - To allow duplicates, add `nonunique`.
  - To show the default explicitly: add `unordered` and `unique`.
  - Collections(=multisets) are unordered and nonunique: add `bag`.

Bidirectional associations

Pair of properties that are linked together as inverses



Car has a property *owner: Person[1]* and person has a property *cars:Car[*]*

This can also be implied by using a verb label such as “owns”

Operations

Actions that a class knows to carry out, they correspond to methods on a class

Operations that simply manipulate properties are often not shown as they can be inferred (setters and getters)

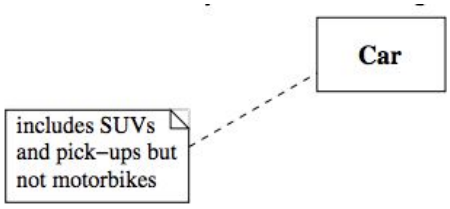
```
visibility name (parameter-list) : return-type {property-string}
```

e.g. : `+ balanceOn (date: Date) : Money`

- `visibility` is public (+), private (–), package (~), protected (#).
- `name` is a string.
- `parameter-list` is the list of parameters for the operation. Parameters are notated in a similar way to attributes:
  `direction name : type = default value`
  - `name`, `type`, and `default value` are the same as for attributes.
  - `direction` indicates whether the parameter is input (`in`, which is the default), output (`out`), or both (`inout`).
- `return-type` is the type of the returned value, if there is one.
- `{property-string}` indicates property values that apply to the operation.

Notes and Comments

Notes are comments in a diagram and they can be linked with a dashed line to the elements they are commenting



Dependency

A dependency exists if changes to the definition of one element (the **supplier** or target) may cause changes to the other (the **client** or source). They only occur on one direction.

With classes dependencies may exist for different 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 has many types of dependency, each with particular semantics and keywords

We should aim to minimize dependencies :)

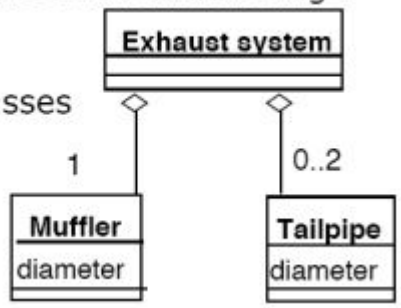
Constraint rules

The basic constructs of association, attribute and generalization do much to specify important constraints but they can't indicate every constraint

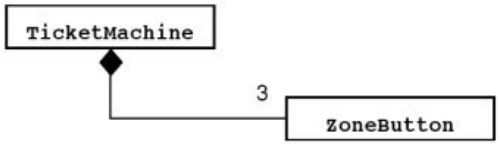
UML allows one to use anything to describe a constraint, the only rule is that **we put them in braces { },** we can use natural language, a programming language or the UML's formal Object Constraint Language (OCL)

Aggregation and Composition

- An *aggregation* is a special case of association denoting a "consists-of" hierarchy
- The *aggregate* is the parent class, the components are the children classes



A **solid diamond** denotes **composition,** a strong form of aggregation where the life of the component instances is controlled by the aggregate (The whole controls/destroys de parts)



In other words:

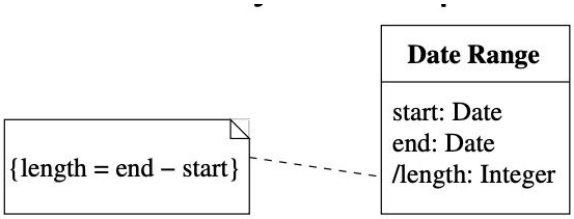
- Aggregation => Has-a relationship
- Composition => "Can't exist without" relationship

Derived Properties

Can be calculated on the basis of other values, and its not necessary.

Notation: `/property -- {constraint}`

- `/` specifies that the property is derived.
- `{constraint}`, if present, specifies how to calculate the value of the property.



Interfaces and Abstract classes

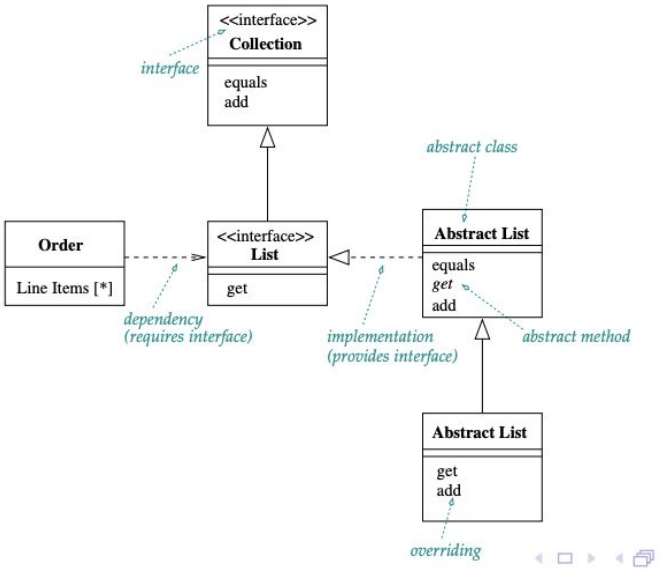
An **abstract class** is a class that can't be directly instantiated, typically an abstract class has one or more **abstract operations**, which are operations with no implementation.

One can indicate that a class is abstract by *italicising* the name or by using the label : {abstract}

An **interface** is a class that has no implementation (all its features are abstract), denoted by the keyword <<Interface>>

Classes have two types of relationships with interfaces:

- A class **provides an interface** if it is substitutable for the interface (generally to get extra features but not necessary)
- A class **requires an interface** if it needs an instance of that interface to work (depends on the interface)

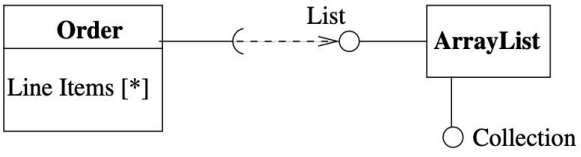


A more compact notation can be obtained by:

The fact that ArrayList implements List and Collection is shown by having ball icons (or lollipops)

The fact that Order requires a List interface is shown with the socket icon

Order and List then connected with the dependency arrow



Reference objects and value objects

All objects have an **identity**, this is more important when dealing with reference objects rather than value objects.

Reference objects are such things as "Customer"

- Identity is very important because one usually wants only one software object to designate a customer in the real world.
- Any reference to Customer will be through a reference pointer (all instances point to the same memory slot), hence if you change it, it will affect all pointers of Customer.
- Equality of reference objects is equality of their identities (same point in memory)

Value object are such things as “Date”

- There are often multiple value objects representing the same object in the real world, and these are all interchangeable copies
- Equality of value objects is equality of the values they represent (they might be stored in different places but have the same value)
- Value objects should be immutable, Modifying a value object is the same as discarding the old one and making a new one.

Generalization and specialization

Generalization expresses a “is-a” relationship, implemented by inheritance.
Inheritance simplifies the model by eliminating redundancy.

Semantically an object of a subclass can be substituted for a superclass object

Multiple Classification vs Multiple Inheritance

Multiple Inheritance: a type may have many 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 (not possible in OO)

We must be careful as both are referred to as a “is-a” relationship.

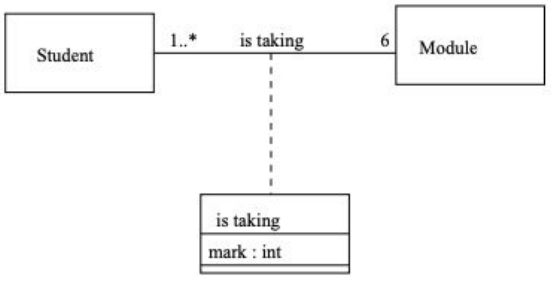
Dynamic Classification

Allows objects to change class type within a number of subtypes using the keyword **dynamic**, while **static classification** does not**.**

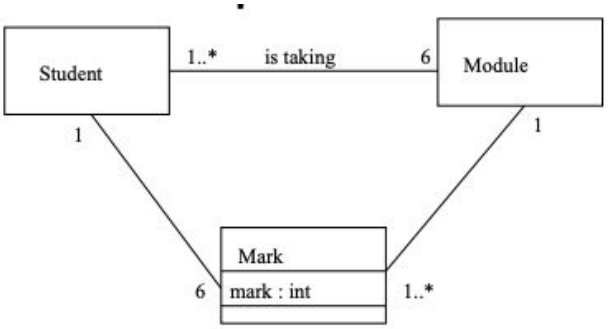
Although useful for conceptual modeling the bast majority of UML diagrams use single static classification.

Association Classes

Allow one to add attributed, operations and other features to associations.



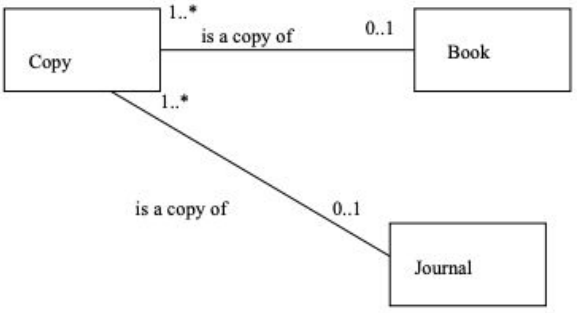
An association class can be replaced with multiple associations

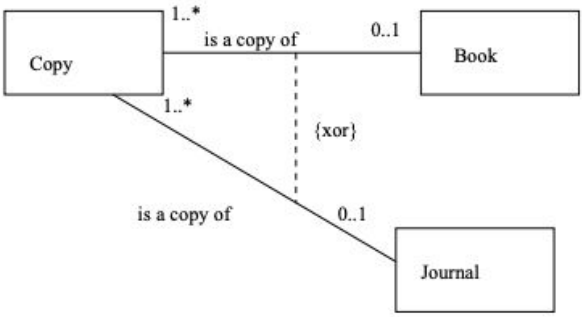


We must note that by using association classes, we are adding an extra constraint. There can only be one instance of the association class between any two participating objects.

Object Constraints

Constraints can be added to associations to tighten the semantics (like on the right)



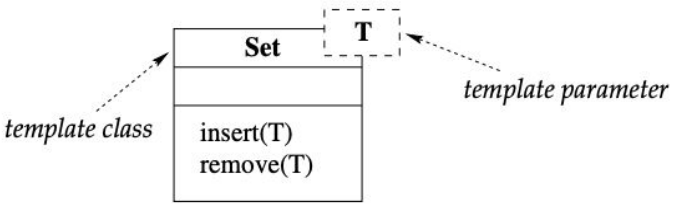


Template (parameterized) classes

Several languages (like C++) have a notion of a **parameterized class** or **template**, which is useful for working with collections in strongly typed languages.

Example: define behavior for sets in general by template class `Set`

```
class Set <T> {
  void insert (T newElement);
  void remove (T anElement); ...
```



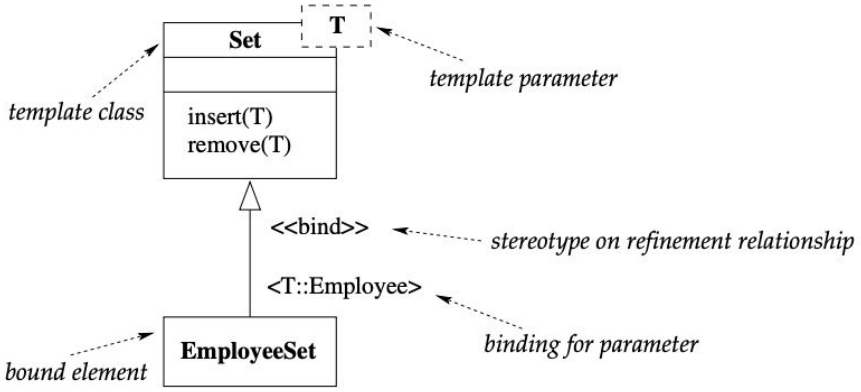
The use of a parameterized class such a as Set <Employee> is called **derivation** and can be shown in 2 ways:

- Derivation expression

  `<parameter-name::parameter-value>`

- Reinforcing the link to the template and renaming the bound element



Stereotype <<bind>> on refinement relationship indicates that EmployeeSet will conform to the interface of Set

EmployeeSet can be considered as a subtype of Set, but EmployeeSet can't add features to the implementation of Set (otherwise it would be a case of subtyping)(subtyping == using an interface)

Directed Names

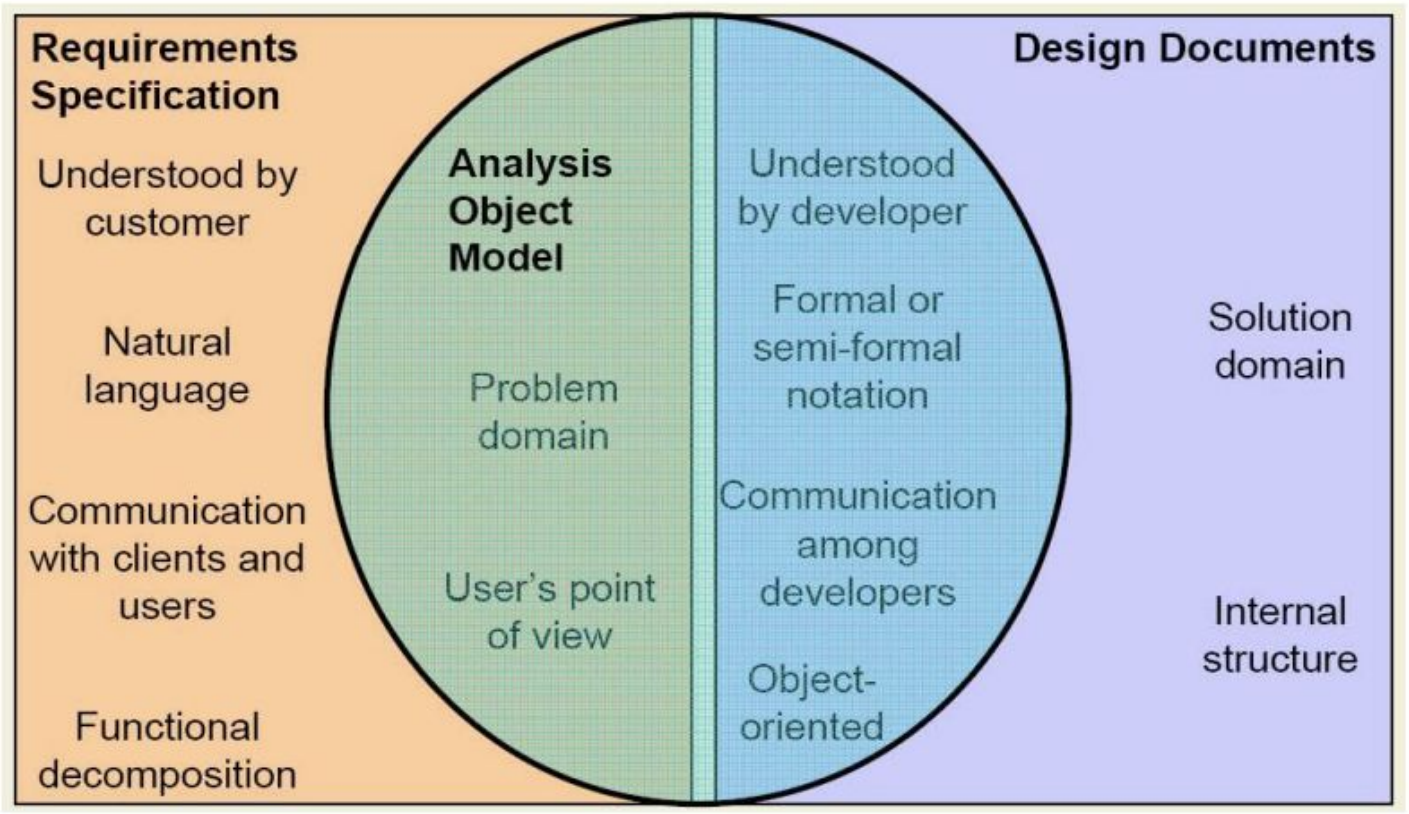
Describes the direction that a name should be read, it is independent of navigation and has no semantic consequences

Unit 4 Analysis Object Model, OCL, CRC Cards

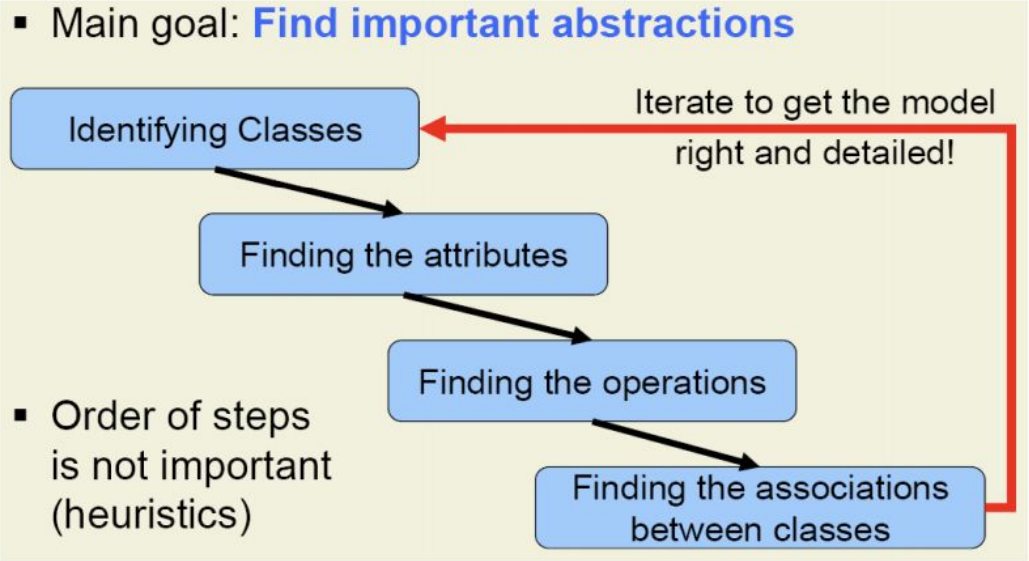
Transforming Use Case diagrams to Class Diagrams.

Analysis Object Model

Sometimes we have seen that what is verbally required to what actually gets implemented is quite different.
The abstraction of the process is somewhat like this.



Traditionally, software engineering has used a waterfall model to transform natural language requirements to system requirements and ultimately code.



Design Pattern: 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 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.

What is a good class model?
- Objects should satisfy the desired requirements
- Classes should represent significant classes of objects in the domain to improve sustainability and maintainability.

How does one create a good class model?
- Use data driven design: identify all system data and divide it into classes. Afterwards consider operations.
- Responsibility driven design: start with the operations or even the responsibilities.

Procedure 1: classes as nouns

1. Look at noun and noun-phrases used in the requirements analysis (use the singular).
2. Consolidate the results, delete those whose result is:

a. redundant
b. unclear (or else clarify)
c. an event or operation (without state, ie a function)
d. a simple attribute
e. outside of system scope

Noun-verb analysis (Abbott's textual analysis)

Take the flow of events or text and find:
- Nouns which are good candidates for classes.
- Verbs which are good candidates for operations.

This works well for problem statements and use cases (small texts).

Part of speech	Model component	Example
▪ Proper noun	▪ Object	▪ Jim Smith
▪ Improper noun	▪ Class	▪ Toy, doll
▪ Doing verb	▪ Method	▪ Buy, recommend
▪ being verb	▪ Inheritance	▪ is-a (kind-of)
▪ having verb	▪ Aggregation	▪ has a
▪ modal verb	▪ Constraint	▪ must be
▪ adjective	▪ Attribute	▪ 3 years old
▪ transitive verb	▪ Method	▪ enter
▪ intransitive verb	▪ Method (event)	▪ depends on

- Natural language is imprecise.
- You normally have to identify and standardize terms and rephrase to clarify requirements specification.
- Normally there are many more nouns than relevant classes.

- Eliminate synonyms
- Many nouns correspond to attributes

Procedure 2: CRC cards
Alternative that was suggested pre-UML. For each class we have a card which has:
- Class: Name
- Responsibilities: of the object
- Collaborators: helpers that aid in fulfilling responsibilities.

If there are too many responsibilities or collaborators, create net classes.

It was tested by distributing the cards to the devs team and playing out each use case scenario through. By doing, so they could discover missing responsibilities or collaborators. Afterwards, we add attributed and methods.

LIBRARYMEMBER	
Responsibilities	Collaborators
Maintain data about copies currently borrowed Meet requests to borrow and return copies	COPY

COPY	
Responsibilities	Collaborators
Maintain data about a particular book copy Inform corresponding BOOK when borrowed and returned	BOOK

Different Types of Objects

Entity Objects: represent the persistent information tracked by the system.
- They are the application domain objects or business objects.

Boundary Objects: represent the interaction between the user and the system. *Objects that live at the boundary of your implementation.*

Control Objects: represent the control tasks performed by the system. *Closer to the admin*

The objects which are further away from the user are more likely to change.

Heuristics to identify Entity Objects

- **Words** that the developers or users must clarify to understand the Use Case (e.g. account).
- **Recurring nouns** in the Use Case (e.g. card).
- **Real-world entities** that the system must track (e.g. cash dispenser).
- **Real-world processes** that the system must track.
- Data sources or sinks (e.g. host).

Account

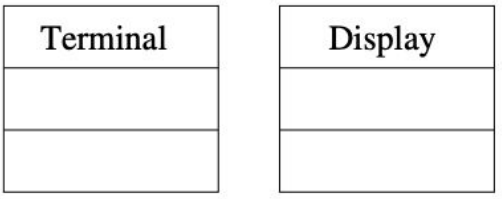
Currency

Use cases and initial analysis models can be improved by cross-checking:

- Which Use Case creates this object?
- Which actors can access this information?
- Which Use Cases modify and destroy this object?
- Which actors can initiate these Use Cases?
- Is this object needed? (Is there at least one Use Case that depends on this information?)

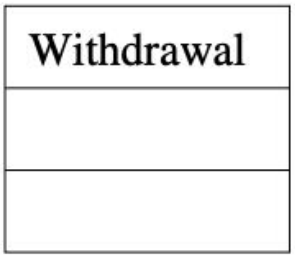
Heuristics to Identify Boundary Objects

- Boundary objects collect information from actor.
- Boundary objects translate information into format for entity and control objects.
- Boundary objects do not model details and visual aspects (e.g. menu item, scrollbar).
- Each actor interacts with at least one boundary object.
- 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)



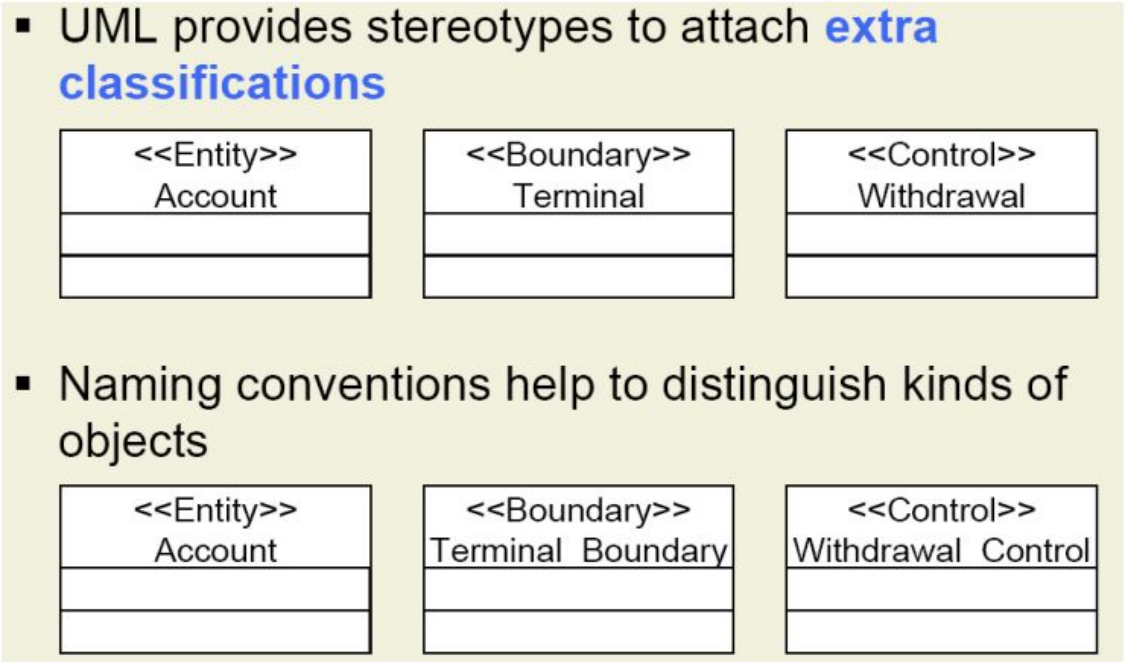
Heuristics to identify Control Objects

- Control objects **coordinate boundary and entity objects**.
- Control objects usually **do not have a concrete counterpart in the real world**.
- Control objects are typically created at beginning of Use Case and exist to its end.
- Control objects collect information from boundary objects and dispatch it to entity objects.
- Identify one control object per Use Case.
- Identify one control object per actor in the Use Case.
- Life span of a control object should cover the extent of a Use Case or user session.



Examples: Sequencing of forms, undo and history queues. Dispatching information in distributed systems.

Stereotypes and conventions



This stereotypes are not require but they help understand the semantics of the system.

A **UML package** is a folder where you can put in classes or class diagrams (similar to what happens in Java). You can then import these. The common rule is not to put more/less than 7+-2 classes in one package.

Ways to find objects: summary

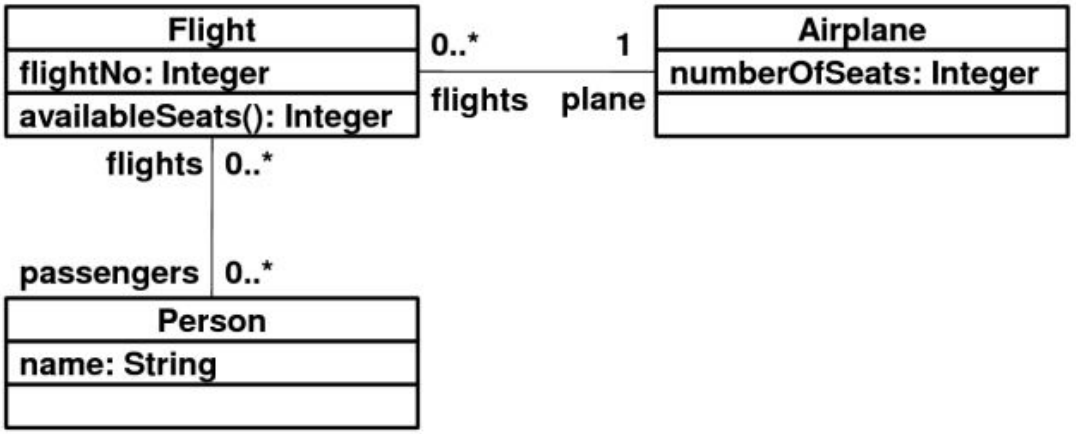
Syntactical investigation with Abbott's technique:
- In the problem statement.
- In the flow of events of Use Cases.

Use of various knowledge sources:
Application knowledge: Interviews of users and experts to determine the abstractions of the application domain.
Design knowledge: Reusable abstractions in the solution domain.
General world knowledge: Use your empirical knowledge and intuition.

The Object Constraint Language OCL

The OCL is a modelling language with which one can build software models. OCL expressions allow one to explain things that often cannot be expressed in a diagram.
- **Constraint**: restriction on one or more values of (part of) an object-oriented model or system.
- **Other information:** expressions also allow for defining queries, referencing values, stating conditions and business rules in a model, ...

Example: in this case we see that a flight can have 0 to * unlimited passengers.



This is not realistic, rather what we want to say that is that a plane can have as many passengers as it has seats. Then OCL can add this description:

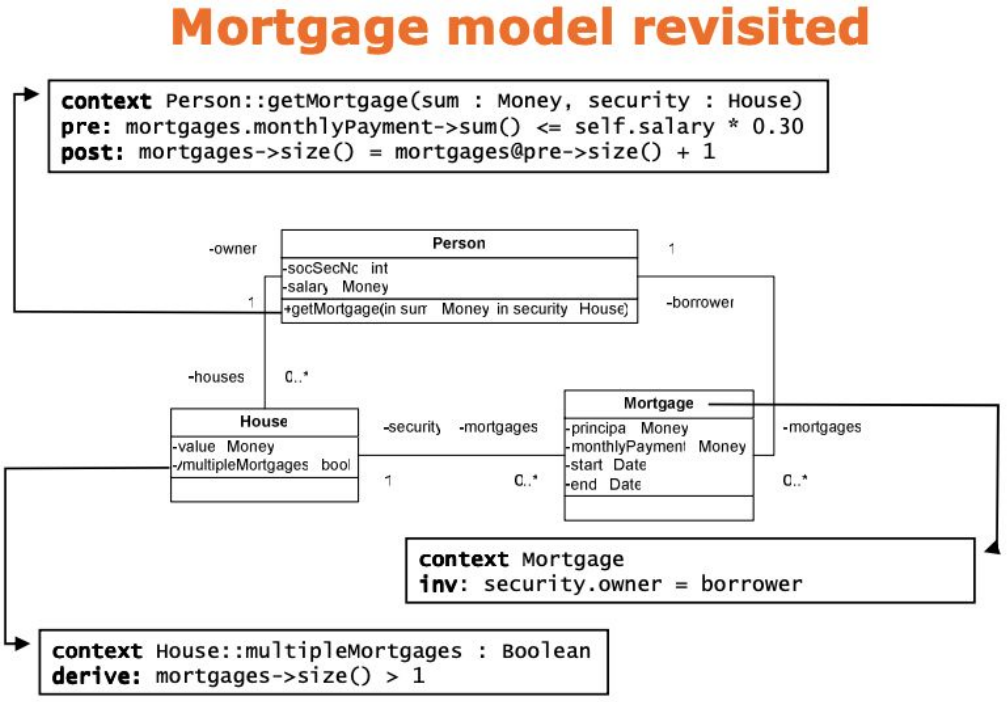
```
context Flight
inv: passengers->size() <= plane.numberOfSeats
```

We use OCL instead of adding constraints in the natural language next to a UML relationship. Another example:

The start date of a mortgage should come before its end date (assuming < works on dates).

```
context Mortgage
inv: start < end
```

After we have added these, our diagram looks like the following.



The combination of OCL + UML makes diagrams which are ambiguous, complete and consistent. It avoids underspecification that is mathematically based which allows for automated tools to check the diagram.

OCL talks about what it wants not how to achieve it. It can be used not only to specify constraints but to indicate the value or object within a system.

- It has types i.e. 1 + 3 is a valid expression under Integer type.
- When the value of an expression is of Boolean type, it may be used as a constraint.

A value can be:

- A simple value
- A collection of values
- A reference to an object
- A collection of references to an object

An expression can:

- Represent a Boolean value used as a condition in a statechart or a message in an interaction diagram
- Be used to refer to a specific object in an interaction or object diagram.
  For example, the following expression defines the body of the availableSeats() of the class Flight:

```
context Flight::availableSeats() : Integer
body: plane.numberOfSeats - passengers->size()
```

OCL is a constraint and query language at the same time. It is based on set theory and predicate logic. However, The notation, however, does not use mathematical symbols. Hence, OCL has the rigor and precision of mathematics, but the ease of use of natural language.

- OCL even allows users to define their own syntax (as long as this syntax can be mapped to the language structures defined in the standard).
- OCL is a **strongly typed language**: Hence, OCL expressions can be checked during modeling, before execution of the system, and errors in the model can be removed at an early stage.
- OCL is a **declarative language**: An expression simply states what should be done but not how.

OCL expressions have no side effects: evaluating an OCL expression does not change the state of the system. That we should calculate the number of seats is defined but how is not.

Lecture 5 Web Application Development

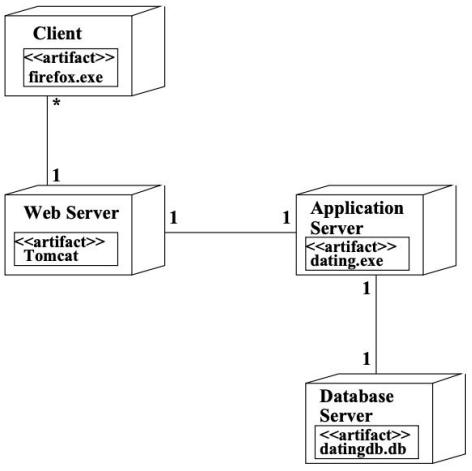
Motivation and Introduction

Web applications are software systems that use world wide web to interact with users

The image on the right is the typical structure of a web application as a **deployment diagram**. There might be many different clients running in different browsers on different client machines.

Web server: takes the information you type in by TCP, IP communication. Were web server software is hosted.

Application server: The hosted components might actually run in an application server that usually processes data in a database and makes changes to the stored information in a **database server**



Web, application and database servers might be in the same machine but this is generally not the case in bigger systems for efficiency and security

Web server used to take the information, application server to process the information.

Model Driven Development and Application Development

MDA provides flexibility by defining Platform Independent Models (PIMs) that specify business data and business rules of a system, independently of particular technologies.

Web applications often have common structure and elements, which means that systematic development process can be applied for these applications

Development of web applications: 3 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 and behaviour of web pages interfacing to clients, e.g., by using animation software such as Flash.
3. Deciding on information content of web pages, choice of words to use, what information to emphasise, etc.

Properties important for web applications

Portability

Means that a system can be moved to different execution environments and behave in the same way in the new environment as in the old

Particularly an issue for the UI (web pages of a system should appear in a similar way and provide identical behaviour regardless of the browser)

Usability

Means that the web interface does not require unreasonable effort to use, and that users can access provided functionality without excess effort.

Usual principles of usability of UIs apply (*this has appeared in exam before*):

- clear information should be provided,
- feedback should be provided after making a data entry,
- related functions should be grouped together, etc.

Web-specific usability guidelines include that length of navigation paths between parts of an interface used by same user in same session should be minimised. (users shouldn't be wondering around)

Accessibility

Means that a web interface can be used by users of different ability - such as visually impaired, color blind, deaf, or senior citizen users - as effectively as by non-disabled users.

A general MDA dev process for web apps

1. Define a PIM abstract data model of the entities involved.
2. Define PIM use cases describing operations required from system.
3. Design outline web pages, based on what **operations** are to be provided (from step 2), e.g. an input page (such as a form) should only require users to enter the minimal information necessary to support operation it is involved in. Define web **page invariants** (e.g., that a name input field should be non-empty) and any **client-side scripts** to check/enforce these.
4. Define user interaction sequence of web pages, using state machines.
5. Define visual design and information content of web pages — these should be consistent in style
6. A complete prototype of client side of system can be produced at this point and reviewed. Check that accessibility and portability requirements have been met, and do usability trials with typical users.
7. Define which web pages are to be hard-coded in HTML, and which are to be generated by server-side components.
8. Transform data model into a Platform Specific Model (PSM) appropriate for data storage approach to be adopted (e.g., relational DB).
9. Define server-side response pseudocode (or full code) for each operation:
    a. extraction of parameters from the request;
    b. checking constraints on parameters;

c. processing of the operation, usually involving database interaction;
d. and construction of a result/next web page.

10. Define data repository queries/updates. For implicit associations these can be based on the constraints defining the association, as for matches in the dating agency search example.
11. Define database interface(s) to support the operations required from server-side functional core components.

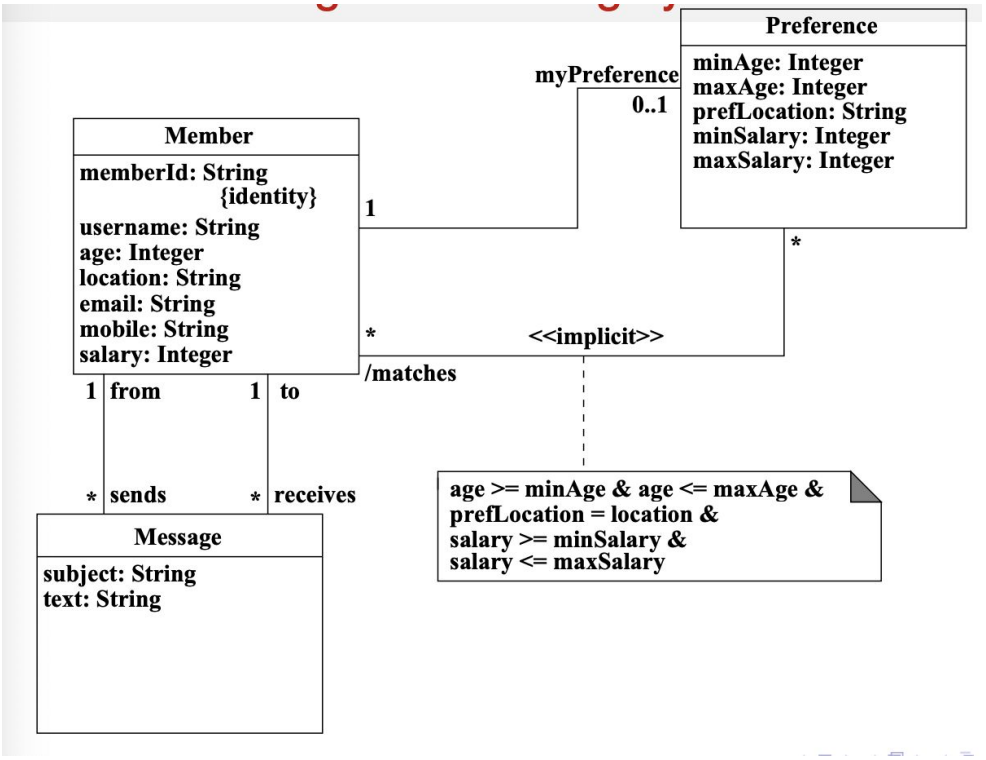
Web application specification

The specification of the functionality of a web application consists of two PIMs:

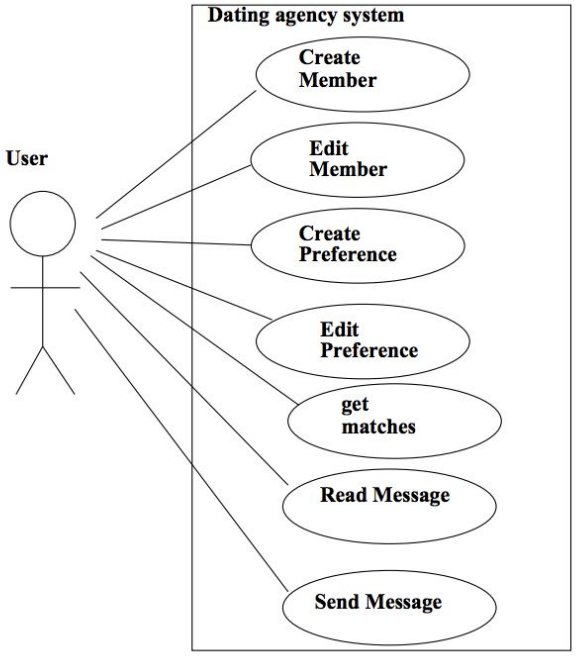
- Class diagram, showing data to be stored and processed by the system
- Use case diagram, showing the operations which the user will be able to perform upon the system

The class diagram should also include documentation of any constraints of the data (like in <<implicit>> below)
<<Implicit>> is a domain-specific stereotype and implies that an association is calculated on demand, rather than stored persistently in a database

i.e. class diagram of a dating system



i.e Use case diagram for a dating system



Web application design

We will focus on the following design techniques:

- Web page design
- Interaction sequence design (using state machines)
- Architecture diagrams.
- Transformation from analysis to design models (using class diagrams).

These are independent of particular server-side programming technologies such as Java Server Pages (JSPs), Servlets, PHP or Active Server Pages (ASPs),

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

The usual usability guidelines for user interfaces also apply specifically to web application interfaces

Web page design issues

- Use clear and simple labels for fields, making clear which are mandatory
- Avoid exposing internal ids, unless these are generally used in the domain (like ISBNs or national Id)
- It is sometimes possible to use default values if user does not fill in a field (0 for minimum price range)
- Avoid reloading an entire web page if only part of page changes
- A web application should provide clear and immediate feedback to user concerning incomplete or incorrect data (Many web applications provide very poor usability because of this issue)

Client-side Scripting

Some data validation and dynamic user feedback can be provided client side with client-side scripting languages such as JavaScript

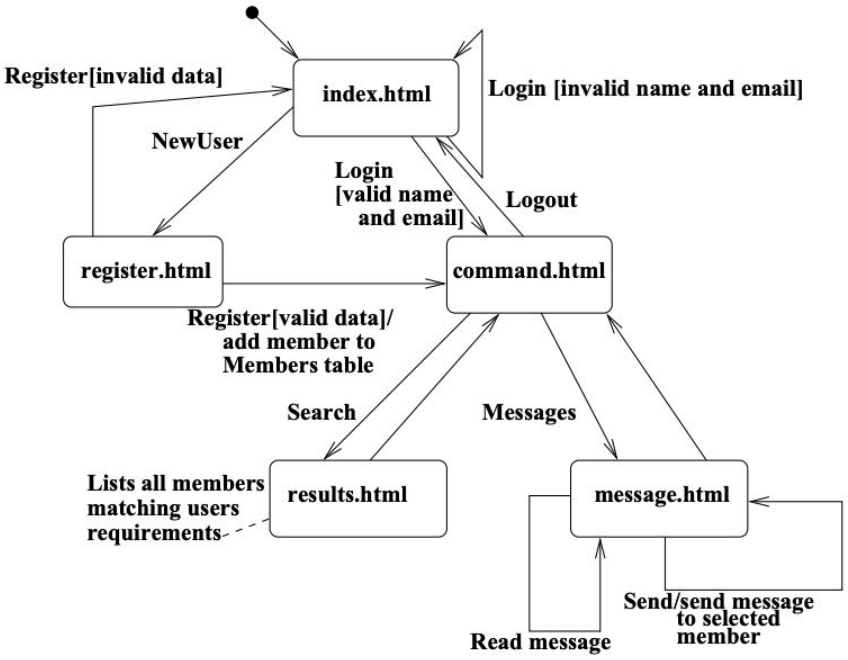
A scripting language is a simplified programming language designed for tasks such as checking that a string is non-empty or that a string represents a number.

A limitation of JavaScript is that some browsers might not support JavaScript (rare) or might have JavaScript execution explicitly switched off. Therefore additional data validation should take place on the server.

Interaction design using state machines

- **Class diagrams** can be used to describe data of web applications: the contents of forms and database tables.
- **State machines** can be used to describe interaction behaviour of a web application: what sequence of pages are displayed to user, and the effect of user commands.
- **States** of a state machine correspond to web pages displayed to user: name of page is given, plus a summary of its content.
- **Transitions** are labelled with events that correspond to user commands or links that can be selected in source state (web page).
  Effect of commands is described, and target state is the next web page shown to user.

Sequence Diagram



Represent behaviour in terms of interaction and complement the class diagram

They are useful to find **missing objects** or to detect and **supply operations for the object model.**

Time consuming but often worth the investment

Events, actions and activities

Event: something that happens at a point in time

Action: operation in response to an event

Activity: operation performed as long as object is in some state e.g. object performs a computation without external triggers

State

An equivalence class that abstracts a number of attribute values from an object

e.g. Books are either borrowable or not, all borrowable books are in the same equivalence class (borrowable) independently of their author, title, …

Dynamic Modelling: State-oriented

Models dynamics aspects of systems: **control** and **synchronization**

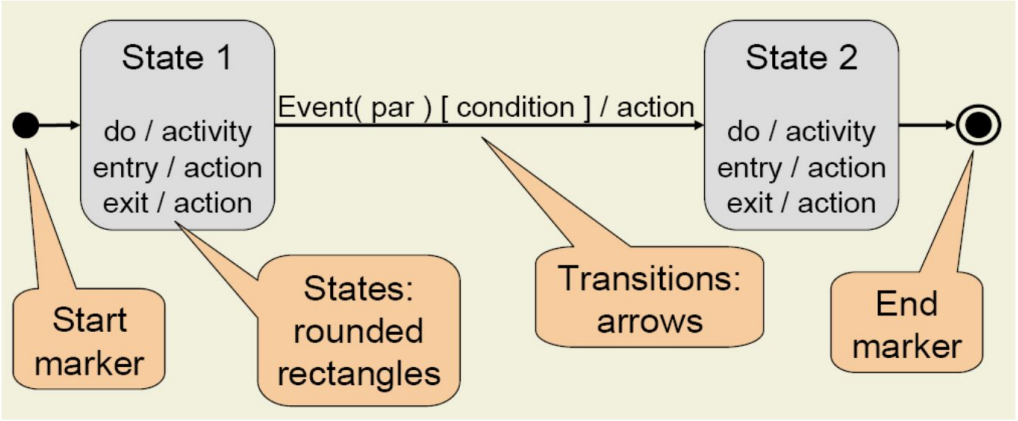
- What are the **states** of the system?
- Which **events** does the system react to?
- Which **actions** are possible?
- When are **activities** started or stopped?

Such models correspond to **transition systems**, also called **state machines**.

State-machine modeling

State-machine modeling: model the behavior of the system in response to external and internal events.

UML State Diagrams
Note that there are different names for state machines/charts/diagrams but they refer to these.

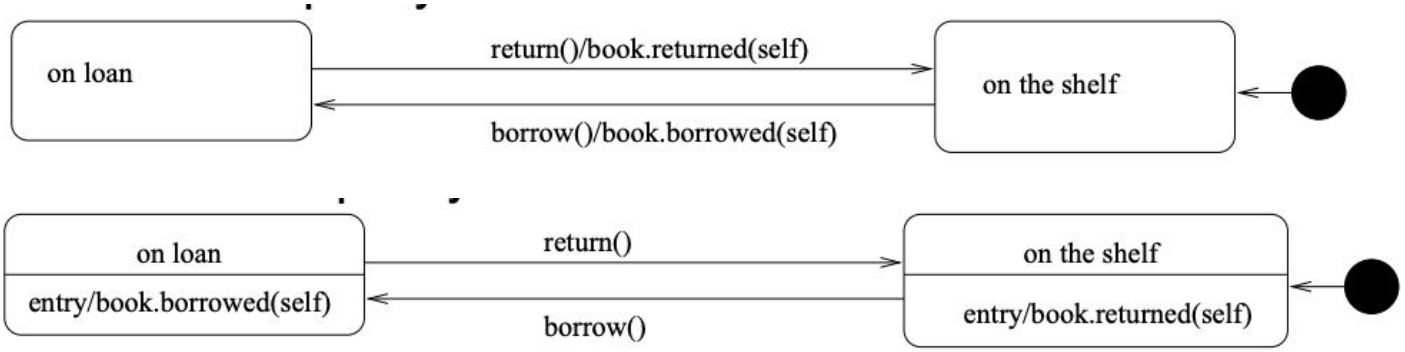


Relates events and states for a class, they are often called **state charts** or **statechart diagrams.**

They explain **how** and **when** an object reacts.

Actions

Transitions and states can also specify actions with the syntax Event/Action



What does a statechart mean?

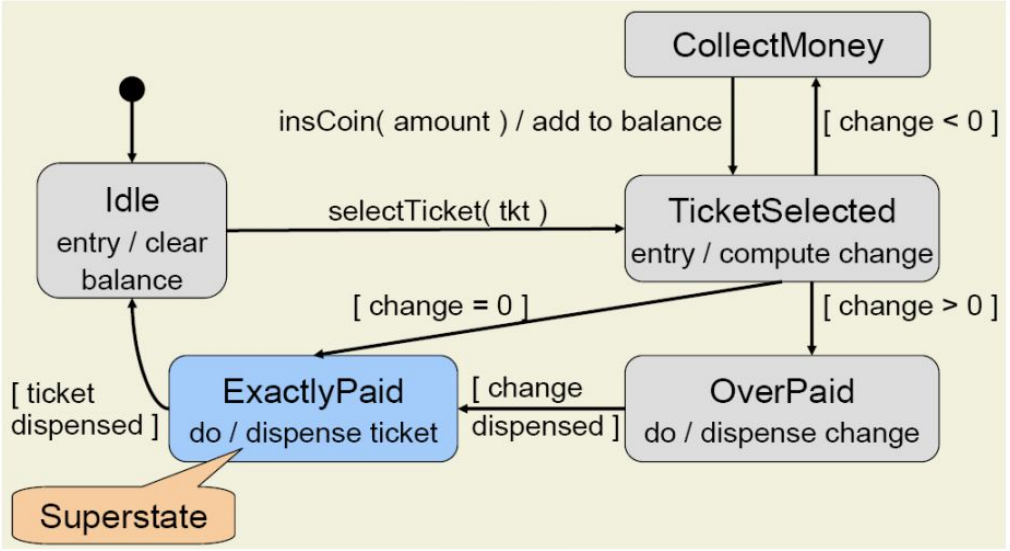
- Semantics as a transition system:

- Transition occurs when:
  - System is in `S1`.
  - Event `E` occurred in the last step.
  - Condition `C` holds.
- Events immediately follow.
- A formal semantics (with time) is actually rather subtle!

Nester State Diagrams

Activities in states can be **composite items** that denote other state diagrams

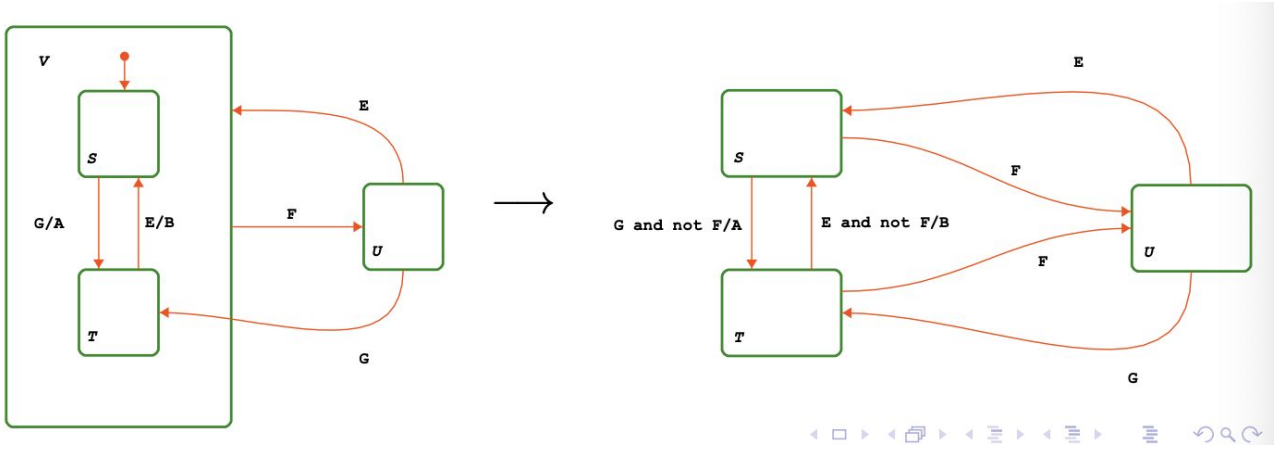
Sets of substates in a nested state diagram can be denoted with a superstate



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)

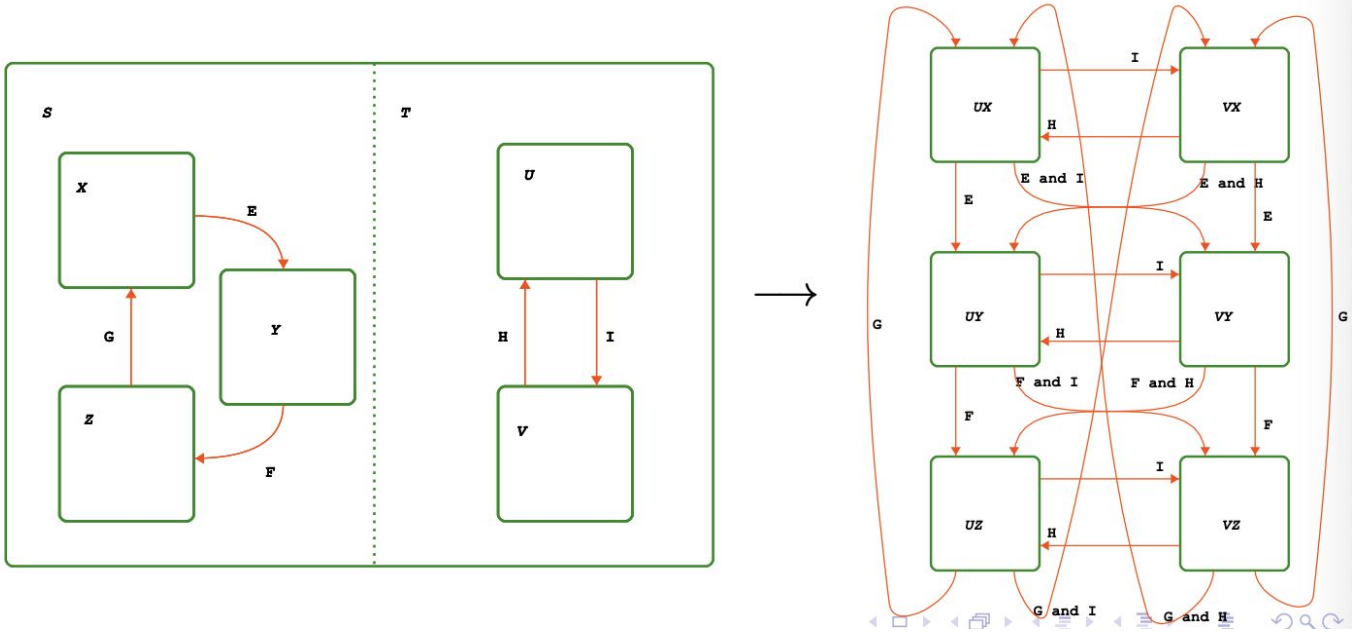
Hierarchy

- States are tree structured
- System is in a **ground state** and in all parent states
- Higher level transitions have priority over lower level ones (event F has priority over E or G)



Parallelism

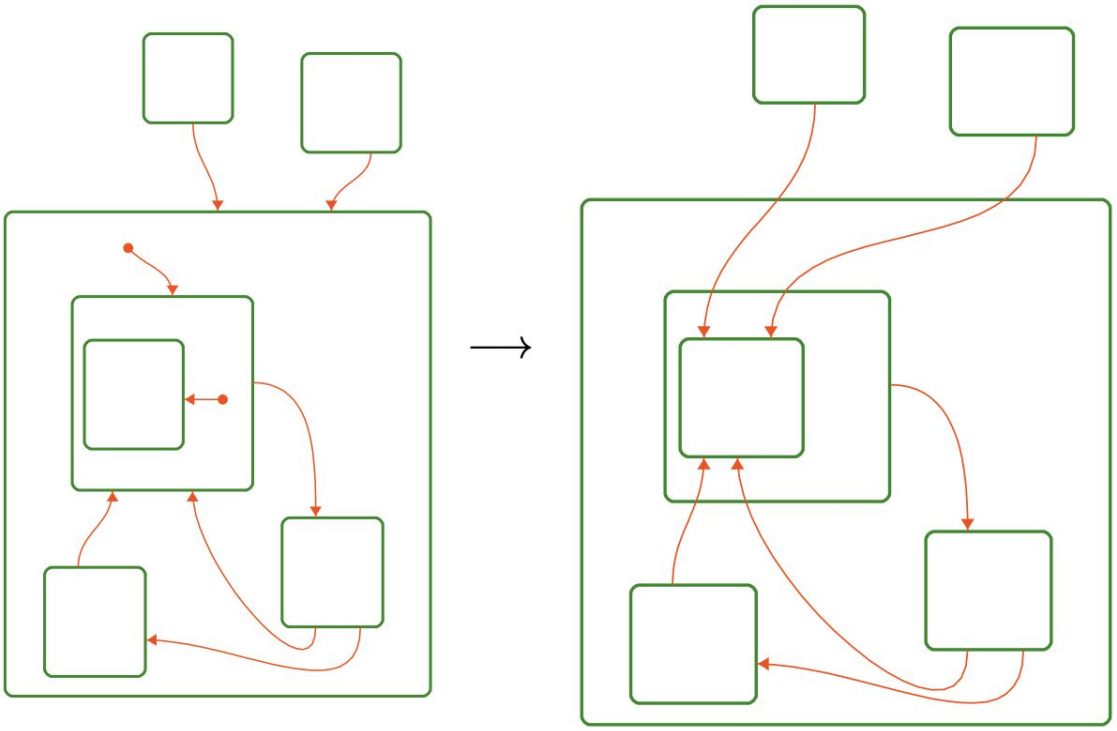
- OR-states: system in at most one state of current level.
- AND-states: simultaneously in all states of current level
  $\Longrightarrow$ parallelism.



The one on the right is an **and-state**

Parallelism complicates semantics

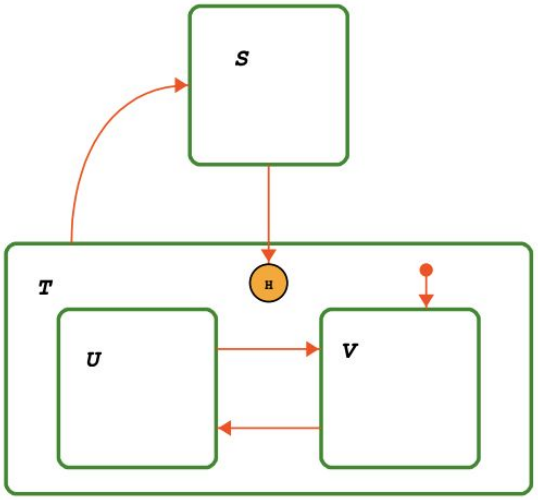
Default connector



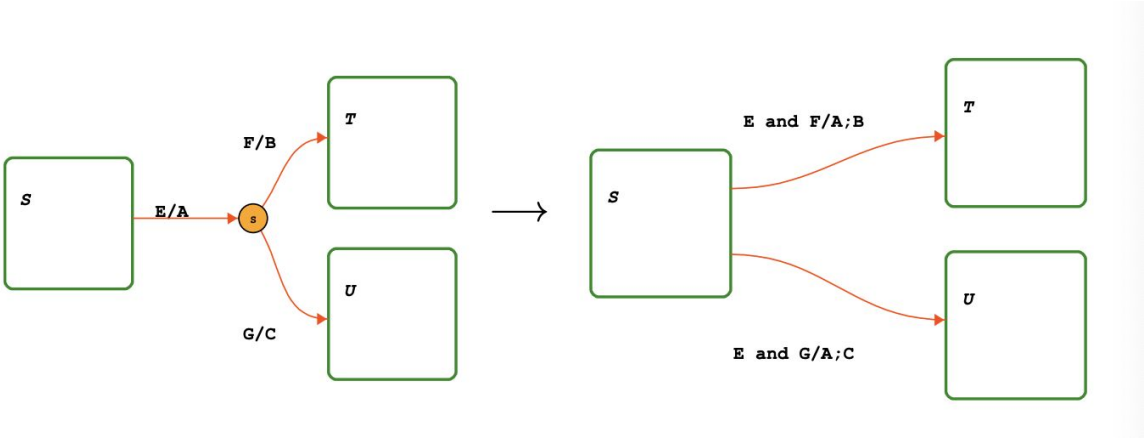
Due to the hierarchy, the inner state is a default connector as you will always start from there

History Connector

Describes which state was last active (not very common)

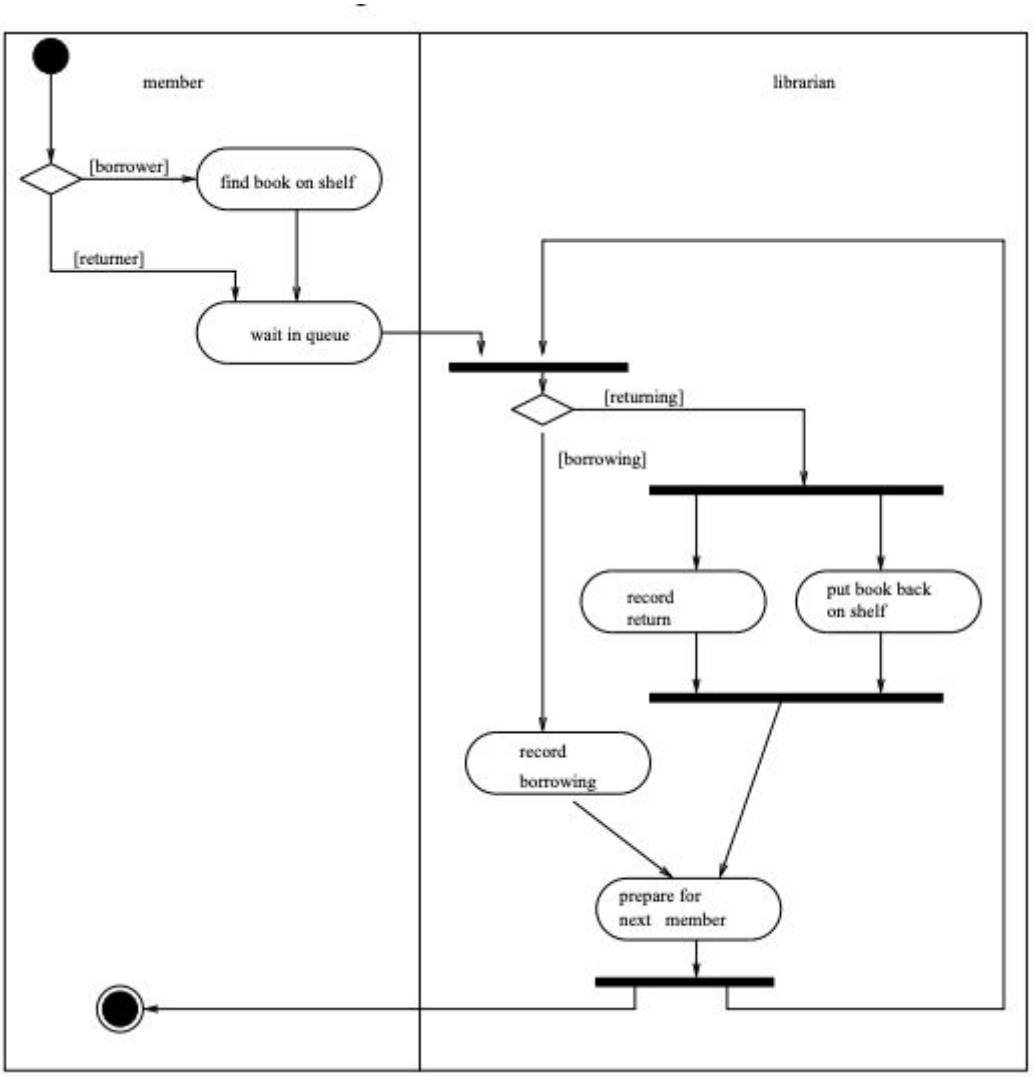


Switch Controller

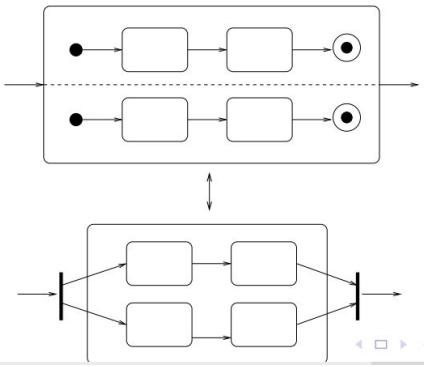


Activity Diagrams

Simple alternative to statecharts, emphasizes synchronization **within** and **between** objects
Example of “work in a library”:



- Syntax:
  Activity: a kind of state, left when the activity is finished.
  Transition: normally not labeled, since the event is the end of an activity.
  Synchronization bars: describes synchronization points.
  Decision diamonds: shows decisions, alternative to guards.
  Start and end markers: like in statecharts.
  Swim-lanes: shows which object carries out which activity.
- Semantics: More-or-less intuitive. Can be translated into statecharts:



State Diagrams vs Sequence Diagrams

- **State Diagrams** help to identify changes to an individual over time
- **Sequence Diagrams** help to identify the
  - Temporal relationship between objects
  - Sequence of operations as a response to one or more events

Practical tips for dynamic modeling

- Construct dynamic models only for classes with **significant** dynamic behaviour
- Consider only **relevant** attributes
- Look at the level of detail necessary when deciding on actions and activities
- Reduce notational clutter (try to put actions into superstate boxes)

Unit 6: 5-Tier JSP, Servlet

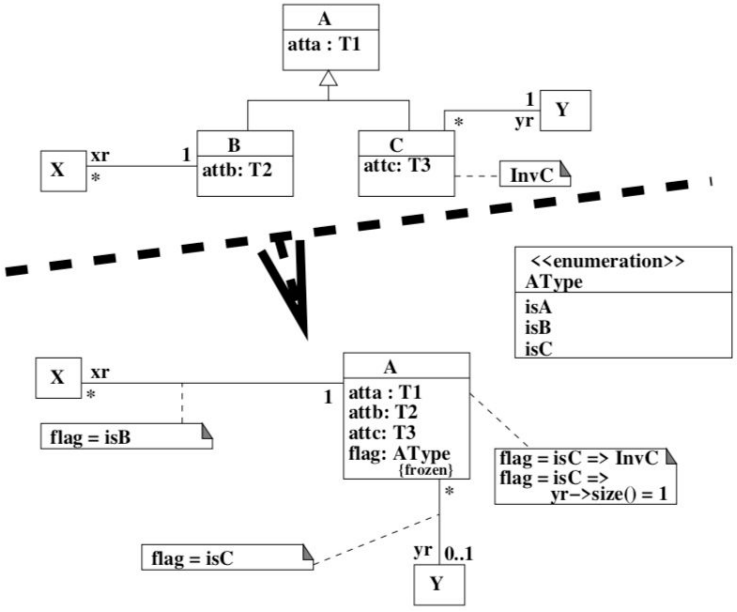
Java Servlet Pages: server side
Servlet: on the client side

Changing Class Diagrams (Analysis) into Design Models

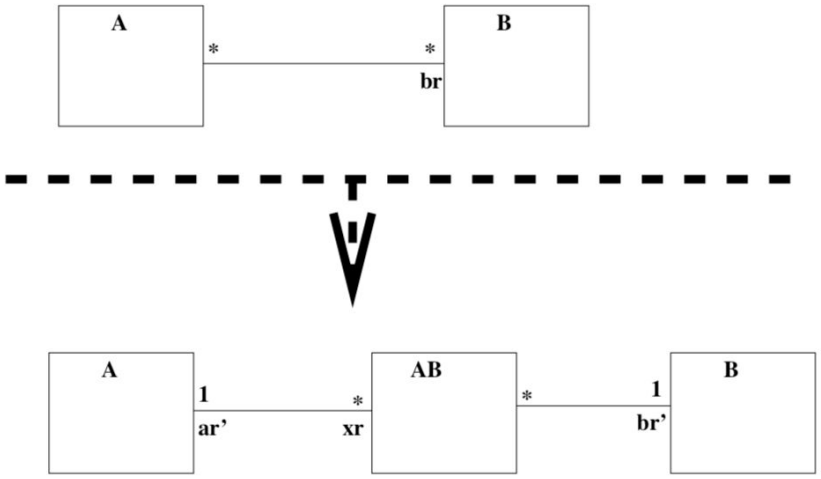
Analysis class diagrams need to be refined to model for a particular implementation platform. This is a special case of PIM to PSM transaformation of MDA

For relational databse impleementation use transformations:

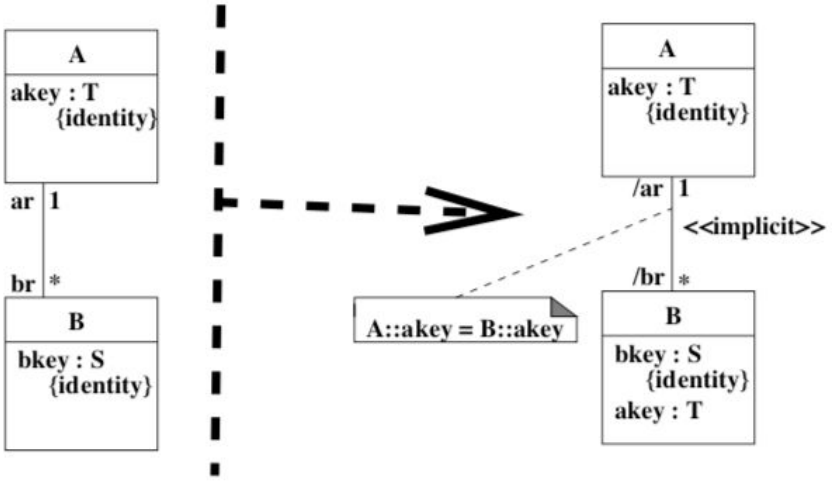
- Remove inheritance by merging subclasses into their superclass, or represent the classes in separate tables, using a *-1 association from sub-to superclass.



- Introduce primary keys for all persistent entities that do not already have an *{identity}* attribute
- Replace *-* explicit associations by two *-1 associations and an intermediate class. Create another table for the class on the left and create a table that maps the class in the many classes on the left to a single class entry on the right.



- Replace explicit *-1 associations by a foreign key from the * entity to the 1 entity

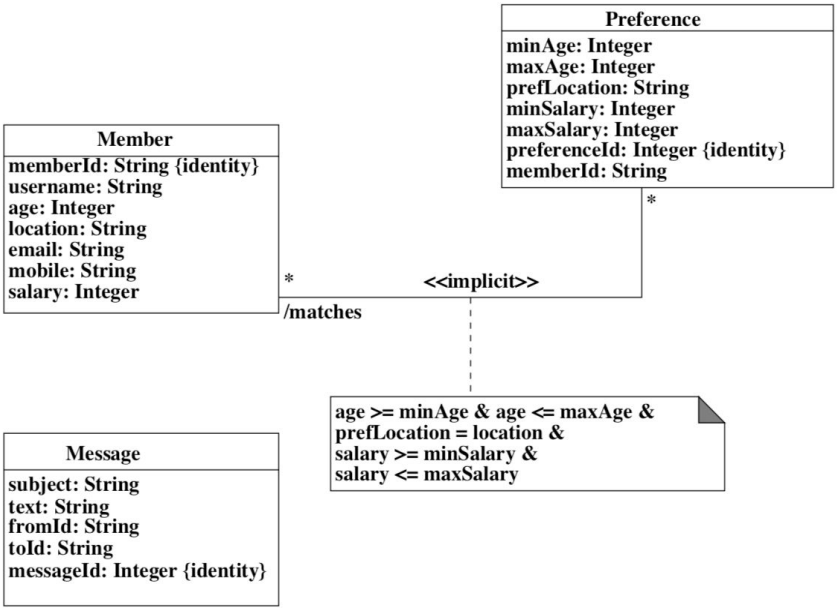


Essentially, add the primary key of the attribute of A to the table in B, thus removing the need of a table of relationships between A and B.

The resulting class diagrams can be directly translated to relational database tables.

- Each transformed persistent class *C* is represented as table with a column for each attribute, and primary keys the identity attributes, or (as compound key) a group of attributes stereotyped as identity.
- Each many-one relationship from *A to B* is represented as foreign key from table for A to table for B.
- Each persistent implicit association is expressed by SQL predicate to calculate entity instances in association using a SELECT query.

These transformations can sometimes be applied automatically.



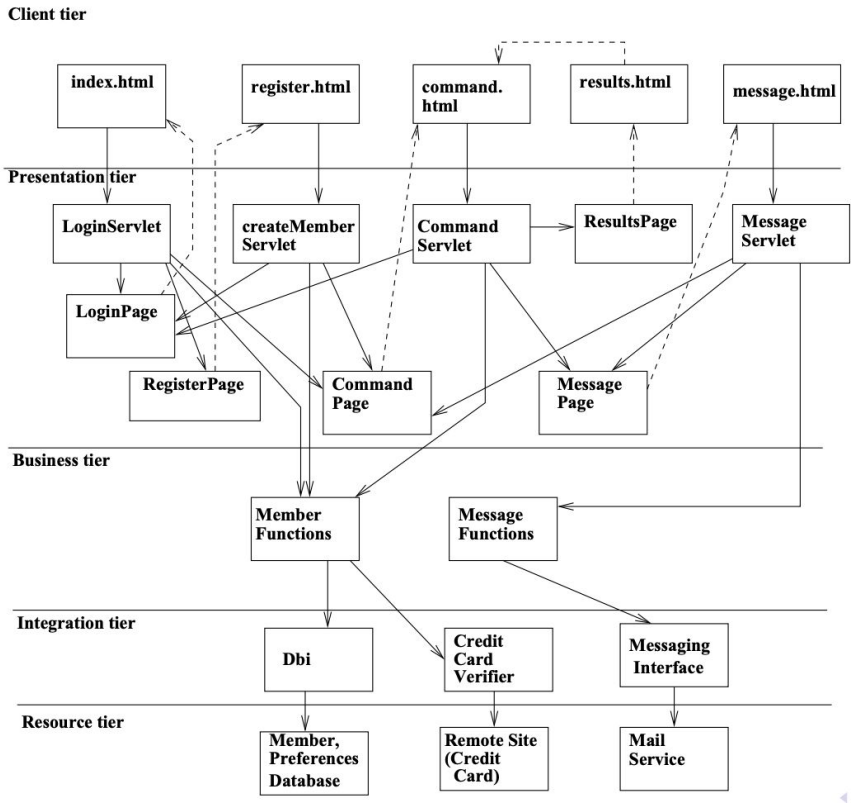
Design of functional core of web applications

The functional part of a web application can happen both on the client side (React) or on the server side.

Typical functional components of a web applications are:

- **Web Pages**: written in HTML/XHTML, Javascript or other scripting code
  - They present information to users and receive information from users
  - Send form data to the server using HTTP
  - Carry out simple processing (ie check validity of input data)
- **Controller:** computes, or other pure processing server-side elements. They process submitted data and take actions on the server side on the system.
  - In Java these are typically defined as **Servlets**, which are java classes that provide specific operations for receiving and responding to HTTP Requests
- **View/Presentation**: server-side elements, which generate web pages and take actions in response to submitted data
  - In Java they are typically defined as **JSPs (Java Server Pages)** which are HTML pages enhanced with server-side Java code.
- **Resources**: such as databases or remote web services which this system uses

These components can be categorized into 5 layers (tiers)



DBi = Database Interface

Client tier = user interface

Presentation tier = some functionality to produce information to the business tier.

Business tier = basic functions that we carry out.

Integration tier = different technology interfaces called by our server to access/read/write resources.

Resource tier = where resources reside

Component Communication

These components can communicate/invoke each other in the following ways:

- HTML pages can transfer to other web pages by naming them in a link (dashed lines in diagram), with an <a /> tag

```
<a href="nextpage.html">Go to next page</a>
```

- HTML pages can invoke Servlets or JSPs by naming them in the ACTION clause of a FORM element (<form> tag)
    - This identifies which servlet or JSP the data of the form should be sent to, when the form is submitted.
    - The method indicates how the data should be sent, either GET or POST techniques

```
<form action="http://www.server.com/servlet/ServletName"
  method = "GET">
```

- Servlets can invoke other servlets or JSPs by forwarding requests to them

```
public void doGet(HttpServletRequest req,
                  HttpServletResponse res)
{ ... process req ...
  res.sendRedirect("Servlet2");
}
```

    - ***doGet*** method responds to web requests. *req* contains the packet of the data sent from the client side with the request (normally values, strings entered in form fields).

    - ***sendRedirect*** method transfers request handling to the named servlet or JSP. Static web pages can be used as argument of this method.
    - Another way of forwarding requests and responses is

      ```
      req.getRequestDispatcher(resource).forward(req,res);
      ```

      where *resource* is a string naming a web component in the current web application. (solid lines)
- JSPs can forward to other JSPs or to servlets (the latter is unusual):

  ```
  <jsp:forward page="next.jsp"></jsp:forward>
  ```
- Servlets and JSPs can invoke normal Java methods of Java objects, such as database interfaces or auxiliary Java classes, called beans. (solid lines)
    - JavaBeans are reusable software components for Java:
        - They are classes that encapsulate many objects into a single object, the bean).
        - They are serializable, have a 0-argument constructor, and allow access to properties using getter and setter methods.

In architecture diagrams we use dashed arrows for HTML links, and generation of web pages, and solid arrows for invocation/forwarding.

Server-side processing

The server side of a web application has the following main tasks:

- Processing data sent from the client side (including checks on correctness of the data and security checks)
- Modifying or retrieving data in lower tiers of the server side, such as a database.
- Invoking operations of lower tiers, including remote web services
- Generating a result web page to be shown to the client: confirmations that a modification has taken place, for update actions, or presenting result data for query actions.

It is recommended to separate this tasks in different components, in particular those that generate resulting web pages and those that communicate with a database, to avoid writing any HTML or database code in controller components such as servlets.

Example of a servlet from dating application

```
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
import javax.servlet.*;
public class createMemberServlet extends HttpServlet {
      private Dbi dbi; // database interface

      public createMemberServlet() {}

      public void init(ServletConfig cfg) throws ServletException {
            super.init(cfg);
            dbi = new Dbi();
      }
```

```
public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
	res.setContentType("text/html");
	PrintWriter pw = res.getWriter();
	ErrorPage errorPage = new ErrorPage();
	String username = req.getParameter("username"); String age =
	req.getParameter("age");

	int iage = 0;
	try { iage = Integer.parseInt(age); }
	catch (Exception e) { errorPage.addMessage(age + " is not an integer");
	}
	String location = req.getParameter("location");
	String email = req.getParameter("email");
	String mobile = req.getParameter("mobile");
	String salary = req.getParameter("salary");

	int isalary = 0;

	try {
		isalary = Integer.parseInt(salary);

	} catch (Exception e) {
		errorPage.addMessage(salary + " is not an integer"); }
		if (errorPage.hasError()) {
			pw.println(errorPage); }
		else
			try { dbi.createMember(username, iage, location, email,
				mobile, isalary);
				CommandPage cp = new CommandPage(); pw.println(cp);

			} catch (Exception e) { e.printStackTrace();
				errorPage.addMessage("Database error");
				pw.println(errorPage); } pw.close();

		public void doPost(HttpServletRequest req
			HttpServletResponse res) throws ServletException,
		IOException {
			doGet(req,res);
}

public void destroy() {
	dbi.logoff();
}
```

}

- ***doPost*** and ***doGet*** methods execute whenever POST or GET requests are received by servlet:
    - POST requests used for data updates, e.g., registration, and GET for data retrieval.
    - GET can be used for updates if data sent isn't confidential, and is of small size (e.g., under 256K bytes).
- GET appends form data to end of URL request is sent to, and GET requests can be cached by browser.
- POST can't be cached (normally), but data is transmitted in packet separate from server URL, so more secure.
- In above servlet, ***doGet*** extracts data entered in registration form by using

```
String par = req.getParameter("parname");
```

  for each parameter and its name (name of corresponding input field in HTML form).
- Data is transmitted across internet as strings, so servlet converts data such as integers or doubles to intended type.
- Checks on web page constraints can be performed.
- Here, if data is correct, it is written to database via ***dbi.createMember***, and a page containing options for further operations is returned.
  Otherwise, an error page is returned.
- Auxiliary components ErrorPage and CommandPage generate web pages to be shown in response:
    - either page listing data entry errors in input,
    - or page listing command options which user can perform.

Five-tier architecture of web applications

Architecture diagrams for a web application typically include up to five main subsystems or 'tiers':
Client tier: consists of web pages

- either hard coded ('static') HTML text files, downloaded from server to the client,
- or generated ('dynamic') pages, produced as result of an HTTP request to a server-side component.

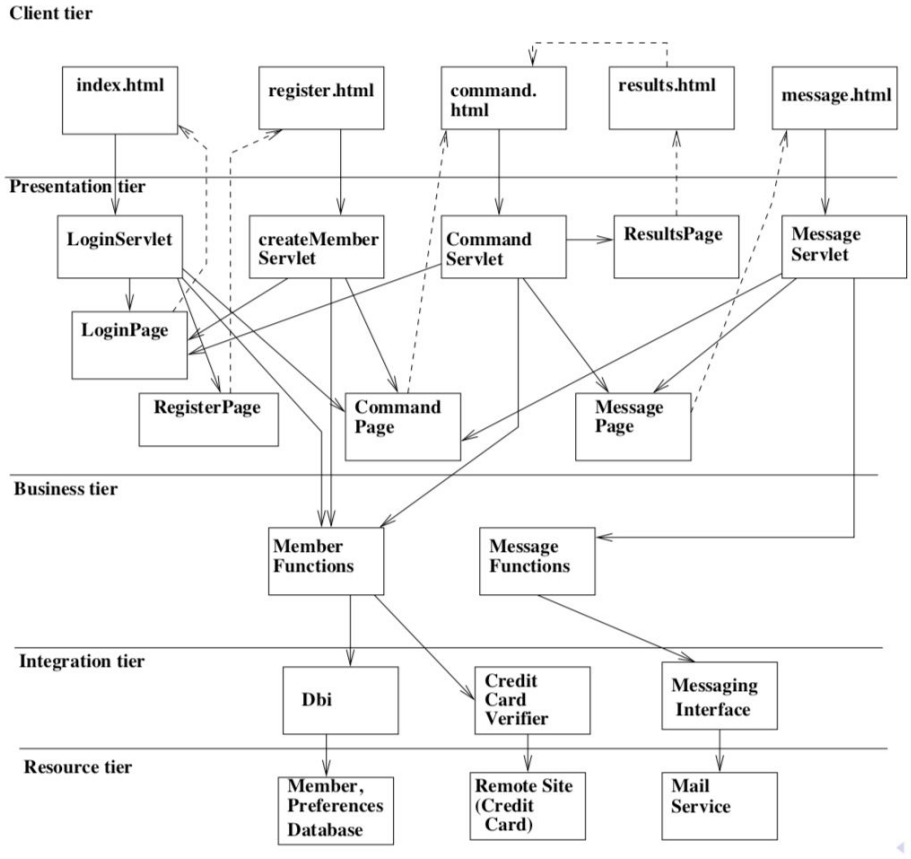
Presentation tier: consisting of controller and view components such as servlets and JSP's.

- 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 interaction state machine.

Business tier: consisting of

- 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: containing database interfaces, interfaces to external web services, etc.
Resource tier: the actual databases, web services and other resources used by the system.



- A solid line arrow from component C to component D means that C invokes an operation/service of D.
- Dashed arrows representing that a server-side component generates a particular web page, or representing HTML links between web pages.
- Solid and dashed arrows from web page to web page should give the same interaction sequences as the interaction sequence state machine.

Different architectural styles for presentation tier

There are three different ways to implement 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 skills or JSP compiler.
- **Pure JSP**: JSPs respond to requests, call DBI/business tier and generate response pages.
- **Servlet/JSP:** like pure servlet approach, but using JSPs to construct response pages, on redirect from servlets.

Servlet-based web architecture

Have shown above the *register.html* input form and its servlet *createMemberServlet* for the dating agency application.

The auxiliary *CommandPage* and *ErrorPage* view components are as follows:

```
public class ErrorPage extends BasePage {
    private int errors = 0;
    HtmlItem para = new HtmlItem("p");

    public void addMessage(String t) {
        body.add(new HtmlText(t,"strong"));
        body.add(para);

        errors++;
    }
    public boolean hasError() {
        return errors > 0;
    }
}
```

Database interface uses SQL statements to read and write data to the database:

```
import java.sql.*;

public class Dbi {

      private Connection connection;

      private static String defaultDriver = "";

      private static String defaultDb = "";
      private PreparedStatement createMemberStatement;

      private PreparedStatement editMemberStatement;

      public Dbi() {

            this(defaultDriver,defaultDb);

      }

      public Dbi(String driver, String db) {

            try {

                  Class.forName(driver);
                  connection = DriverManager.getConnection(db);

                  createMemberStatement = connection.prepareStatement("INSERT INTO
            Member" + "  (username,age,location,email,mobile,salary) VALUES
            (?,?,?,?,?,?)");
```

```
editMemberStatement = connection.prepareStatement("UPDATE Member SET " +
"  username = ?, age = ?, location = ?, email = ?," + "  mobile = ?, salary =
? WHERE memberId = ?");

} catch (Exception e) { } }

public synchronized void createMember(String username,int age, String
location, String email,String mobile,int salary) {

      try {

            createMemberStatement.setString(1, username);

            createMemberStatement.setInt(2, age);

            createMemberStatement.setString(3, location);
            createMemberStatement.setString(4, email);
            createMemberStatement.setString(5, mobile);
            createMemberStatement.setInt(6, salary);
            createMemberStatement.executeUpdate();

            connection.commit();

      } catch (Exception e) {

            e.printStackTrace();

      }


}

public synchronized void editMember(String username,int age, String
location,String email,String mobile,
int salary,String memberId) {

      try {

            editMemberStatement.setString(1, username);
            editMemberStatement.setInt(2, age);
            editMemberStatement.setString(3, location);
            editMemberStatement.setString(4, email);
            editMemberStatement.setString(5, mobile);
            editMemberStatement.setInt(6, salary);
            editMemberStatement.setString(7, memberId);
            editMemberStatement.executeUpdate();
            connection.commit();
```

```
} catch (Exception e) { e.printStackTrace(); }

public synchronized void logoff() {

      try { connection.close(); }
            catch (Exception e) { e.printStackTrace(); } }
}
```

JSP-based web architecture

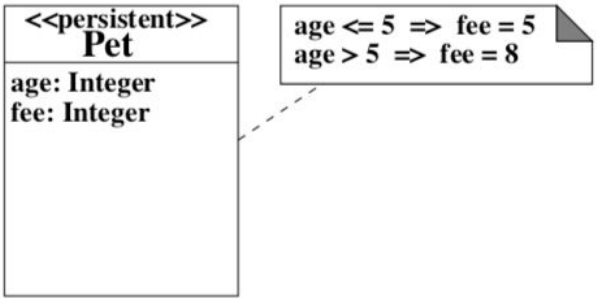
- Instead of using helper classes such as *CommandPage* or *ErrorPage* to generate result web pages, can write JSP files, which describe result pages as a mixture of fixed HTML text and dynamically generated text, produced by Java statements embedded in the JSP.

- Can also separate out database update code into **session beans** or **entity beans** invoked from JSPs, manipulating/representing data (e.g., instances of entities) being processed.

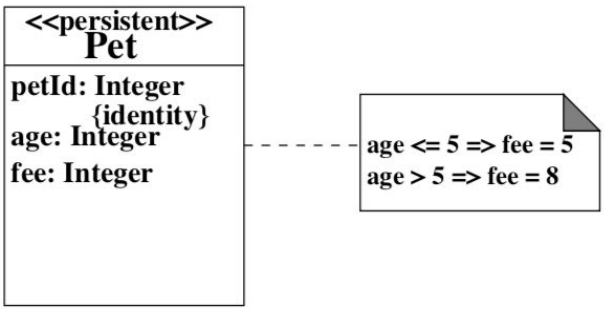
- The following example, of part of pet insurance system, illustrates web architecture based on JSPs instead of servlets.

Pet Insurance Example

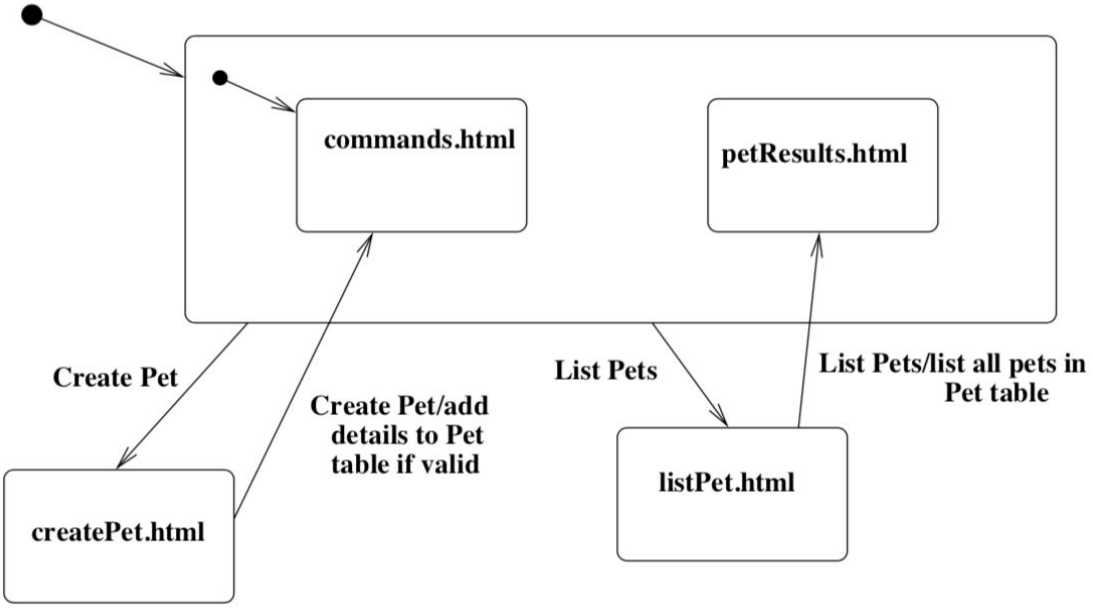
- System records information on pets insured by a pet insurance company, and maintains business rule that if pet is not more than 5 years old, its monthly insurance fee is £5, otherwise its fee is £8.
- The use cases of this system are to create a new pet and to list all pets.
- The design class diagram adds an integer *petId* : *Integer* identity attribute to the *Pet* entity.

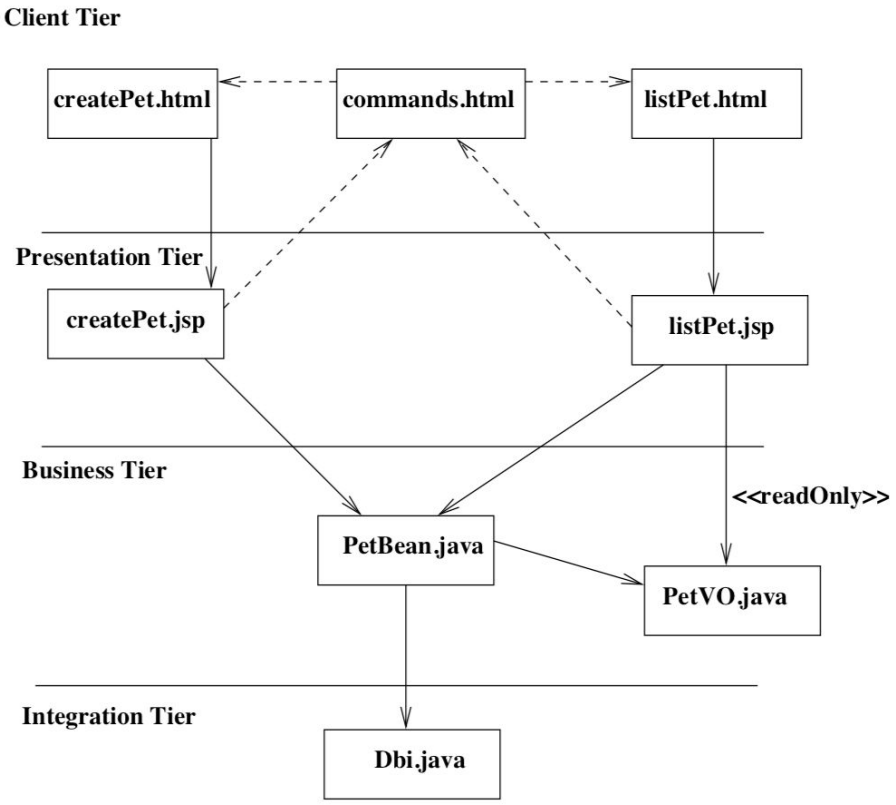


- The HTML files *createPet.html* and *listPet.html* define input forms for these operations, and invoke corresponding JSPs *createPet.jsp* and *listPet.jsp*.


<<persistent>>
Pet
petId: Integer
{identity}
age: Integer
fee: Integer
age <= 5 => fee = 5
age > 5 => fee = 8


<<form>>
CreatePet
petId: Integer
age: Integer
fee: Integer
createPet()
<<create>>
<<form>>
ListPet
listPets()
<<list>>


commands.html
petResults.html
Create Pet
Create Pet/add details to Pet table if valid
List Pets
List Pets/list all pets in Pet table
createPet.html
listPet.html



- The file commands.html is included in each JSP to provide navigation to the command options:

```
<p><a href="createPet.html">createPet</a></p>
<p><a href="listPet.html">listPet</a></p>
```

- The following JSP, createPet.jsp, defines an instance pet of the PetBean class (line 1), then copies the form data to this bean (lines 2, 3, 4).
- The part of the JSP enclosed within <html> ... </html> defines the response web page returned to the client.
    - In this case the page is simply a message that the creation has occurred or failed.
    - For ***listPet***, below, the response page consists of a table with rows containing the petId, age and fee of each pet in the database.
- Using iscreatePeterror method of ***PetBean***, the JSP checks if form data was correct (of correct type and satisfying the invariants) and displays any errors if any exist. If there are no errors it updates database via the bean:

```
<jsp:useBean id="pet" scope="session" class="beans.PetBean"/> <jsp:setProperty
name="pet" property="petId" param="petId"/> <jsp:setProperty name="pet" property="age"
param="age"/> <jsp:setProperty name="pet" property="fee" param="fee"/>

<html>
<head><title>createPet</title></head>
<body>
<h1>createPet</h1>
```

```
<% if (pet.iscreatePeterror())
{ %> <h2>Error in data: <%= pet.errors() %></h2> <h2>Press Back to re-enter</h2> <% }
else { pet.createPet(); %>
<h2>createPet performed</h2>
<% } %>

<hr>

<%@ include file="commands.html" %> </body>
</html>
```

- Code between < % and % > brackets is executable Java code.
- Code outside these brackets is HTML, plus special purpose JSP tags.
- The notation < % = exp% > evaluates exp and substitutes its value into the result web page at this point.
- JSPs are compiled into servlets by a JSP compiler, this is normally carried out automatically by web server, eg., Resin or Tomcat. Compilation causes delay when system is initialised. May also be necessary to view generated servlet in order to debug the JSP file, if errors occur in processing

listPet.jsp obtains the current list of pet objects from the bean and formats them into an HTML table:

```
<%@ page import = "java.util.*" %> <%@ page import = "beans.*" %> <jsp:useBean id="pet" scope="session"

class="beans.PetBean"/>

<html>
<head><title>listPet results</title></head>
<body>
<h1>listPet results</h1>
<% Iterator pets = pet.listPet(); %>
<table border="1">
<tr><th>petId</th> <th>age</th> <th>fee</th></tr>
<% while (pets.hasNext())
{ PetVO petVO = (PetVO) pets.next(); %>
<tr><td><%= petVO.getpetId() %></td> <td><%= petVO.getage() %></td> <td><%= petVO.getfee() %></td></tr>
<% } %>
</table>

<hr>

<%@ include file="commands.html" %> </body>
</html>
```

The ***PetBean*** session bean performs type and invariant checking of attributes, and interfaces to the ***Dbi*** to update and query the database table for ***Pet*****:**

```
package beans;

import java.util.*;
```

```
import java.sql.*;

public class PetBean {
	Dbi dbi = new Dbi();

	private String petId = "";
	private int ipetId = 0;
	private String age = "";
	private int iage = 0;
	private String fee = "";
	private int ifee = 0;

	private Vector errors = new Vector();
	public PetBean() {}

	public void setpetId(String petIdx) { petId = petIdx; }

	public void setage(String agex) { age = agex; }

	public void setfee(String feex) { fee = feex; }

	public void resetData() { petId = ""; age = ""; fee = ""; }

	public boolean iscreatePeterror() {
		errors.clear();

		try {
			ipetId = Integer.parseInt(petId);
		} catch (Exception e) {
			errors.add(petId + " is not an integer");
		}
		try {
			iage = Integer.parseInt(age);
		} catch (Exception e) {
			errors.add(age + " is not an integer");
		}
		try {
			ifee = Integer.parseInt(fee);
		} catch (Exception e) {
			errors.add(fee + " is not an integer");
		}

		if (!(iage <= 5) || (ifee == 5)) { }
		else {
			errors.add("Constraint: !(iage <= 5) || (ifee == 5) failed");
		}
		if (!(iage > 5) || (ifee == 8)) { }
```

```
            else {
                errors.add("Constraint: !(iage > 5) || (ifee == 8) failed");
            }

            return errors.size() > 0; }

    public boolean islistPeterror() {
            errors.clear();
            return errors.size() > 0;
    }

    public String errors() { return errors.toString(); }

    public void createPet() {
            dbi.createPet(ipetId, iage, ifee);
            resetData();
    }

    public Iterator listPet() {
            ResultSet rs = dbi.listPet();

            List rs_list = new ArrayList();

            try{
                while (rs.next()) {
                    rs_list.add(new PetVO(rs.getInt("petId"), rs.getInt("age"),
                    rs.getInt("fee")));

                }
            } catch (Exception e) { }

            resetData();
            return rs_list.iterator();

} }
```

Here, a constraint $A \Rightarrow B$ is evaluated as !(A) || B in Java.
In this case bean only checks that invariants are true before permitting an update, however a more proactive approach would enforce a change in *fee* if a change in *age* occurs.
Business tier is correct place for such business rule code, and components such as J2EE entity beans may be necessary to ensure that invariant-maintenance code is carried out in transactional manner.

PetVO is a 'Value Object' for Pet entity, used to transfer data between presentation and business tier (to avoid exposing classes such as *ResultSet* to presentation tier):

```
package beans;
```

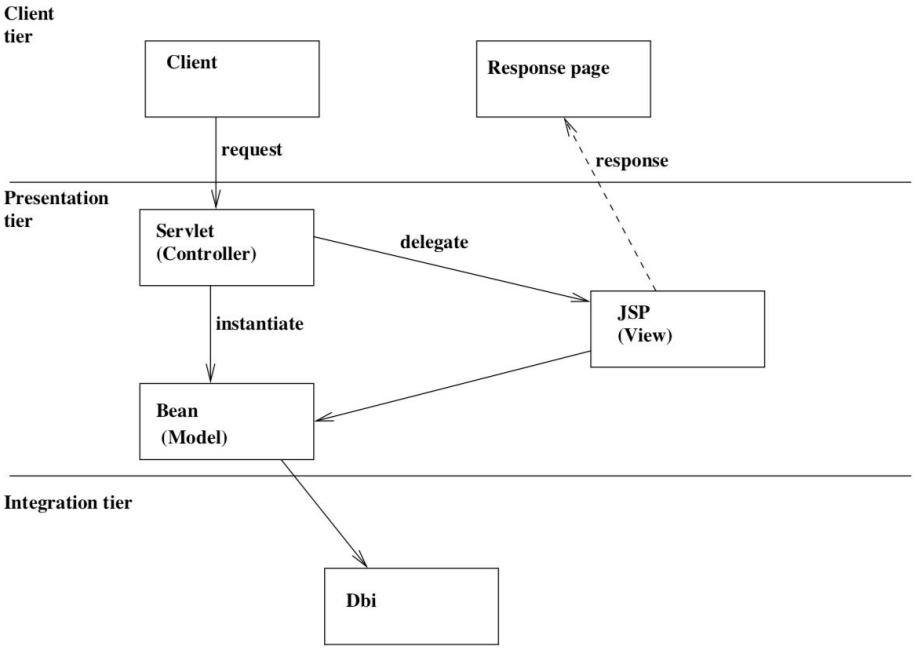
```
public class PetVO {
    private int petId;

    private int age;
    private int fee;

    public PetVO(int petIdx,int agex,int feex) {
        petId = petIdx;
        age = agex;
        fee = feex;
     }

     public int getpetId() {
         return petId;
     }

     public int getage(){
         return age;
     }

    public int getfee() {
         return fee;
    }
}
```

Combined servlet/JSP approach

Pure JSP approach can lead to complicated programming within JSP, as scriptlets. An alternative is a hybrid approach where servlets (**controllers**) initially handle requests, interact with database via business tier beans (**models**), and also create beans for use by JSPs (**views**).

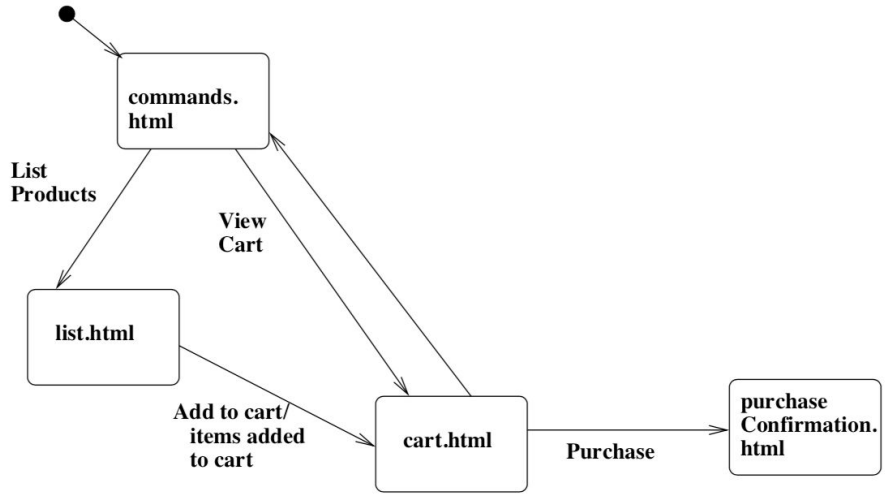
- Servlet controllers decide which JSP to forward a request to, and generally control what sequence of interaction is to be followed.
- The JSP components are only concerned with presenting data as web pages.

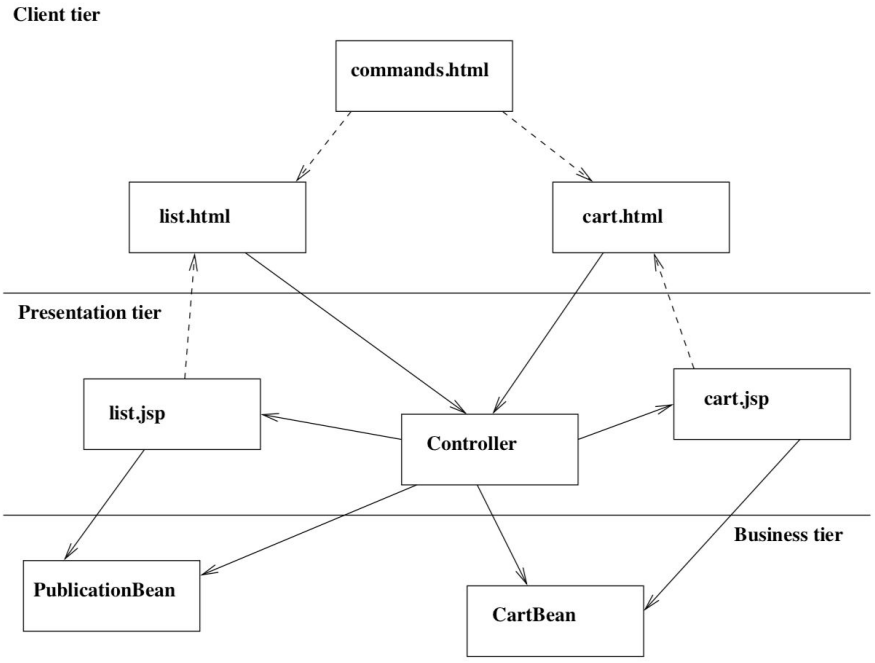
Known as **MVC (Model-View-Controller) architecture**.
More flexible and extensible than pure JSP or servlet approaches, so is more appropriate for larger and more complex systems.



This system maintains shopping cart, using servlet front controller, constructing the cart in a bean, and JSPs reading the cart data to display it.

- list.html shows current list of products, a checkbox for each product allows multiple selections. It invokes Controller.
- cart.html shows contents and total cost of shopping cart, and gives an option to purchase. It invokes Controller.
- Controller servlet which adds items to cart (CartBean), and carries out purchases.
- list.jsp generates list.html using the database table of products (PublicationBean).
- cart.jsp generates cart.html using CartBean.





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.

Java Apache Struts framework uses MVC organisation for presentation tier, as does Ruby on Rails.

Summary

- Web application development involves
  - development of the server-side functional code;
  - development of the UI as static or dynamic web pages; and
  - design of the visual appearance and information content of the web pages.
- Properties which are particularly important for web applications are portability, usability and accessibility.
- A systematic MDD/MDA process for web applications can be defined.
- Web applications typically consist of five levels or tiers:
  - client;
  - presentation;
  - business;
  - integration and
  - resource.
- Java components for the presentation tier include servlets, JSPs, and auxiliary Java classes.
- Using Java technologies, a choice of three different architectural styles is possible for the presentation tier of a web application:
  - pure servlet;
  - pure JSP; and
  - model-view-controller (MVC), of which the MVC approach is the most flexible and appropriate for complex systems.

Lecture 7 Enterprise information System

EIS Concepts

An EIS is a software system that holds and manipulates businesses' core data.

This systems are generally large and complex, and usually involve distributed processing and distributed data. Additionally, an EIS will often be used as a resource by different applications (web or non-web).

e.g. 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 customer data.

EIS Specification and design techniques

Enterprise systems can be specified in terms of the platform-independent 'business rules' which defines the proportions of their data and operations

e.g. (bank account system)

$$balance \geq -limit$$
$$balance < 0 \implies charge = interestRate * (-balance)$$
$$balance \geq 0 \implies charge = 0$$

EIS Architecture and Components

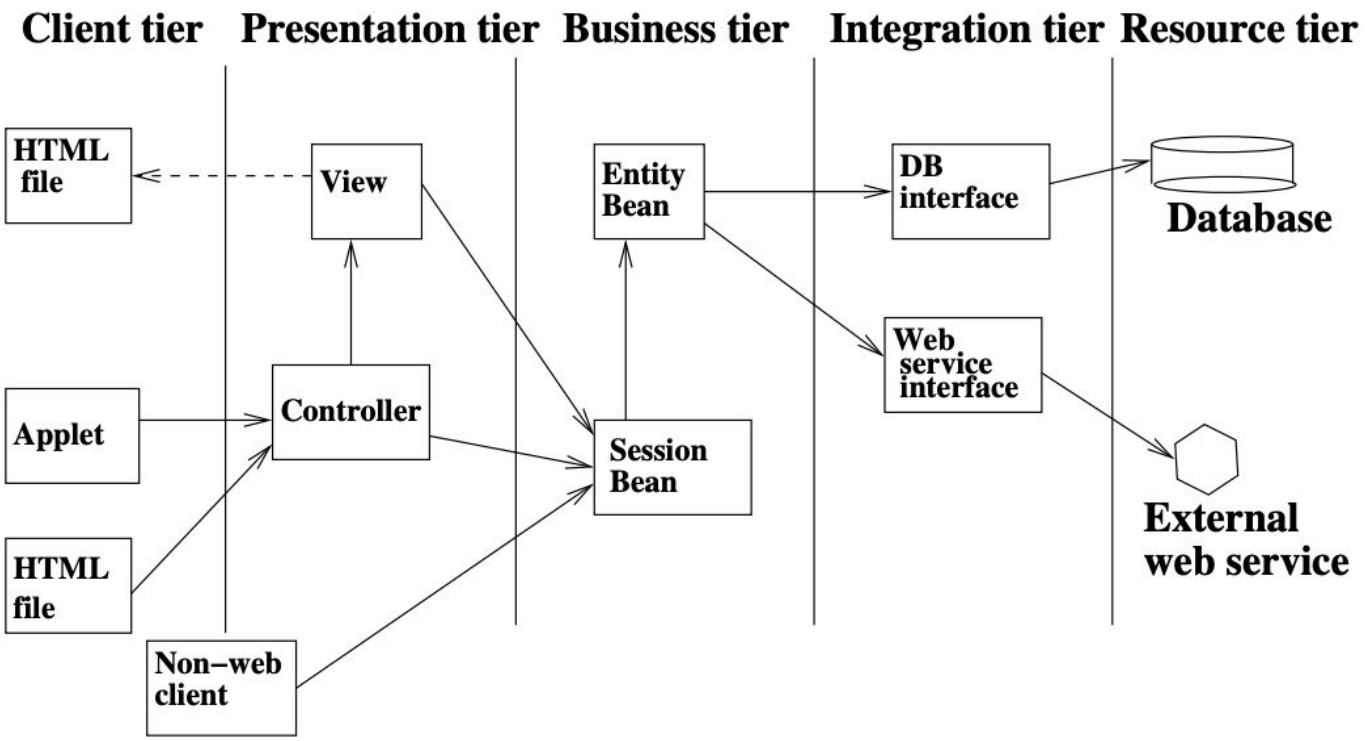
A 5 tier architecture is typically used to describe EIS Applications

- **Client tier**:
  - Responsible for displaying information and receiving information
    - It might be a **thin client** with minimal processing other than visual interface
    - or a **fat client** doing more substantial computation
  - The trend is towards thin clients, using web browsers and known as **web clients**
  - Typical components in this tier are HTML or applets
- **Presentation tier:** (Very close to the Client Tier)
  - Has responsibility of managing presentation of information and controlling the sequence of interactions to be followed + user requests to the business tier
  - Typical components in this tier are controller and view components (servlets and JSPs)
- **Business tier:**
  - Contains components implementing the business rules and data of the application such as **session beans** and **entity beans**.

- **Integration tier:**
  - Mediates between business tier and resource tier
  - Manages data retrieval, using interfaces such as JDBC (Java Database Connectivity)
  - It insulates business and higher tiers from direct knowledge of how data is stored and retrieved (privacy and security)

- **Resource tier**
  - Contains persistent data storage, such as databases, and external resources such as credit card authorisation services or business-to-business services.

Five-tier EIS architecture



Business tier design

- Should be the most stable part of EIS, based on PIM specification of business data and functionality
- Integration tier insulates business tier from changes in data storage technology or resource details while presentation tier insulates it from changes in UI technology or client applications
- Separating core business functionality into separate tiers enables client applications and UIs to access this functionality on regardless of the medium used.
- Introduction to business tier enables client applications/UIs to be remote from business functionality, and this functionality to be remote from data storage

- Components of business tier define business data and logic (In a java-based system this could be through ordinary Java classes or specialised classes defined in Java Enterprise Edition known as **Enterprise Java Beans (EJBs)**.

Forms of Business Tier component:

Session Bean: Business component

- Dedicated to a single client
- that lives only for the duration of the client's session
- that is not persistent
- that can be used to model stateful or stateless interactions between the client and business tier components

A session bean encapsulates a group of operations, usually corresponding to use cases of the system, which operate on one or more of the entities of the system

Stateless session beans provide a service by a single method call like:

- utility funcitons or logging services
- sending an email confirming that a client request has been received

Since stateless session beans don't hold client-specific state they can be shared between clients and reused from one client to another.

Stateful session beans typically provide a service specific to a single client (they can't be shared) and involve several related operations (e.g. a shopping cart, or messaging operations of dating agency).

Stateful session beans can be used for things like

- encapsulating checks and business rules on data which require access to persistent data or external services (e.g. credit checks on a new customer application to a bank)
- providing an object-oriented interface to one or more relational database

Entity Bean: a business component which

- provides an object view of persistent data
- is multiuser and long-lived

Entity beans act as an object-oriented facade for data of a system, which may actually be stored as relational tables or XML datasets.

Other parts of the system (like session beans) can use entity beans as if they stored the data in a purely object-oriented manner.

Normally each entity from the PIM class diagram will be managed by some entity bean.

Persistence management

Some EIS platforms like J2EE/Java EE, provide automated synchronisation of the entity bean and the actual data stored in the resource tier, this is known as **container-managed persistence (CMP)**

Alternatively, this can be achieved by the programmer of the bean explicitly providing suitable logic, such as saving data to a database using JDBC, this is known as **bean-managed persistence (BMP)**

CMP provides potentially greater portability, avoiding use of platform-specific code within bean classes. However, BMP allows for a greater level of customisation of the database interface code.

Development process of EIS applications

We can extend the web application development process defined in unit 3 to EIS applications by including the following steps:

- **Business Tier:**
  - Group classes into modules which are suitable for implementation as entity beans, with strong connections between classes within each module and weaker ones between classes in different modules
  - Group use cases into session beans, according to entities which they operate on and according to which use cases are frequently used together (same session by same actor)
  - Grouping should minimise dependencies within business tier, and dependencies of the presentation tier on the 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

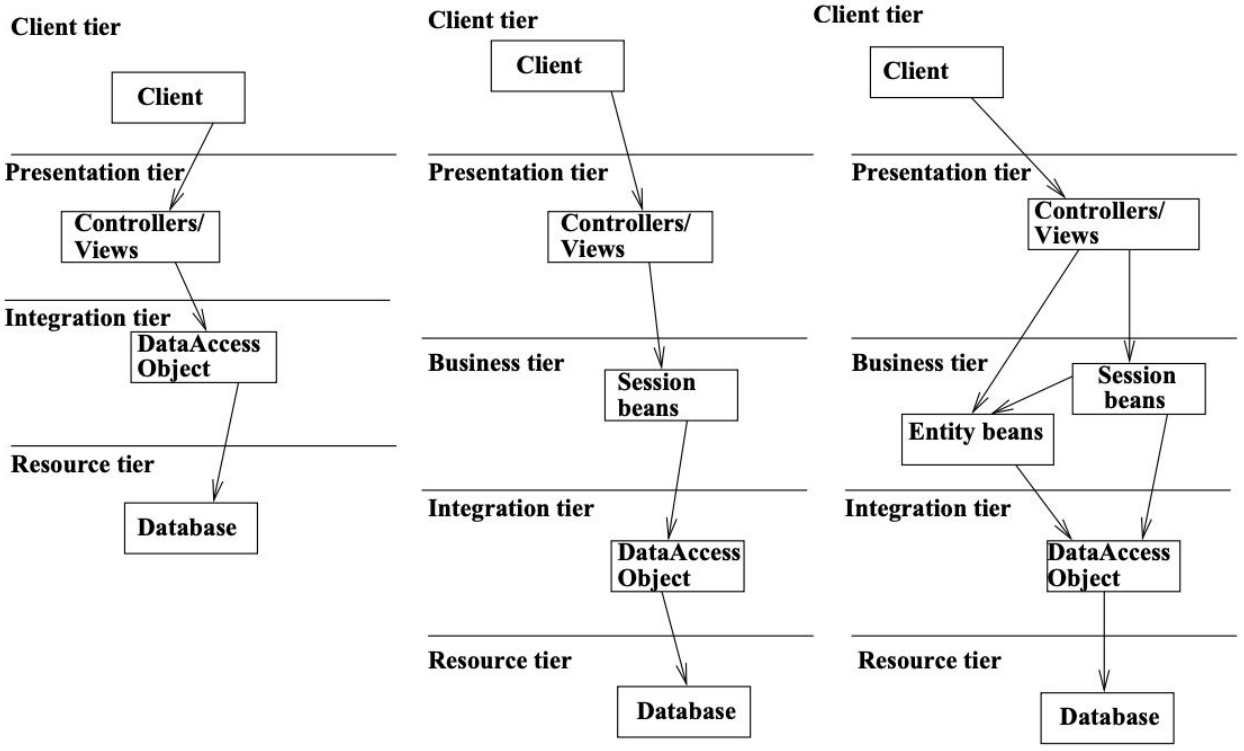
Introducing EIS tiers

The full EIS architecture is not necessary for all systems

- Session beans should be introduced:
  - When the mix of business logic and view/control logic in the presentation tier becomes too complex
  - 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 example below show different EIS tiers being used depending of the service or client



Design choices in the business tier

The following steps can be taken to develop an enterprise system using an MDA approach to model-driven development:

- Define PIM data model and use cases
- Derive PSM data model by applying model transformations to PIM
- Derive architecture and implementation of system using PIM constraints to derive operation and transaction code

These steps ensure that:

- specification properties remain true after model transformations,
- and code generation step produces code designed to maintain these properties at all times of execution of the system

Constraints which are class invariants define data validity checks, carried out by an entity bean derived from the class and can also be used to define transactions which modify dependent attributes when an attribute changes value.

Constraints linking data of two different classes can be used to define transactions involving operations of entity beans of both classes

- Any change to data of one entity may require changes in the other to maintain constraint
- Updates should take place within uninterruptible transactions

Constraints also influence architectural decisions, e.g. if a constraint links two classes we would implement use cases for both classes to be in the same session bean

Lecture 7 goes on to provide 4 examples of EIS design

- Example of a stateless session bean to calculate the maximum mortgage loan for someone based on their monthly income and the duration of the loan
- Pet insurance system
- BMP entity bean example (estate agent system)
- CMP entity bean and stateful session bean (bank account)

EIS Design Issues

Data security

Passwords should only be stored in encrypted format so they can be compared with encrypted user input but never exposed (Organisations which send your password to you on request must be storing the data in unencrypted form)

HTTPS should be used systematically in all security critical parts of a website,

Remove web-specific coding from business tier

Business tier code should not refer to HTTP request structures like the one below

```
public class House
{ String address;
  String style;

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

Using HttpServletRequest as an input parameter type prevents non-web clients from using the business object. Instead data should be 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

Another 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 design for specific tasks and give basis of this separation:

- Controller and view components for presentation processing
- Session beans for business processing
- Entity beans for complex business data

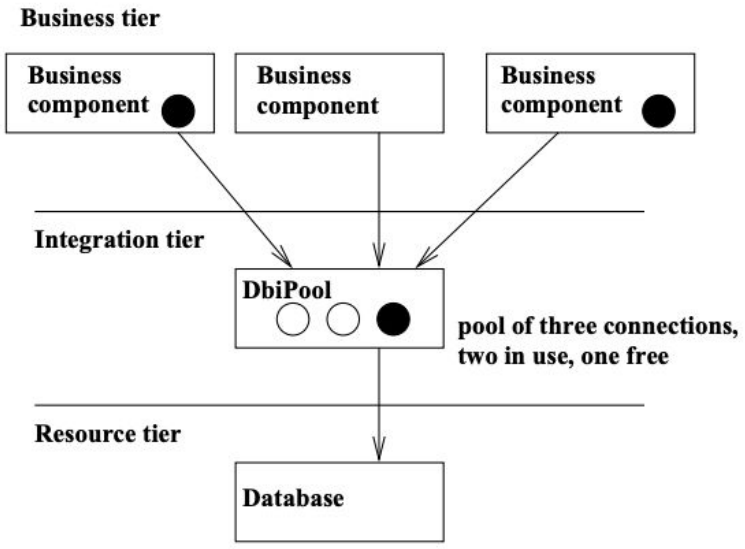
Database connection pooling

A particular technique that increases efficiency of a database interface since it minimises the computing expense of creating a connection to a database.

This can be achieved by creating a connection management component, ConnectionPool, which holds a set of pre-initialised connections.

Components which require a connection must ask the pool for a free connection, and when they are done with it, they must return it to the free set.

JDBC provides pooling in javax.sql.DataSource



Unit 8 Enterprise Information Systems Pattern, Web Services

EIS Patterns

One solution to complexity of EIS design is to provide patterns or standard solutions for EIS design problems.

Design patterns define microarchitecture within an EIS, to implement a particular required property/functionality of system or to rationalise system structure.

Presentation tier patterns include:

- **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.
- **Composite View:** uses objects to compose a view out of parts (subviews). Service to Worker: combines front controller and view helper to construct complex presentation content in response to a request.

Business tier patterns include:

- **Value Object:** an object which contains attribute values of a business entity (entity bean), this object can be passed to presentation tier as a single item, so avoiding cost of multiple getAttribute calls on the entity bean.
- **Session Facade:** use a session bean as facade to hide complex interactions between business objects in one workflow/use case.
- **Composite Entity:** use 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:** provide efficient interface to examine a list of value objects (e.g., result of a database search).

Integration tier patterns include:

- **Data Access Object:** provides abstraction of persistent data source access.

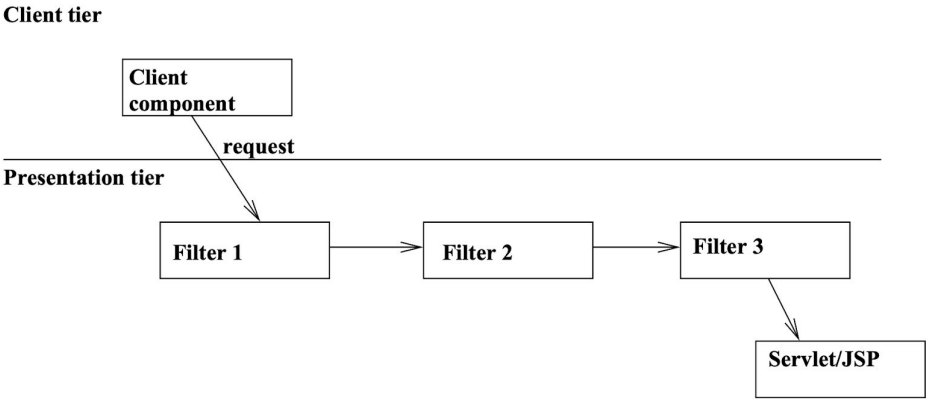
Intercepting Filter

Intercepting Filter: provides 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.

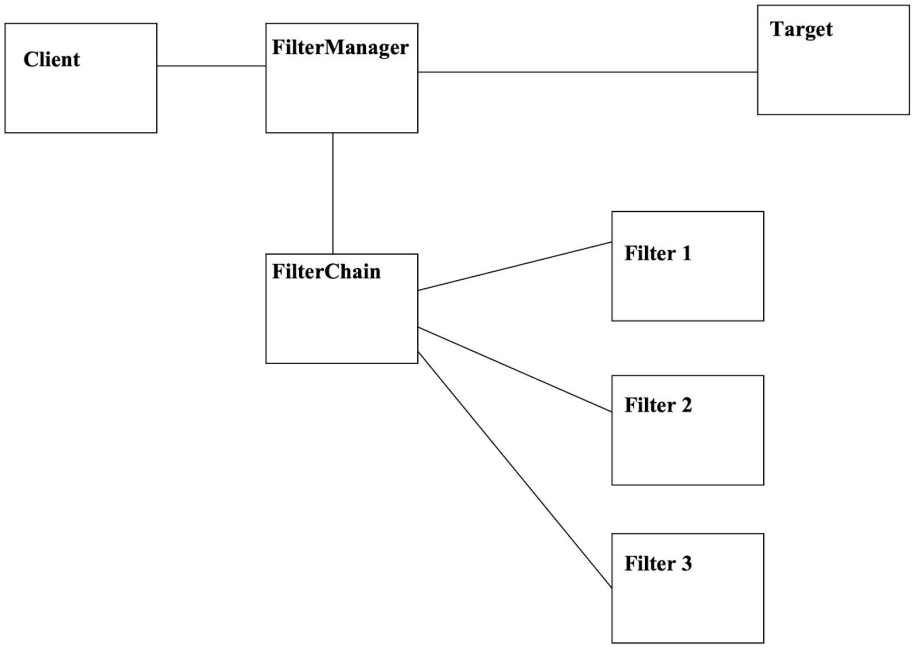
- Is the client's IP address from a trusted network?
- Does the client have a valid session?
- Is the client's browser supported by the application?

It would be possible to code these as nested if tests, but is more flexible to use separate objects in a chain to carry out successive tests (cf., the Chain of Responsibility pattern)



Elements of an Intercepting Filter:

- **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.



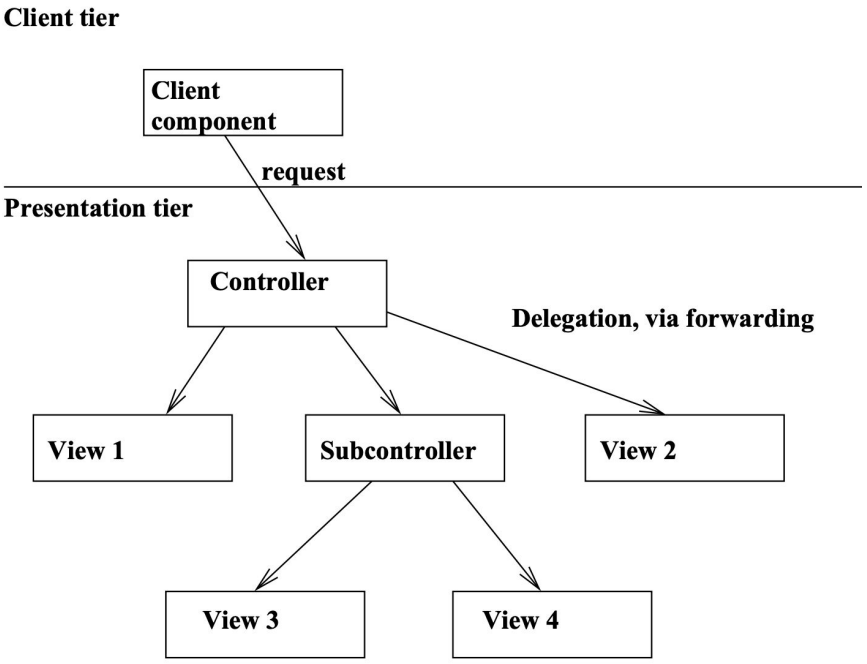
```
public interface Processor
{ public void process(ServletRequest req,
                      ServletResponse res)
  throws IOException, ServletException;
}

public class Filter1 implements Processor
{ private Processor target;

  public Filter1(Processor t) { target = t; }

  public void process(ServletRequest req,
                      ServletResponse res)
  throws IOException, ServletException
  { // do filter 1 processing, then forward request
    ....
    target.process(req,res);
  }
}

public class Filter2 implements Processor
{ private Processor target;

  public Filter2(Processor t) { target = t; }

  public void process(ServletRequest req,
                      ServletResponse res)
  throws IOException, ServletException
  { // do filter 2 processing, then forward request
    ....
    target.process(req,res);
  }
}

public class Target implements Processor
{ public void process(ServletRequest req,
                      ServletResponse res)
  throws IOException, ServletException
  { // do main resource processing  }
}

public class FilterManager
{ Processor head;

  public void setUpChain(Target resource)
  { Filter2 f2 = new Filter2(resource);
    head = new Filter1(f2); }

  public void processRequest(ServletRequest req,
                             ServletResponse res)
  { head.process(req,res); }
}
```

This pattern is provided using standard interfaces and components in Java EE

Front Controller

Front controller: provides a central entry point for an application that controls and manages web request handling. The controller component can control navigation and dispatching.

The pattern factors out similar request processing code that is duplicated in many views (e.g., the same authentication checks in several JSP's). It makes it easier to impose consistent security, data, etc, checks on requests.



Elements:

- **Controller**: initial point for handling all requests to the system. It forwards requests to sub controllers 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:** components which process specific requests, forwarded to them by the controller.

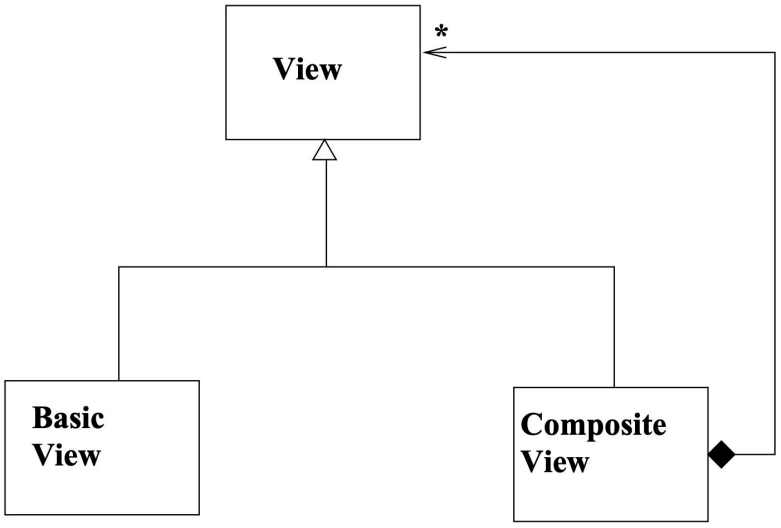
```
public class PropSysController extends HttpServlet
{ public void init(ServletConfig cf)
  throws ServletException
  { super.init(cf); }

  public void doGet(HttpServletRequest rq,
                    HttpServletResponse rs)
  throws ServletException, IOException
  { // carry out any common security/authentication checks

    String regC = rq.getParameter("Register");
    if (regC != null)
    { // pass request to register servlet
      dispatch(rq,rs,"RegisterUserServlet");
      return;
    }
    String editC = rq.getParameter("Edit");
    if (editC != null)
    { // pass request to edit servlet
      dispatch(rq,rs,"EditUserServlet");
      return;
    }
    ...
  }
}
```

This pattern helps to improve security and flexibility, by disallowing direct access to specific components of the system: all requests must pass through the controller.

Composite View

Composite View: manages views which are composed from multiple subviews.

Hard-coding page layout and content provides poor flexibility. The pattern allows views to be flexibly composed as structures of objects



Elements:

- **View:** a general view, either atomic or composite.
- **View Manager:** 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 tag in JSP,

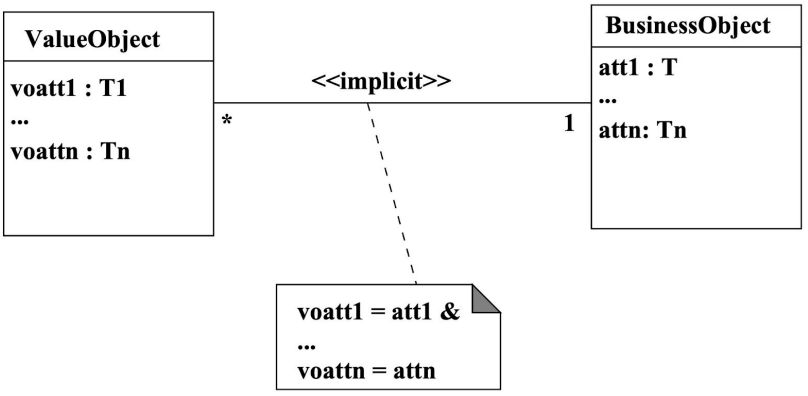
```
<jsp:include page = "subview.jsp">
```

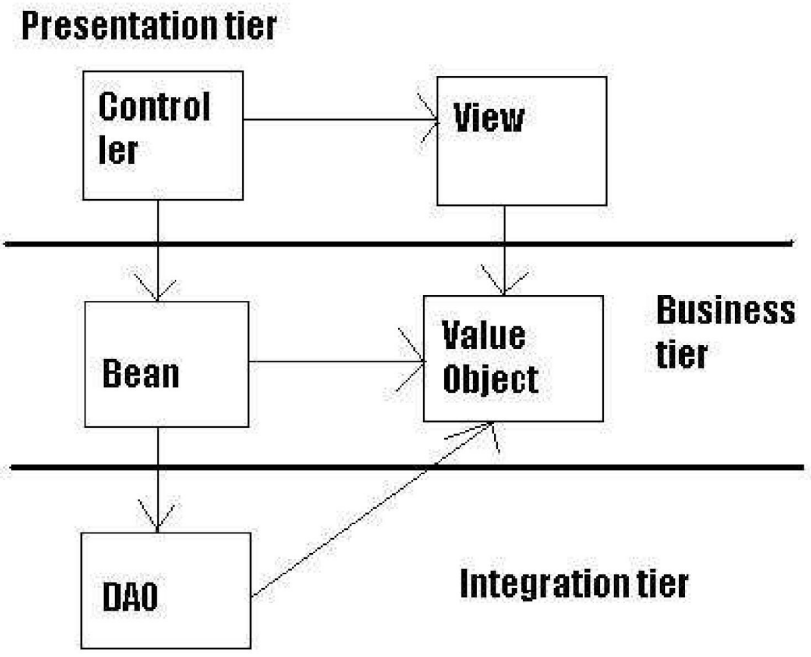
used to include subviews within a composite JSP page. Other approaches include custom JSP tags and XSLT (if data is stored as XML).

Value Object

Value Object: improves the efficiency of access to 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 getatt() calls, since these calls are potentially remote.
- The pattern reduces data transfer cost by transferring data as packets of values of several attributes.
- Can transfer data between presentation and business tiers, and between integration and business tiers.





The elements of the pattern are:

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

This pattern satisfies an invariant *voatt* = *att* for each attribute *att* of the business object and corresponding attribute *voatt* of the value object.

```
public class BusinessObject implements EntityBean
{ private T1 att1;
  ...
  private Tn attn;

  ...

  public ValueObject getData()
  { return new ValueObject(att1,...,attn); }
}

public class ValueObject implements Serializable
{ public T1 voatt1;
  ...
  public Tn voattn;

  public ValueObject(T1 v1, ..., Tn vn)
  { voatt1 = v1;
    ...
    voattn = vn;
  }
}
```

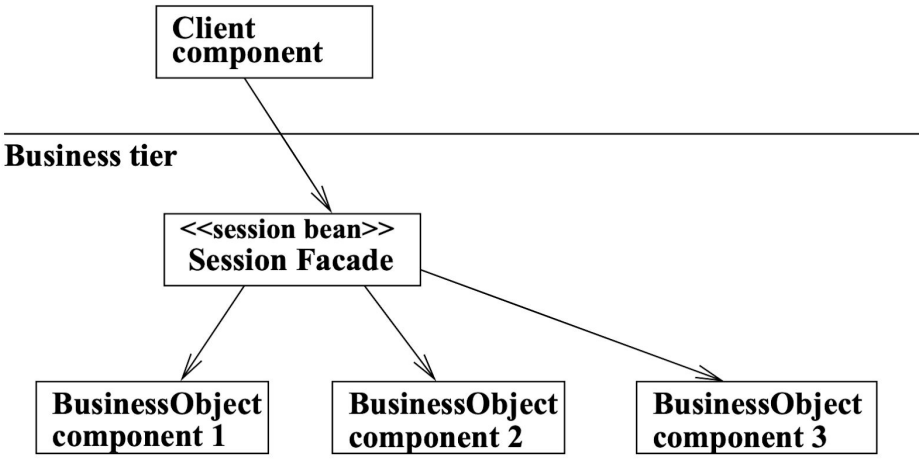
- The value object can also be used to update business objects via a *setData(ValueObject vo)* method.
- Example: getDetails method of *CustomerControllerBean*.

Session Facade

Session Facade: encapsulates the details of complex interactions between business objects.

A session facade for a group of business objects manages these objects and provides a simplified and fewer set of operations to clients.

- Interaction between a client and multiple business objects may become very complex, with code for many use cases written in the same class.
- Instead this pattern groups related use cases together in session facades.



The elements of the pattern are:

- **Client:** client of session facade, which needs access to the business service.
- **SessionFacade:** implemented as a session bean. It manages business objects and provides a simple interface for clients.
- **BusinessObject:** can be session beans or entity beans or data.

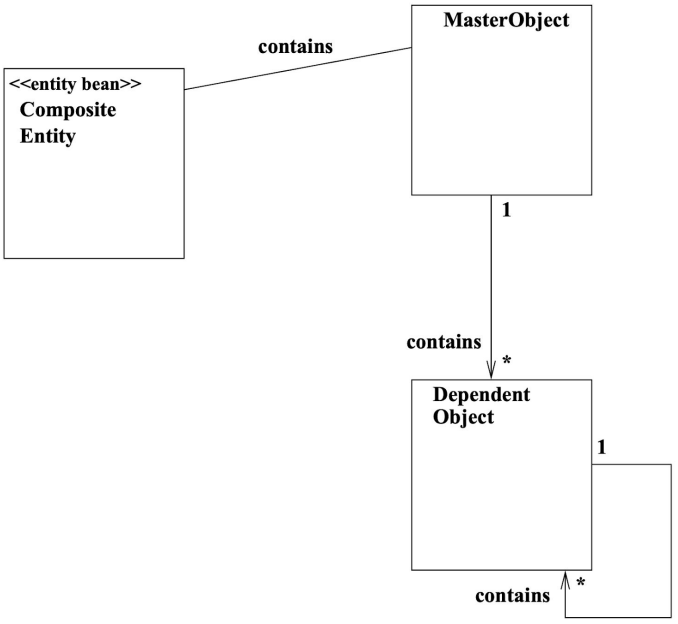
Several related use cases can be dealt with by a single session facade: if these use cases have mainly the same business objects in common.

Example: *CustomerControllerBean, AccountControllerBean, TxControllerBean*

Composite Entity

Composite Entity: manages 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. Also it leads to very many classes.
- Instead, this pattern groups related objects into single entity beans.



The elements of the pattern are:

- **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 dependents 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 dependents.
  - Dependent objects cannot be shared with other object sets.

```
.
public class BillEntity implements EntityBean
{ public int billTotal = 0;
  public List billItems = new ArrayList(); // of BillItem
  public List payments = new ArrayList(); // of Payment
  ...
}
```

Subordinate classes, *BillItem* and *Payment*, do not need their own entity beans. Can be standard Java classes.

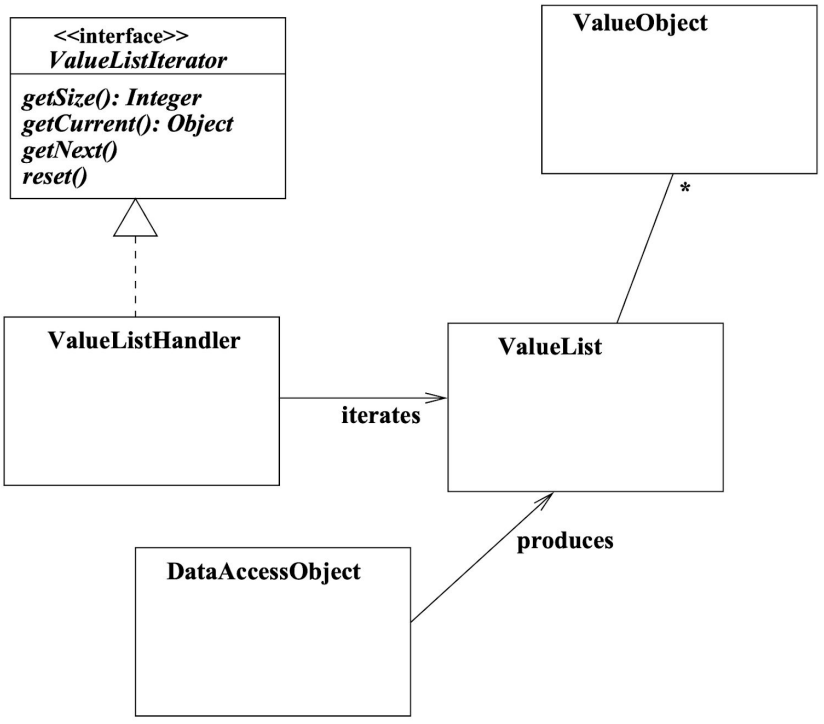
Guidelines for composite objects

- If there is association $E \rightarrow D$ 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 $E \rightarrow D$, $F \rightarrow D$, 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

Value List Handler: manages a list of data items/objects to be presented to clients.

- 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.



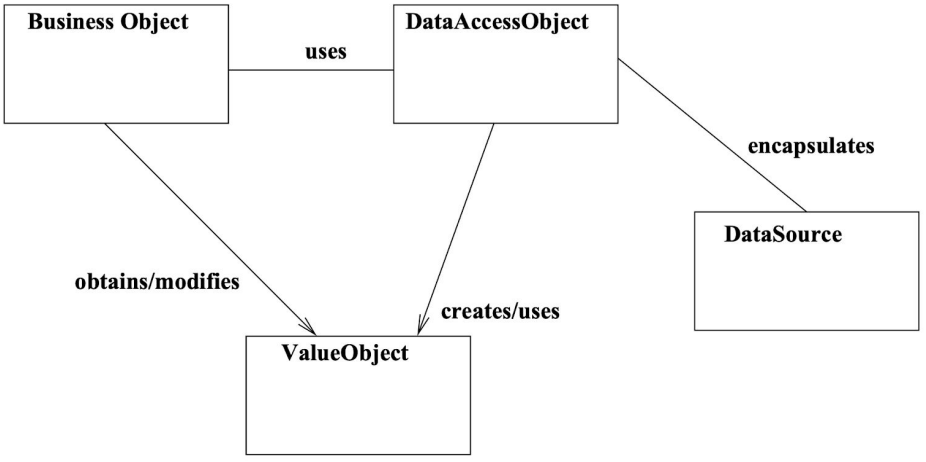
The elements of this pattern are

- **ValueListIterator:** an interface with operations such as *getCurrentElement(), getNextElements(int number), resetIndex()* to navigate along the data list.
- **ValueListHandler:** implements *ValueListIterator.*
- **DataAccessObject:** implements the database/other data access.
- **ValueList:** the actual results of a query. Can be cached.

Data Access Object

Data Access Object: 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 (JDBC, B2B services, etc) 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 DAO to interact with a data source instead.



This pattern has the following 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.

Summary

- Enterprise information systems typically involve distributed processing, and multiple client applications using same core business functionality and data.
- Business tier of an EIS can be structured around session beans and entity beans, which directly reflect high-level PIM specification of EIS as use cases and class diagrams
- For each constraint of system there should be some component within business tier which is responsible for maintaining the constraint.

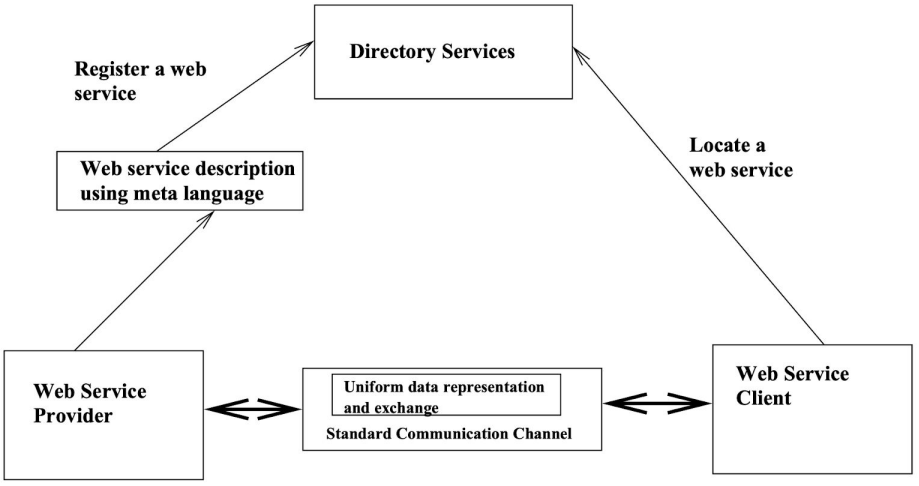
    - Class invariants and local business rules of a class can be maintained by entity bean which implements semantics of class.
    - Constraints which link states of two or more classes (and constraints on explicit associations between these classes) can be maintained either by an entity bean which encapsulates data of all these classes — in case that these classes represent closely related data, such as a main class and one or more subordinate auxiliary classes — or by a session bean which invokes operations of all entity beans implementing the classes.
- Design patterns for EIS can be defined to simplify EIS development and provide standard solutions to common EIS design problems.

Web Services

Web services: are software functions that can be invoked by clients across internet. They support integration of applications at different network locations, enabling these applications to function as if they were part of a single large software system.

Kind off outsourcing code

Web services are example of services in service-oriented architecture (SOA), and can be used to provide functionalities offered by public or private clouds in cloud computing.



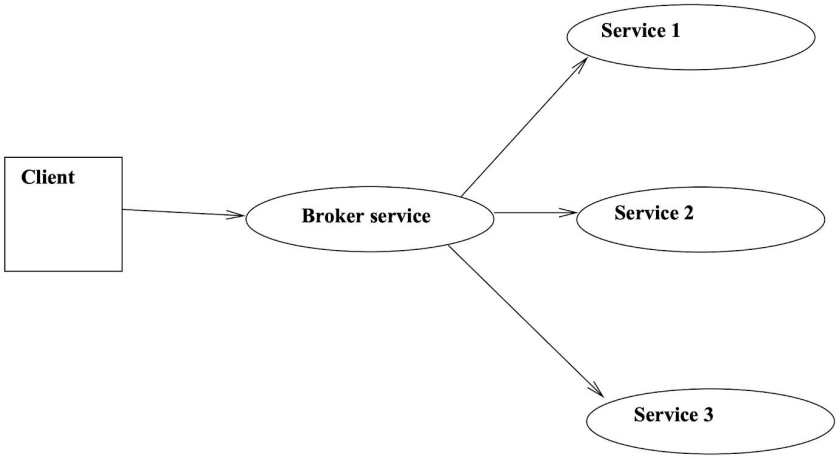
An application can make data and services available to other applications over internet:

- **Raw HTML**
- **CSV**
- **FTP**
- **SOAP**
- **WSDL**

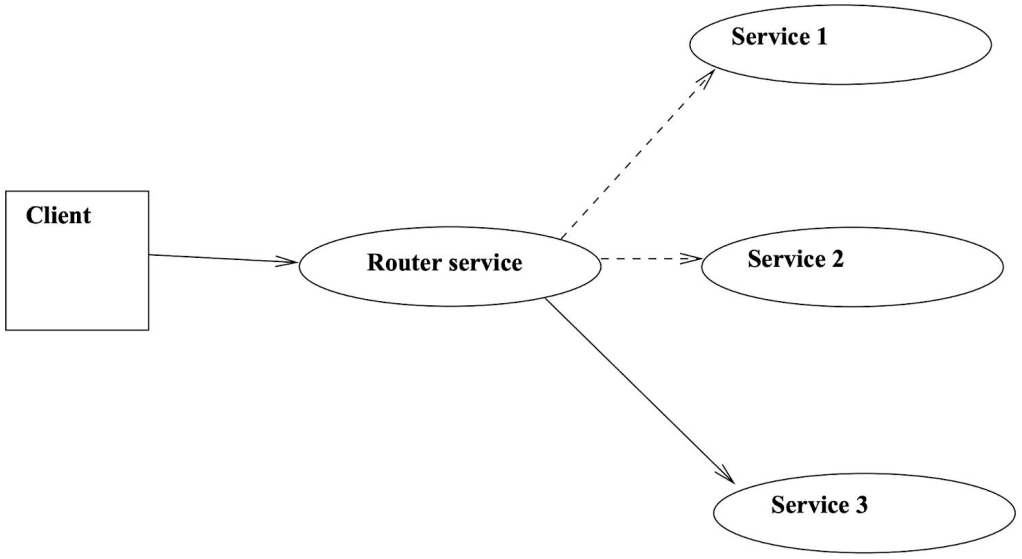
A task may be made into a web service if:

- It involves access to remote data, or other business-to-business (B2B) 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.

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



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



Python with Flask

Flask is a microframework, which is a framework that allows you to use a programming language in a way that allows for some level of customisation, for Python.

He believes that it is pretty good for quick work but that it doesn't scale too well

Some important parts

Routes Module: connects the links of the application to you implementation
e.g.

```
@app.route('/hello')
def hello_world():
   return 'hello world'
```

Which means that if the user visits http://localhost:5000/hello they will see the returned value of the function

View Template: Separates business logic from presentation
e.g.

```
<!doctype html>
<title>{% block title %}{% endblock %} - Flaskr</title>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<nav>
  <h1>Flaskr</h1>
  <ul>
    {% if g.user %}
      <li><span>{{ g.user['username'] }}</span>
      <li><a href="{{ url_for('auth.logout') }}">Log Out</a>
    {% else %}
      <li><a href="{{ url_for('auth.register') }}">Register</a>
      <li><a href="{{ url_for('auth.login') }}">Log In</a>
    {% endif %}
  </ul>
</nav>
<section class="content">
  <header>
    {% block header %}{% endblock %}
  </header>
  {% for message in get_flashed_messages() %}
    <div class="flash">{{ message }}</div>
  {% endfor %}
  {% block content %}{% endblock %}
</section>
```

It also has **great support for security**

Databases

Flask **does not support** native databases but there are a lot of **flask-wrapped databases libraries** like SQLAlchemy

Virtual Environment

Since we might have multiple projects and they might require packages in different versions or even different versions of python, we should use a virtual environment for our project with **virtualenv**

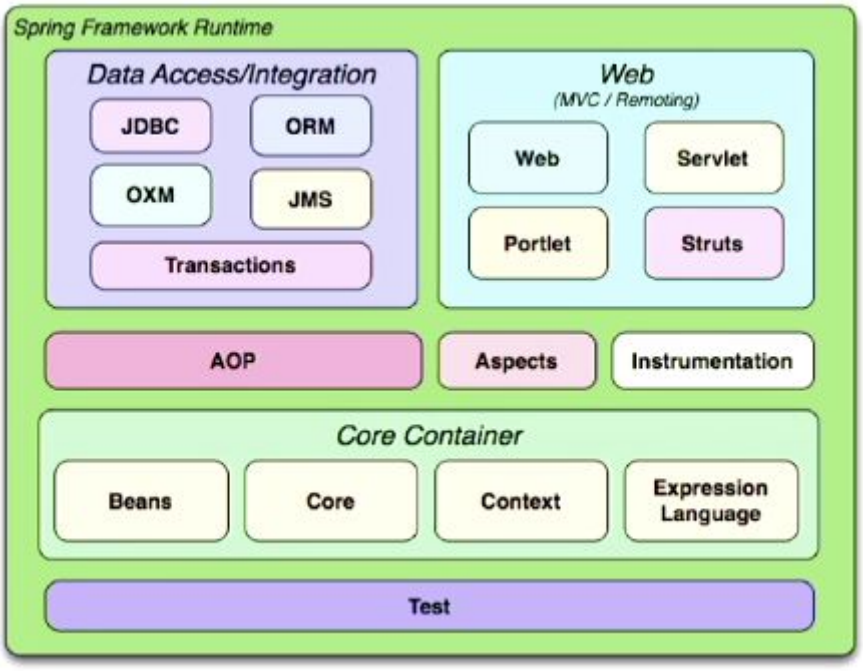
Java with Spring

Spring is possibly the best application framework on Java right now, it is open source and one of their best features is Spring Boot (for bootstrap) which makes creating a Java project really quick and simple.

How it works:

The framework is spit in modules and each in groups. There are 20 modules organized in the following groups:

- Core Container: Beans, Core, etc…
- Data Access/Integration: JDBC, Transactions, etc…
- Web (MVC/Remoting): Web, Servlet, etc…
- Aspect Oriented Programming (AOP)
- Instrumentation
- Test



Gradle

Simple and easy to use dependency manager, very good for large scale projects.

Another option is Maven, which is in XML

Tomcat

Apache Tomcat is a Servlet and JSP Servlet
You need to add this dependency to build.gradle (First one)

Ruby on Rails

Ruby is a very high level language, which means Ruby abstracts away most of the complex details of the machine. you can quickly build something from scratch with less lines of code

As a dynamically typed language, it does not have hard rules and so its close to spoken language

It has a very big and vibrant community (8th largest in StackOverflow)

Drawbacks

Scalability: because it is dynamically typed, the same thing can mean different things depending on the context. As a ruby app grows it is difficult to track and fix errors

Slow: Ruby is slow because of its flexibility, since things might have different meanings, the machine has to do a lot of referencing to know what the definition of something is. Rails is also more resource-hungry

Ruby on Rails

Ruby on Rails: is a full stack web framework that lets you build web applications quickly . It is on demand, in part because of it ease of rapid prototyping. Thought as one of the most fun frameworks and is very popular as a starting point for beginners

Features

- **Open Source, well documented and direct access to HTML, CSS, and JS**
- **Customized URL**: great for Search Engine Optimization
- **Generators**: built-in scripts designed to kickstart applications. e.g. to create a new application $ rails new appName

More on Generators

After creating a new app with the generator you get a folder with:

- **app/** - contains controllers, views, helpers, mailers, channels, jobs and assets for your app
- **config/** - application's routes, database etc..
- **db**/ - contains your database schema, as well as the db migrations
- **Gemfile, Gemfile.lock** - all the dependencies of the application
- **tests/** - Unit tests, fixtures and other test apparatus
- …

How to run

$ bin/rails server

(on Windows you have to do $ ruby bin/rails server)

Generate a new page

With $ bin/rails generate controller "ControllerName" "ActionName"

The controller will be located at:
app/controllers/ControllerName controller.rb

and the view, located at:
app/views/ControllerName/ActionName.html.erb

Routing

The routing file will be placed in config/routes.rb
Inside, it should look something like this:

```
Rails.application.routes.draw do
        get 'ControllerName/ActionName'
        root 'ControllerName#ActionName'
end
```

Javascript and ReactJS

It is a front-end library developed by Facebook initially released in 2013. It was made specifically for Single Page Applications, making it easier to for the browser to load all the data from the beginning.

Components

Components - everything is a component in React. Components are written in JSX, a version of JavaScript that has XML in it.

- It has no: controllers, view models, templates, etc...

React has some transpilers that convert the JS + JSX into JS, HTML and CSS

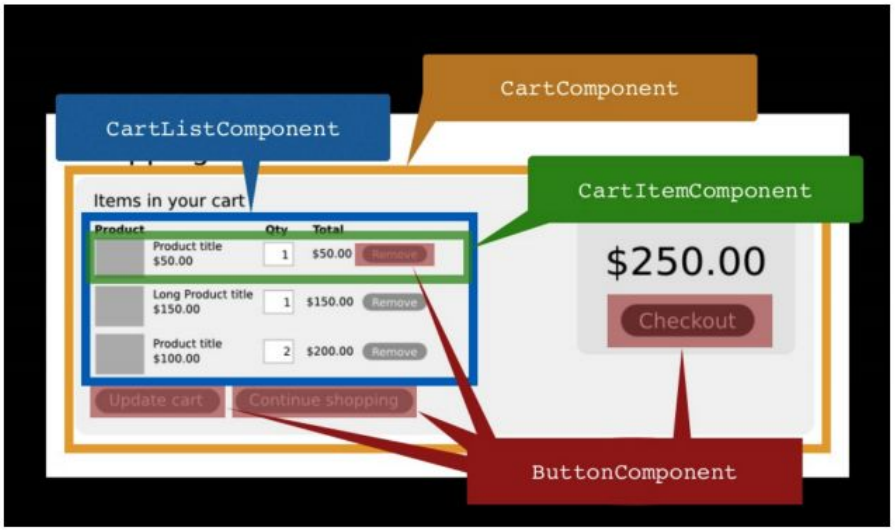


Figure: Shopping cart with all the components.

The shopping cart component would be a component with all the subcomponents.

It contradicts the Separation of **Concerns principle by adding JS and XML together**. Which should be bad! *

- However, for ReactJS it works. Instead of the norm, it follows a Separation of Components idea. This makes code more:
    - Composable
    - Reusable
    - Maintainable
    - Testable

Data Flow

Flow of Information - most other frameworks or libraries implement a 2 way data binding. React changes this by having only **one way flow**.

- Data flows from parent to child component, being sent down as a **prop** or a **state**.
    - Props are immutable
    - state is mutable

While data flows down the order, events flow up

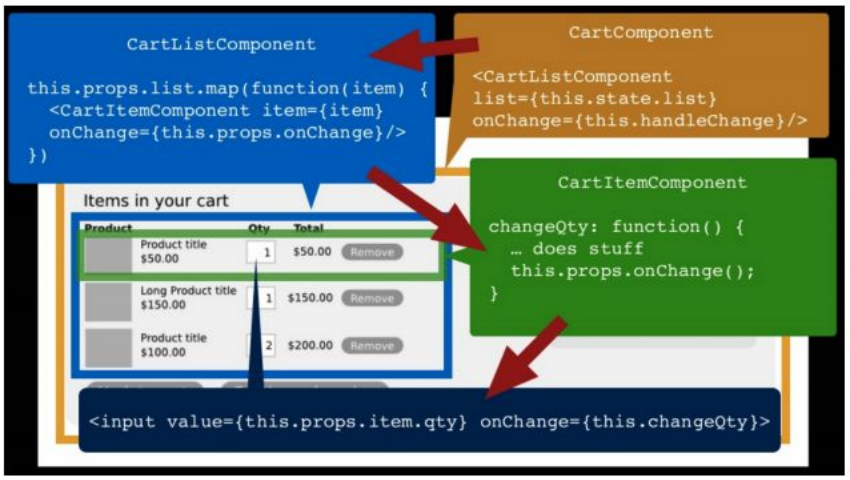


Figure: Flow of Data.

Virtual DOM

Virtual DOM: is a pure JS in memory representation of the Document Object Model.

Document Object Model (DOM) is a programming API for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. (A tree of HTML with its components inside of it).

render() function is triggered when something has been changed. It compares the real DOM to the VDOM and only updates the components that have changed.

VDOM makes React pages:

+ Really fast (once they load)
- Use a **lot** of RAM

Running Code

We use the command **$npm run start**

Routing

ReactJS is only a View Library, not a framework, and as such it does not handle anything but the front-end. You can implement any routing system that you are comfortable with. I strongly recommend React-Router.

Same for a better Data Store for our application, we might want to use redux.

Haskell and Scotty

Haskell is a polymorphically statically typed, lazy typed, purely functional language.

The language is named for Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages.

Haskell is based on the lambda calculus.

Scotty: It is a web framework in Haskell, similar to Sinatra (which is in Ruby). Unlike ReacJS which is a library.

- It is used for writing RESTful, declarative web applications.
  - A page defined as the verb, url pattern, and Text content
  - It is template-language agnostic. Anything that returns a Text value will do.

We can define a simple page running on port 3000 and giving it a parameter as *:word*

```
{-# LANGUAGE OverloadedStrings #-}

import Web.Scotty

import Data.Monoid (mconcat)

main = scotty 3000 $
  get "/:word" $ do
    beam <- param "word"
    html $ mconcat ["<h1>Scotty, ", beam, " me up!</h1>"]
```

This returns an html test list.

How to do GET, POST and PUT requests.

```
get "/" $ do
  text "gotten!"

delete "/" $ do
  text "deleted!"

post "/" $ do
  text "posted!"

put "/" $ do
  text "put-ted!"
```

```
get "/404" $ file "404.html"
```