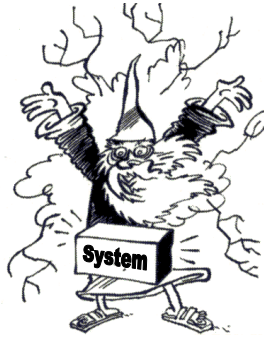
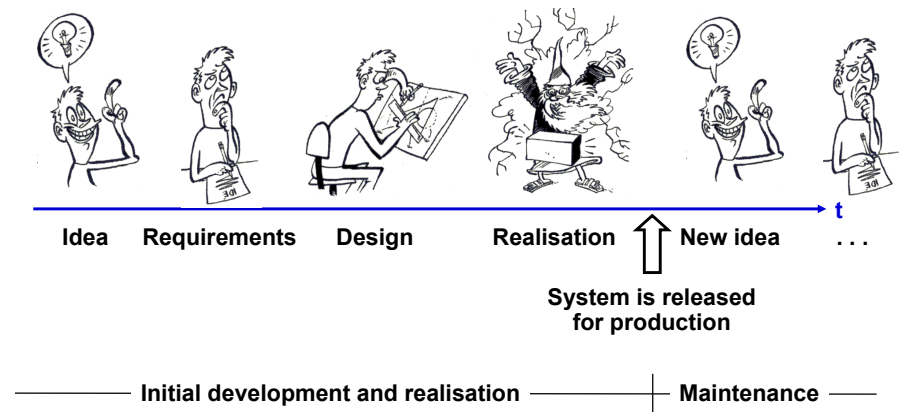


## We are going to create systems...



## The systems life cycle



## The object-oriented philosophy

- Things that should be done, should be done by a set of *collaborating objects*, each having a well-determined responsibility.
- The objects collaborate by sending *messages* to each other.
- *How* an object fulfils its responsibility, is irrelevant and unknown to the other objects (the encapsulation principle).

## What is an object?

- An object is a representation of a real "thing" or a concept.
- An object has a unique identity, an inner state, and the ability to react on messages from outside.
- An object, then, has "life". The objects in the reality are animals, plants, machines and mechanisms, and more abstract phenomena.

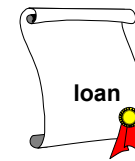
## Bringing “life” to “dead” things

- ❑ In fairy tales and animated movies, almost everything is possible – including bringing “life” to otherwise “dead” things like stones, lakes, roads, loans...  
Then they can participate actively in the story....
- ❑ Object-oriented models and programs can be designed the same way!
- ❑ Let us have a look at a couple of illustrative examples...

## The conscious loan

A loan-object may for example

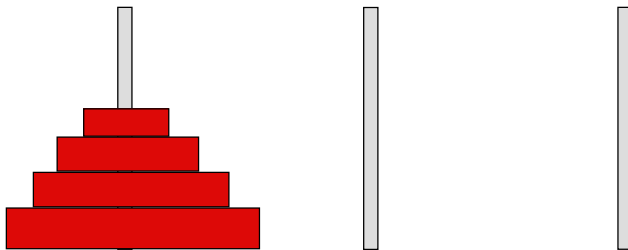
- know its own balance, interest rate, due dates etc.
- bear interest on itself
- send reminders on instalments
- accept payments



you have forgotten to pay the instalment!



## Example system: The Towers of Hanoi



The task: To move all discs from a peg to another.  
The rules: You may only move one disc at a time, and you are not allowed to place a bigger disc on a smaller disc.

## The Towers of Hanoi – a Play for 8 Actors

Manuscript-writer and director:  
Eise Nordhagen

- ❑ Role list:
  - 4 discs - 1, 2, 3, 4
  - 3 pegs – one of which is the startPeg
  - 1 player
- ❑ Each object (i.e. each person) has instructions about what to do when receiving a message.

## Class Player

```
Tell startPeg "moveDiscs";  
If returned value equal "Game successfully completed" then  
    jump up and down and clap your hands;
```

## Class Disc1

```
Boolean disc2Underneath = true;  
Method move  
Move to the Right peg;  
if disc2Underneath then  
    disc2Underneath = false;  
    Tell Left peg "moveDiscs";  
else  
    disc2Underneath = true;  
    Tell Right peg "moveDiscs";
```

## Class Disc2and4

```
Method move  
Move to the Right peg;  
Tell the Right peg "moveDiscs";
```

## Class Disc3

```
Method move  
Move to the Right peg;  
Tell the Right peg "moveDiscs";
```

## Class Peg

Method *moveDiscs*

Do you have discs? - ->

Yes: Tell the Top disc: "move"

No: Tell the Right peg "moveDiscs"

## Class StartPeg

Method *moveDiscs*

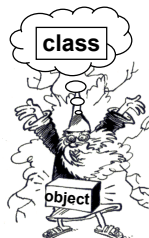
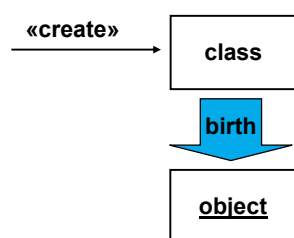
Do you have discs? - ->

Yes: Tell the top disc "move"

No: Return "Game successfully completed"

## Where does the object come from?

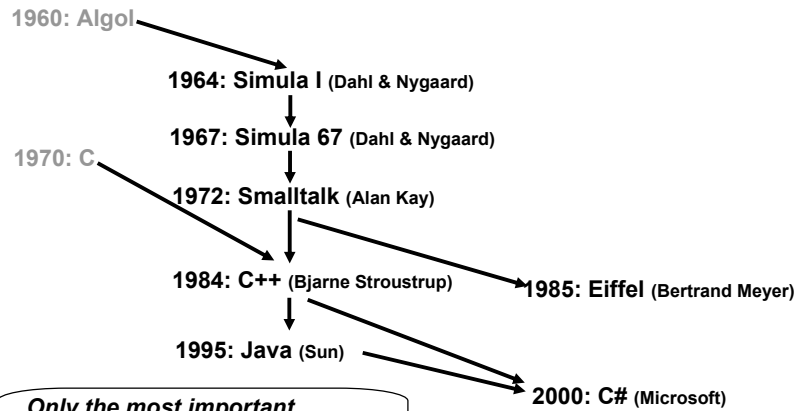
- ❑ The class – the object-oriented DNA
- ❑ A class can be regarded as a construction drawing for objects
- ❑ An object is created by sending the message «create» («new») to the class



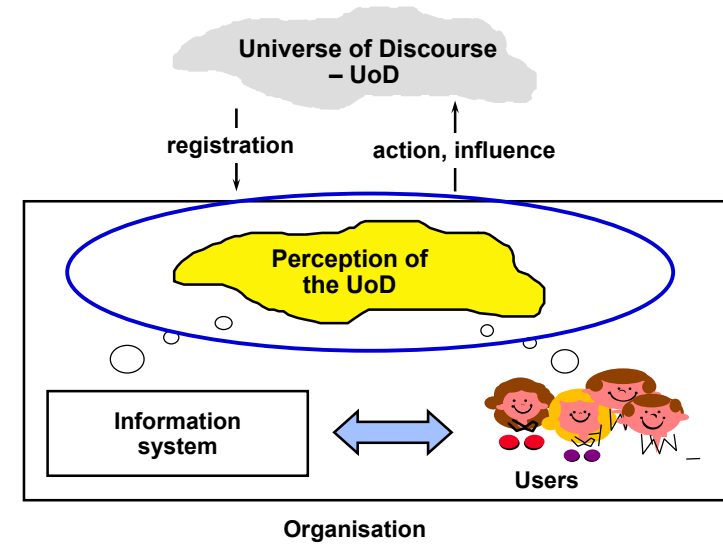
## How does an object die?

- ❑ By killing itself
- ❑ By being killed by another object
- ❑ By being killed by the "garbage collector" because no other object knows about it

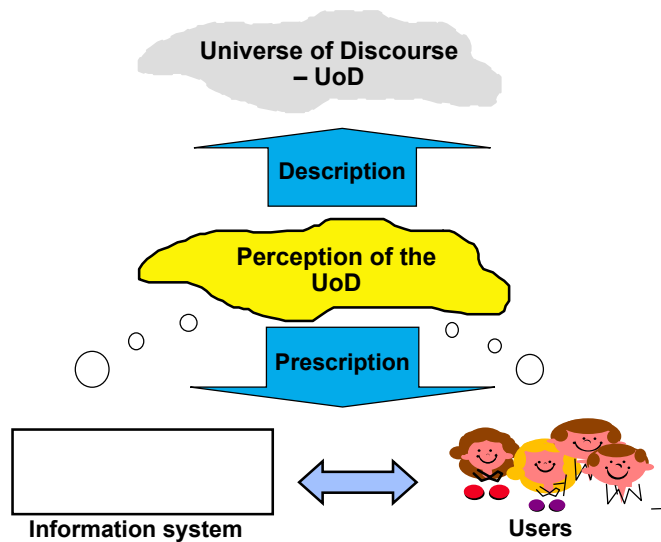
## The history of Object orientation – the languages



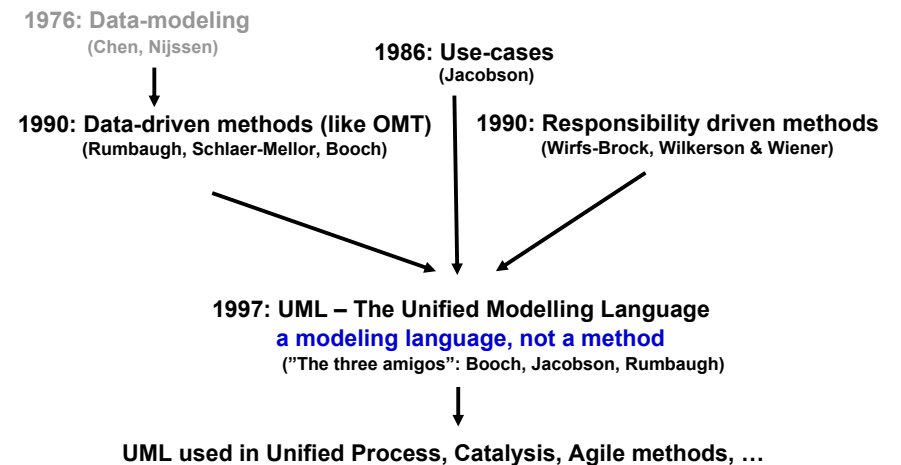
## The model documents our perception of the reality



## The dual purpose of the model



## The history of Object orientation – the methods



# The Unified Modeling Language - UML



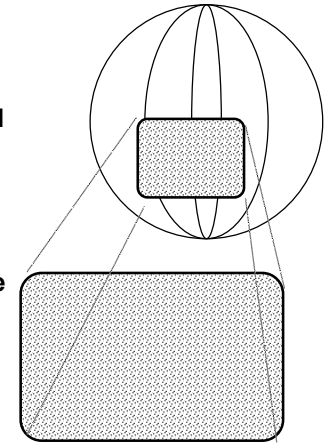
A set of formal diagramming techniques, developed by trendsetting groups within OO, "standardised" by Object Management Group (OMG)

Usage Diagram type	Use-Case view	Logical view	Component view	Concurrency view	Deployment view
Use-Case diagram	■				
Class/object diagram		■			
Sequence diagram		■		■	
Collaboration diagram		■		■	
State diagram		■		■	
Activity diagram		■		■	
Component diagram			■		
Deployment diagram				■	■

# We are going to make models of systems

Two important definitions:

- A *system* is a part of the world that we choose to regard as an entity, separated from the rest of the world within the observation period.
- A *model* is a representation of something, where certain features, which are important for the purpose of the model, are emphasised, whereas the other features are left out.



# Why models?

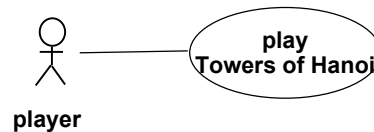
- To understand the system
- To describe the system
- To have a basis for developing an alternative (computerised) system

# Our models

- should be object-oriented, i.e. build on the paradigm of collaborating objects
- should have a purpose
  - otherwise it is impossible to decide what should be included in the model, and what should be left out...  
... there is no such thing as "a complete model"
- should be realised in an object-oriented language, i.e. Java

## The Towers of Hanoi – Use case diagram

	Usage	Use-Case view	Logical view	Component view	Concurrency view	Deployment view
Diagram diagram						
Class/object diagram						
Sequence diagram						
Collaboration diagram						
State diagram						
Activity diagram						
Component diagram						
Deployment diagram						



## Tower of Hanoi – Use case textual specification

**Name:** play Tower of Hanoi

**Actor:** Player

**Precondition:** All disks on the startPeg in the correct sequence

**Postcondition:** All discs on another peg in the correct sequence

**Trigger:** Actor wants to play the game

**Normal sequence of events:**

- 1 Player asks the system to play the game
- 2 System responds "Game successfully completed"
- 3 Player jumps up and down and claps his hands

**Variations:**

- 2a The system does not respond within reasonable time
  - 1 Player goes to 1 or give up

**Related information:** None

## Tower of Hanoi – Another Use case textual specification

**Name:** play Tower of Hanoi

**Actor:** Player

**Precondition:** All disks on the startPeg in the correct sequence

**Postcondition:** All discs on another peg in the correct sequence

**Trigger:** Actor wants to play the game

**Normal sequence of events:**

- 1 System asks the player to move a disc
- 2 Player tells the system to move a disc
- 3 System moves the disc.  
If this was the final move, it responds "Game successfully completed", otherwise it goes to 1.
- 4 Player jumps up and down and claps his hands

**Variations:**

- 3a The system finds that the move as illegal
  - 1 The system responds "Illegal move" and goes to 1

**Related information:** None

## What can we learn from this example?

- We must decide upon the distribution of responsibilities between the actor and the system
- The documentation of this distribution is laid down in the Use case model
- The amount of interaction between the actor and the system may vary from almost nothing (like in pure simulation systems or in pure calculations) to a lot (like in administrative systems)
- If we regard the "system" as the "computerised system", we hereby decides upon the distribution of responsibilities between the users and the computer!

... and this is a mixed blessing...

## The next step – design of object collaboration

- What objects do we need, how should they behave and how should they collaborate in order to fulfil the obligations of the system, as laid down in the Use case model?
- This is the real difficult part of object oriented design...

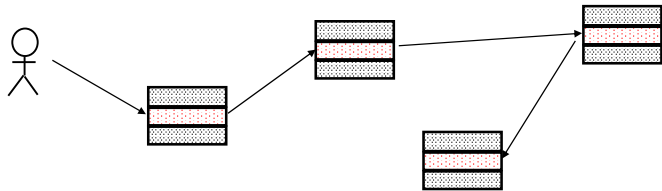
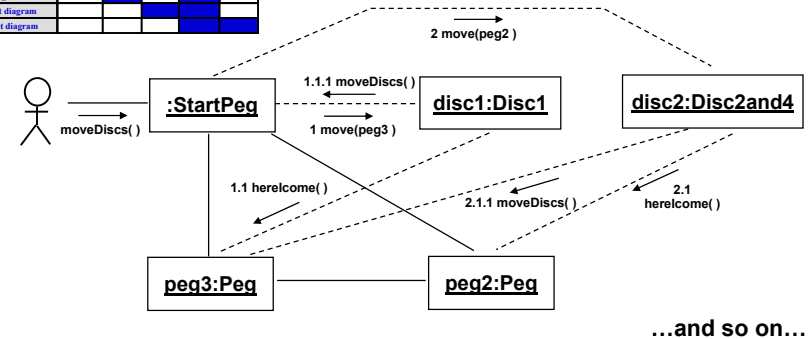


Diagram type	Usage	Use-Case view	Logical view	Component view	Concurrency view	Deployment view
Diagram type						
Class/object diagram						
Sequence diagram						
Collaboration diagram						
State diagram						
Activity diagram						
Component diagram						
Deployment diagram						

## The Towers of Hanoi – the Collaboration diagram

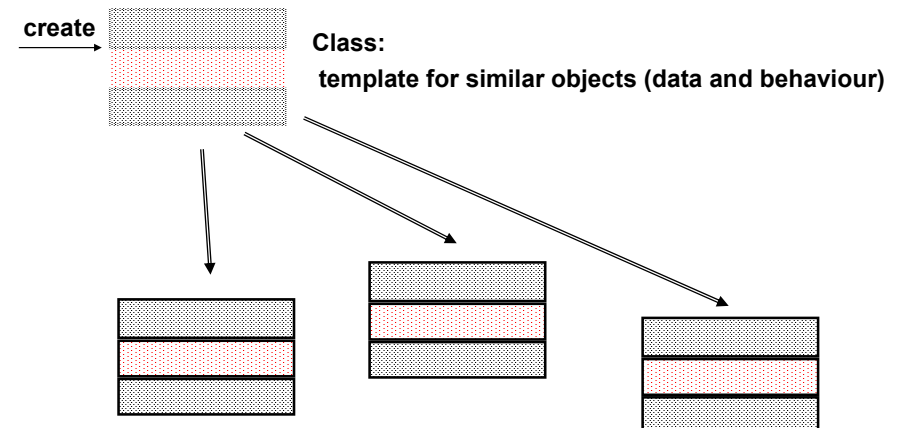


Note that a disc must ask its peg for what peg is left or right. An alternative is that the disc ask its peg to relay the message to the left or right peg. This will make the collaboration diagram simpler – and perhaps even the program

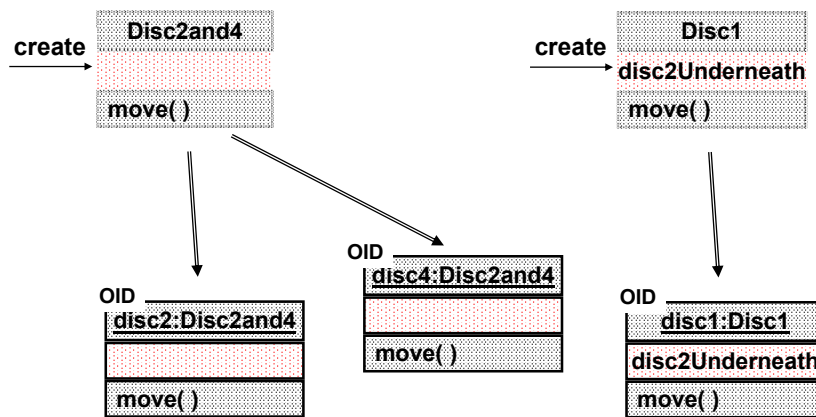
## Objects and classes

- Class
  - A description (template) for what the objects should look like (e.g. What a bank-account should look like)
- Objects
  - There may exist an arbitrary number of objects of a class (e.g. several 'bank-account'-objects in a running banking system)

## Objects and classes



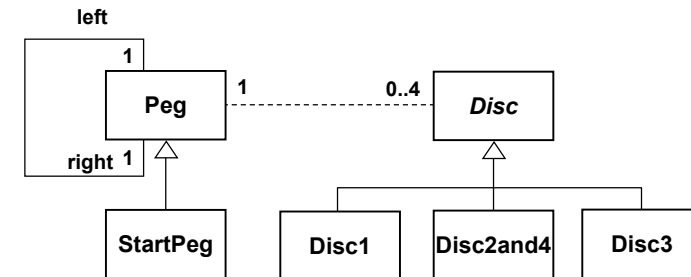
## Objects and classes - example



	Usage	Use-Case view	Logical view	Component view	Concurrency view	Deployment view
Diagram <b>Diagram</b>						
Class/object diagram						
Sequence diagram						
Collaboration diagram						
State diagram						
Activity diagram						
Component diagram						
Deployment diagram						

## The Towers of Hanoi – Class diagram

After some work (see later), we find that the following Class diagram is sufficient for creating the objects in the Collaboration diagram



## From model to program

Now we only have to

- write the program, i.e. code all the classes in an object-oriented language
- compile the program
- call the first class, which will create an object – an instance of itself – and call a method in that object
- ...then just watch things go!
- well... oh yes... we have something called the User interface...

## Development tools

Development tools make the programming easier

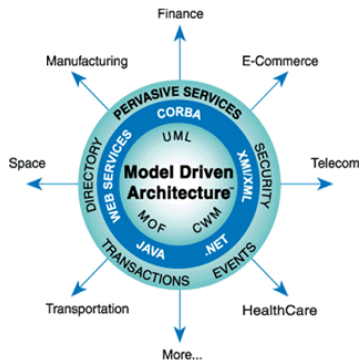
- on-the-fly syntax check
- class-hierarchy shown graphically
- list of callable methods readily available
- execution with breakpoints...
- ... and with display of selected states (object variables)
- ...

## The MDA-vision – do it automatically ☺

<http://www.omg.org/mda/>

OMG Model Driven Architecture

### How Systems Will Be Built



MDA<sup>®</sup> provides an open, vendor-neutral approach to the challenge of business and technology change. Based firmly upon OMG's established standards\*, MDA aims to separate business or application logic from underlying platform technology. Platform-independent applications built using MDA and associated standards can be realized on a range of open and proprietary platforms, including CORBA<sup>®</sup>, J2EE, .NET, and Web Services or other Web-based platforms. Fully-specified platform-independent models (including behavior) can enable intellectual property to move away from technology-specific code, helping to insulate business applications from technology evolution, and further enable interoperability. In addition, business applications, freed from technology specifics, will be more able to evolve at the different pace of business evolution.

\* **Key standards** that make up the MDA suite of standards include Unified Modeling Language (UML); Meta-Object Facility (MOF); XML Meta-Data Interchange (XMI); and Common Warehouse Meta-model (CWM).

## Why object-orientation?

- ❑ **Natural way of thinking**  
(... at least when you get used to it)
- ❑ **Better possibilities for reuse of models and programs**
- ❑ **Ease of maintenance**  
As long as an object fulfils its obligations with regard to services, it may be changed in any way you like
- ❑ **Easy distribution**  
Messages may be passed over a network

## Our case: The forest simulator



- ❑ We are going to build a system which simulates the evolution of a forest.
- ❑ We will do that by populating the forest with tree-objects, each having its own state and behaviour.
- ❑ We may intervene with nature by felling or planting trees.

## Template for a requirement specification

- ❑ **General system description (vision)**
  - What is the purpose of the system?
- ❑ **Functional requirements**
  - What services are the system going to offer?
- ❑ **Quality requirements (non-functional requirements)**
  - i.e. requirements regarding response-times, user friendliness, maintainability, up-time...
- ❑ **Limitations (influences functional and quality requirements)**
  - Legislation, rules, ethics
  - Technology
  - Standards
  - Economy
  - Time

## How to find the requirements

- ❑ Interviews with customers and potential users
- ❑ Observation of existing working processes
- ❑ Investigation of standards, legislation and business rules
- ❑ Incremental (iterative) system development, including prototyping

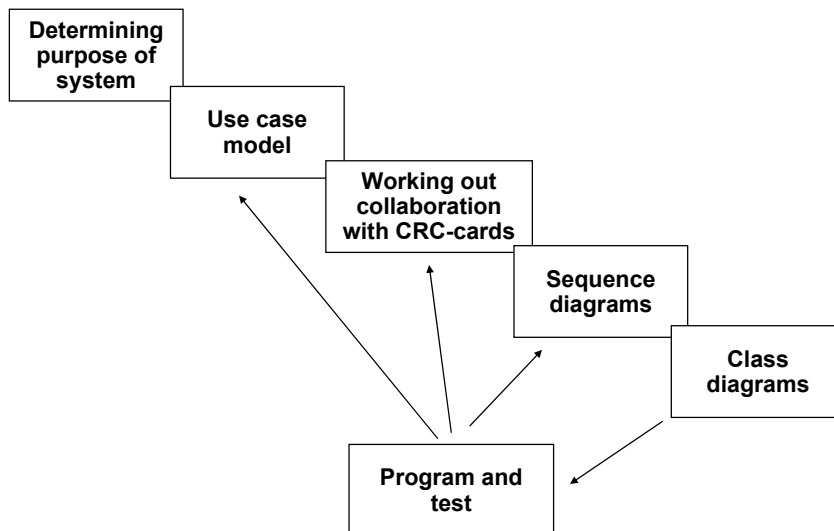
Such user-centric requirement specification may make the user happy. Will it make the customer, the shareholder, or the society at large happy?



## System development strategies

- ❑ Iterative development with multiple deliveries of usable systems
- ❑ Mainly responsibility driven development
- ❑ Basic functionality first, with only rough tree-growth models and a simple user interface
- ❑ Improvement according to needs and priorities
  - some ideas:
    - Better tree-growth models
    - Better reproduction models
    - Including weather and climate
    - Functionality for tree felling and planting
    - ...

## Sequence of activities (Responsibility driven)

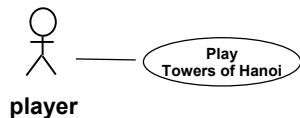


## Method for responsibility driven Object oriented analysis and design

- ❑ Analysis of requirements
  - ➔ 1. Identify actors and their goals
  - 2. Make a high level Use case diagram
  - 3. Make a textual specification of each Use case with normal flow of events and variations
- ❑ Object design
  - 4. Identify objects and assign responsibilities to them (CRC)
  - 5. Make Sequence diagrams for important Use cases
  - 6. Make Class diagram for each Use case
  - 7. Make an integrated Class diagram for the system

## What is a Use case model?

- ❑ A Use case model describes the *functional requirements* for the system
- ❑ A complete Use case model consists of two components:
  - *UML Use case diagrams* that gives a visual representation of the model (an overview of actors, use cases and the relationships between them)
  - *Textual specifications* of each Use case



**Name:** play Tower of Hanoi  
**Actor:** Player  
**Precondition:** All disks on the startPeg in the correct sequence  
**Postcondition:** All discs on another peg in the correct sequence  
**Trigger:** Actor wants to play the game  
**Normal sequence of events:**  
1. Player asks the system to play the game  
2. System responds "Game successfully completed"  
3. Player jumps up and down and claps his hands  
**Variations:**  
2a The system does not respond within reasonable time  
1 Player goes to 1 or give up  
**Related information:** None

## Actors

- ❑ **Actor:** Describes a certain type of user of the system
- ❑ Two types of actors
  - A human playing a certain role (“a human with a hat”) having interest in using the system
  - An external information system interfacing with the system at hand
- ❑ **UML symbol for an actor:**



## Actors (cont.)

- ❑ An actor describes one (and only one) role that a user (a human or another information system) has in relationship with the system at hand
- ❑ Example: A person (Bob) may have several roles
  - Actor 1: “Bank customer”
  - Actor 2: “Bank employee”... but “Bob” is no actor in a banking system



## Actors (cont.)

- Actors delimit the usability area of the system:
- ❑ What types of users will the system have?
  - ❑ What are the *goals* of these users?
    - The goals of the actors determines what services the system should offer
  - ❑ *Primary actors* initiates use cases that fulfils their goals.
  - ❑ *Secondary actors* facilitates the execution of use cases
  - ❑ What actors a system has, and whether an actor is primary or secondary, depends on where we draw the boundary between the system and its environments

## Actors (cont.)


Actors have *goals*. The goals are excellent departing points for finding Use cases:

- “Bank customer”:
  - Goal1: Withdraw money
  - Goal2: Transfer money
  - Goal3: Deposit money
  - Goal4: Get a loan
- “Bank employee”
  - Goal1: Decide on loan application

## Method for responsibility driven Object oriented analysis and design

- Analysis of requirements
  1. Identify actors and their goals
  - ➔ 2. Make a high level Use case diagram
  3. Make a textual specification of each Use case with normal flow of events and variations
- Object design
  4. Identify objects and assign responsibilities to them (CRC)
  5. Make Sequence diagrams for important Use cases
  6. Make Class diagram for each Use case
  7. Make an integrated Class diagram for the system

## Use case

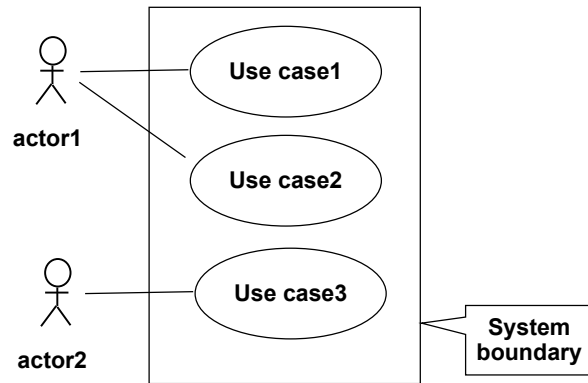
- A Use case
  - describes a certain functionality of the system
  - UML symbol for a Use case: 
- The name of a Use case is derived from a flow of events where everything goes well (“happy day scenario”).  
Tip: Use *active* sentences!
- Examples:
  - *Withdraw money* (not “money withdrawal”)
  - *Transfer money* (not “money transfer”)
  - *Deposit money* (not “money deposit”)
  - *Create account* (not “account creation”)

## A Use case diagram

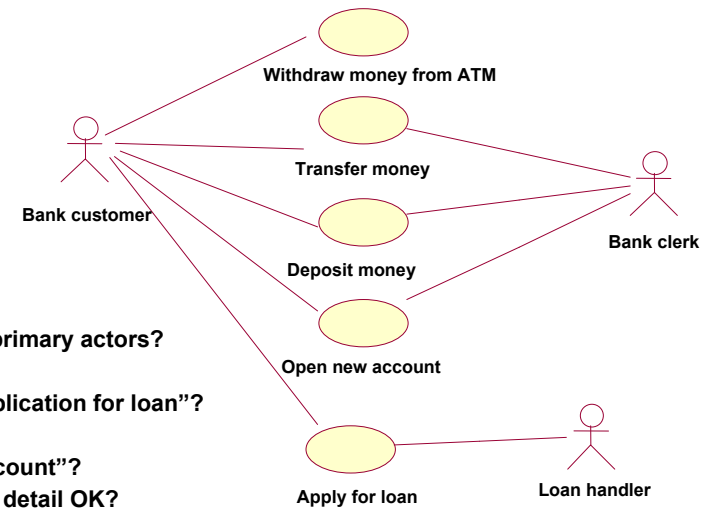
- A Use case diagram shows the existence of interactions between an actor and a Use case.
- The connection line indicates a *communication relationship*, i.e. that there is some kind of interaction between the actor and the Use case.



## A complete Use case diagram



## Example: Bank Use case diagram



Who are the primary actors?  
What about  
"Handle application for loan"?  
What about  
"Manage account"?  
Is the level of detail OK?

## How to make Use case models

- It is often difficult to find the "right" Use cases:
  - How big should a Use case be?
  - How detailed should a Use case be written?
  - How many Use cases are there in the complete model?
- There are no definite answers, but here are some guidelines:
  1. Identify the boundary between the system and the environment. The boundary determine the primary actors.
  2. Let the goals of the primary actors govern the choice of Use cases.
  3. If the Use case model turns out to be very complex, you may go to a higher level of abstraction with less details, and where each Use case includes a broader functionality.

## Method for responsibility driven Object oriented analysis and design

- Analysis of requirements
  1. Identify actors and their goals
  2. Make a high level Use case diagram
  - ➔ 3. Make a textual specification of each Use case with normal flow of events and variations
- Object design
  4. Identify objects and assign responsibilities to them (CRC)
  5. Make Sequence diagrams for important Use cases
  6. Make Class diagram for each Use case
  7. Make an integrated Class diagram for the system

## What is a Use case textual specification?

- ❑ A Use case is incomplete without a textual specification (... or some other detailed description)!
- ❑ A Use case textual specification describes the complete sequence of interactions between the actor and the Use case in order to reach the goal of the actor.

## Template for the textual specification of a Use case

**Name:** The name of the Use case

**Actor:** The role that is initiating the Use case

**Precondition** (optional): The conditions that must be fulfilled in order to execute the Use case

**Postcondition** (optional): The state of system and actor after the Use case has been executed, both in case of normal sequence of events and in case of variations.

**Trigger:** A situation or an event that triggers the actor to initiate the Use case

**Normal sequence of events:** The happy day scenario, when the goal of the actor is fulfilled in the simplest possible way

**Variations:** The special cases which causes exceptions from the normal sequence of events. Format: **<Condition>: <consequences>**

**Related information** (optional): For example quality requirements to the Use case (i.e. "average response time must be shorter than 10 seconds")

## Tower of Hanoi – Use case textual specification

**Name:** play Tower of Hanoi

**Actor:** Player

**Precondition:** All disks on the startPeg in the correct sequence

**Postcondition:** All discs on another peg in the correct sequence

**Trigger:** Actor wants to play the game

**Normal sequence of events:**

- 1 Actor asks the system to play the game
- 2 System responds "Game successfully completed"
- 3 Actor jumps up and down and claps his hands

**Variations:**

- 2a The system does not respond within reasonable time  
1 Actor goes to 1 or give up

**Related information:** None

## Method for responsibility driven Object oriented analysis and design

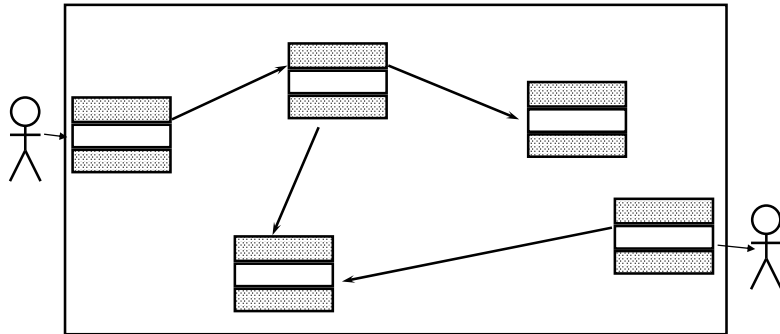
- ❑ Analysis of requirements

1. Identify actors and their goals
2. Make a high level Use case diagram
3. Make a textual specification of each Use case with normal flow of events and variations

- ❑ Object design

- ➡ 4. Identify objects and assign responsibilities to them (CRC)
5. Make Sequence diagrams for important Use cases
6. Make Class diagram for each Use case
7. Make an integrated Class diagram for the system

## The challenge in designing Object oriented models

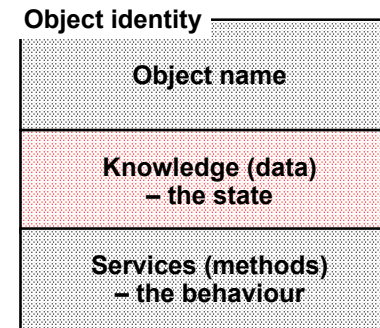


Given a set of Use cases:  
How to find objects and assign responsibilities such that the Use cases are satisfied!

## The anatomy of an object

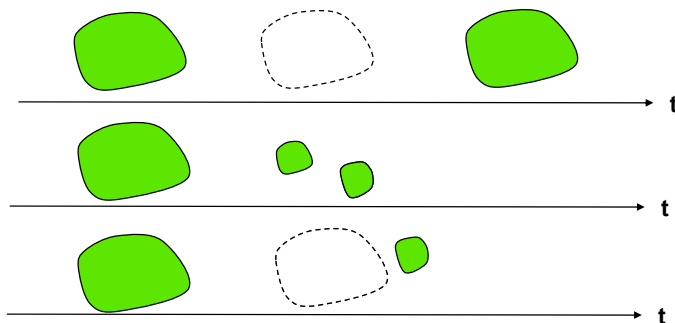
The encapsulation principle:

- ❑ Only necessary services are accessible from the outside
- ❑ The knowledge (the state) is accessible only through services

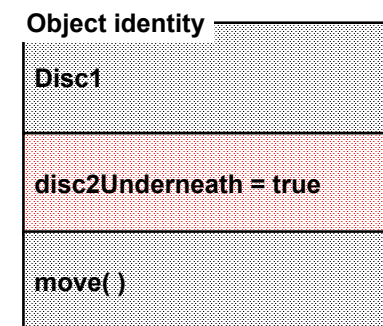


## The Object identity

- ❑ An object never changes its identity...  
... a glitchy concept!
- ❑ If we – over time – replace all the parts of a bicycle – is it then still the same bicycle?
- ❑ When is a forest the same or a different forest?



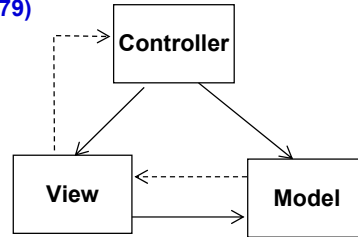
## The anatomy of an object – example



## The Model-View-Controller paradigm

(Reenskaug 1979)

- ❑ The user interacts with the user interface in some way (e.g., user presses a button)
- ❑ A controller handles the input event from the user interface, often via a registered handler or callback.
- ❑ The controller accesses the model, possibly updating it in a way appropriate to the user's action.
- ❑ A view uses the model to generate an appropriate user interface. The view gets its own data from the model. The model has no direct knowledge of the view. However, the *observer pattern* can be used to allow the model to indirectly notify interested parties – potentially including views – of a change.



## Three types of objects

The Model-View-Controller paradigm asks for three types of objects

- ❑ In the model: Business objects (“entity objects”)
- ❑ In the controller: Control objects
- ❑ In the view: Boundary objects

## Business objects (entity objects)

- ❑ Represents the “things” the business is handling, like article, offer, order, account, customer, peg, disc, tree, ...
- ❑ A business object may have a very long lifetime – may be at least as long as the business! We want the business objects to be *persistent*, hence, business objects are candidates for being stored in a database.

## Control objects

- ❑ Represents something that *is done* in the business
- ❑ To simplify things, we will stick to the rule that there should be one control object per Use case. Then we can name the control object after the name of the Use case.
- ❑ The control object manages the flow of events in a Use case.
- ❑ The control object normally does not live longer than until the execution of the Use case is completed.

## Boundary objects

- ❑ Boundary objects are activated by actor actions through the User interface
  - e.g. when the actor wants to start to play a game by clicking the button "Start" in the user interface.
- ❑ The boundary object will then create a control object and leave the control of the flow of events to it.
- ❑ We may regard a boundary object as a kind of link between an unspecified user interface and a control object. Boundary objects communicate only with actors (through the user interface) and control objects.

## How to find business objects

- ❑ A good point of departure is the language of The Universe of Discourse
  - e.g. "product", "ingredient", "customer", "student", "account", "deposit", "disc", "peg"
  - Look for concepts in the Use case specifications!
- ❑ Look at the informal description of the system – if there is one
  - Look for nouns
  - Check the adjectives
  - Make passive sentences active
- ❑ Start at the boundary
  - The interactions with the user
  - Interfaces with other information systems

## Types of business objects

### Types of business objects [Shlaer & Mellor]

- ❑ Tangible or "real world" things – book, tree, course
- ❑ Roles – student, director, manager
- ❑ Events – arrival, leaving, request
- ❑ Interactions – meeting, intersection

## Setting up the collaboration

- ❑ The primary task is to design the collaboration between the control object and the business objects (also between the business objects) so that the Use case specification is satisfied.
- ❑ Whether the control object should limit itself to just start the flow of events and leave the rest to the business objects (delegated control style) or take a more active part in controlling the flow of events (centralised control style) is an interesting design decision with no definite answer.
- ❑ A good tool for experimenting with the collaboration design: Class-Responsibility-Collaboration-cards (*CRC-cards*).

## A CRC-card

<b>Name of object</b> «type of object»	
<b>Responsibility</b> <ul style="list-style-type: none"><li>• What the object must know</li><li>• What the object should do</li></ul>	<b>List of collaborating objects</b> (objects that may be delegated tasks)

Class-Responsibility-Collaboration  
Kent Beck, Ward Cunningham:  
*A Laboratory for Teaching Object-Oriented Thinking*

## Example of CRC-card for a class

<b>Disc</b>	
<b>Responsibility</b>  Represents a disc in Tower of Hanoi Should know where it is and where it should go	<b>Collaborators</b>  The pegs

## Example of CRC-card for an object

<b><u>disc1:Disc</u></b>	
<b>Responsibility</b>  Represent the smallest dics in the stack of discs in Tower of Hanoi: Should know where it is and where it should go (is always able to move)	<b>Collaborators</b>  Peg A, B og C

## Example of CRC-card for an object (more detailed)

<b><u>disc1:Disc1</u></b> «entity»	
Know whether Disc2 is under me	
Ask a certain peg to move its top disc	Peg

## How to find responsibilities?

- ❑ What is the purpose of the object?
- ❑ Why did you select it as an object?
- ❑ Look for information that the object must remember
- ❑ Look for actions the object must perform
- ❑ Look for relationships between objects

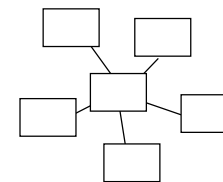
## Rules of thumb for assigning responsibilities

- ❑ Distribute the responsibilities evenly in the system  
(... however: Centralised vs. delegated control style)
- ❑ Initially, describe a responsibility in rather general terms
- ❑ Keep responsibilities and the data needed to fulfil those responsibilities together
- ❑ Keep data about one thing in one place
- ❑ A part of a responsibility may be delegated to another object
- ❑ The principle of high cohesion and low coupling

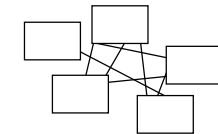
## Playing the collaboration game with the CRC-cards

- ❑ Get together a group of people
- ❑ Distribute the CRC-cards to the individuals
- ❑ Play the collaboration game by executing what the object should do when receiving a message
- ❑ Try the “happy day scenario” first, then the exceptions
- ❑ Note that we play with *object-cards*, not with class-cards!  
(Classes are just drawings, and can not execute anything!  
... well, with some slight modifications, jf. static methods in Java)
- ❑ The purpose of the game is to find
  - objects that are missing
  - objects that are superfluous because they do nothing
  - messages that are not handled by any object
  - bad design

## Warning signals in the design



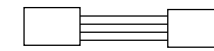
Central power



Spaghetti

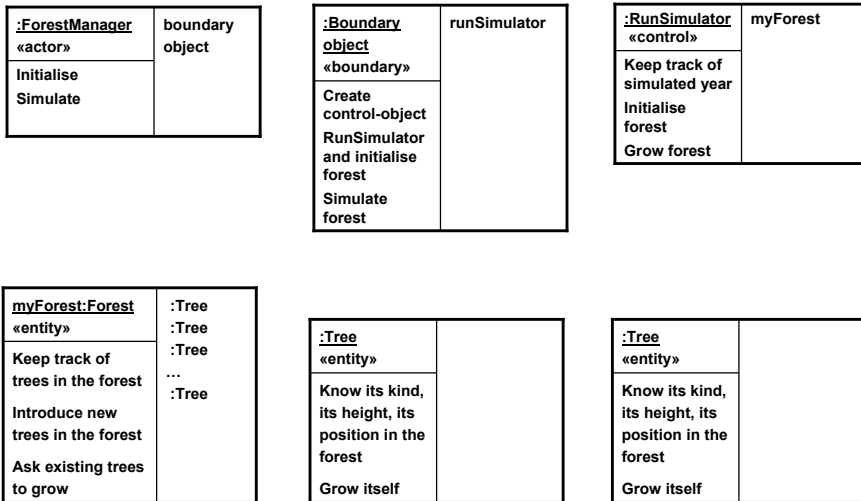


Avoiding responsibilities  
- too much delegation



Standing on  
each others feet

## CRC-cards for the Forest simulator



## Method for responsibility driven Object oriented analysis and design

### Analysis of requirements

1. Identify actors and their goals
2. Make a high level Use case diagram
3. Make a textual specification of each Use case with normal flow of events and variations

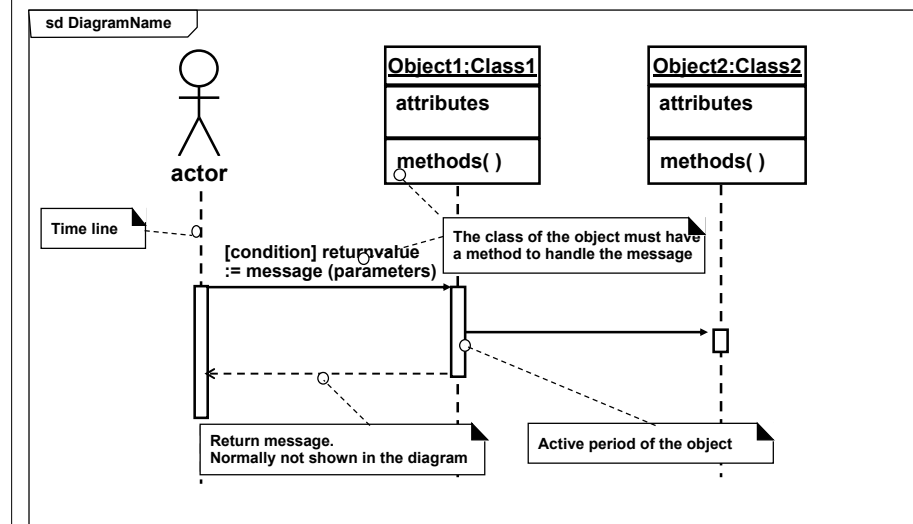
### Object design

4. Identify objects and assign responsibilities to them (CRC)
- ➔ 5. Make Sequence diagrams for important Use cases
6. Make Class diagram for each Use case
7. Make an integrated Class diagram for the system

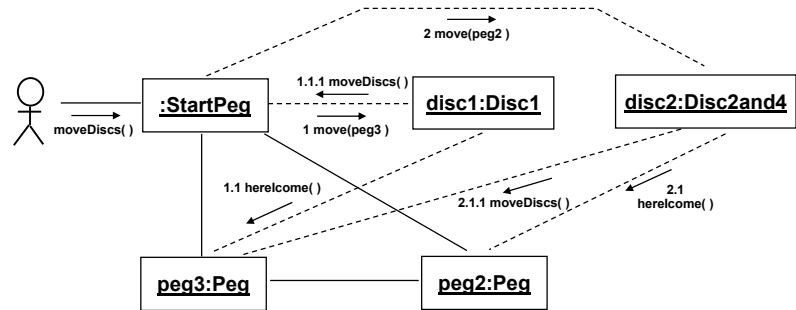
## UML Sequence diagrams

- A UML Sequence diagram shows the interactions between actors and objects for a certain Use case
  - The focus is on how objects collaborate to solve a certain task – a Use case
  - The sequence diagram is often useful to identify the necessary methods of the objects
- An alternative way to describe the interactions is by an UML Collaboration diagram

## The anatomy of a Sequence diagram



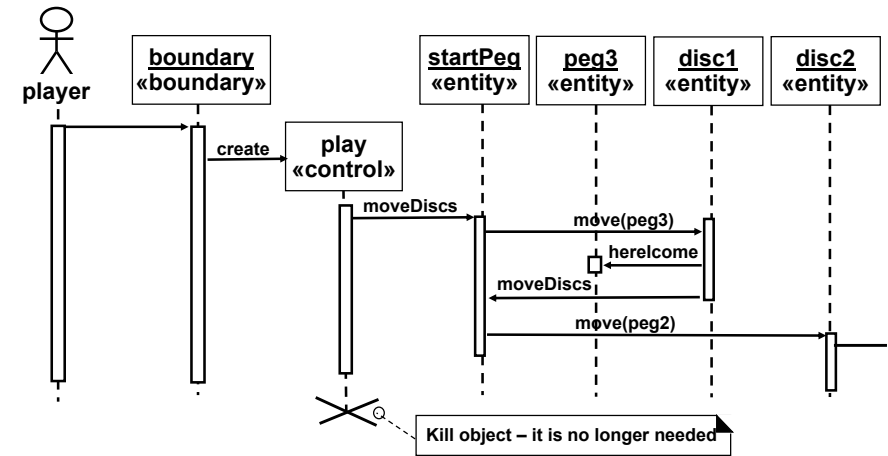
## The Towers of Hanoi – the Collaboration diagram



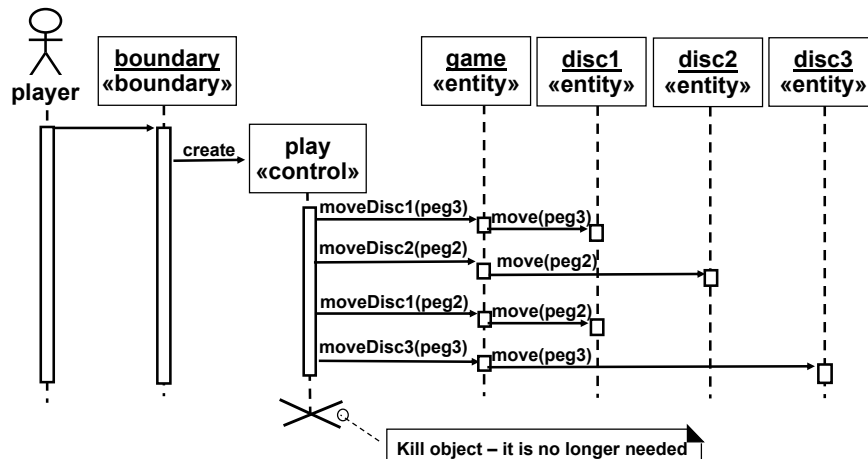
...and so on...

Note that a disc must ask its peg for what peg is left or right. An alternative is that the disc ask its peg to relay the message to the left or right peg. This will make the collaboration diagram simpler – and perhaps even the program

## Typical appearance of an UML Sequence diagram – delegated control style



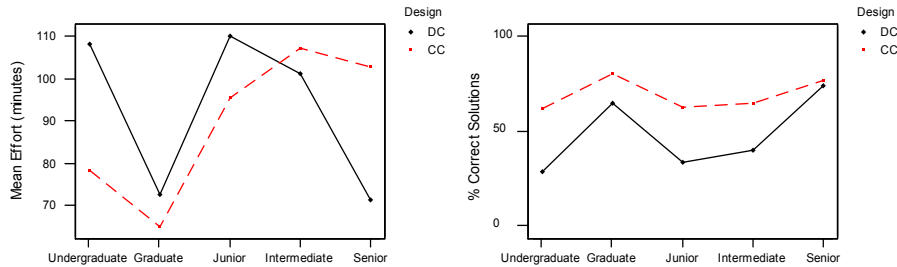
## Typical appearance of an UML Sequence diagram – centralised control style



## Delegated vs. centralised control style

- Centralised control style
  - Easy to comprehend what is going on in a Use case
  - Exceptional situations/variants that need feedback from an actor (via the boundary object) can easily be handled by the control object
  - However: Introduces more dependencies between the control object and the business objects. Potentially less reusable and maintainable models and programming code.
- Delegated control style
  - More "elegant" object oriented design, but...
  - Exaggerated use of delegation makes it difficult to comprehend what is happening – especially if Sequence diagrams are not available ;-)
  - Slightly more complicated to handle exceptional situations/variants that need feedback from an actor, since all communication must go via the control object.

## Results from a controlled experiment(\*)



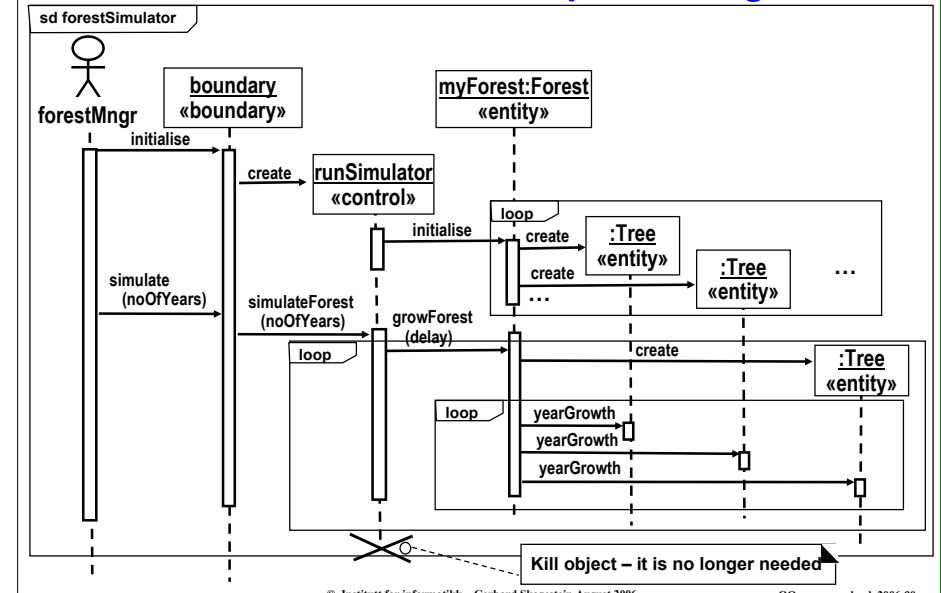
DC = Delegated Control Style  
CC = Centralized Control Style

158 Java-developers were given the task to make changes in a DC or a CC design for the same system.

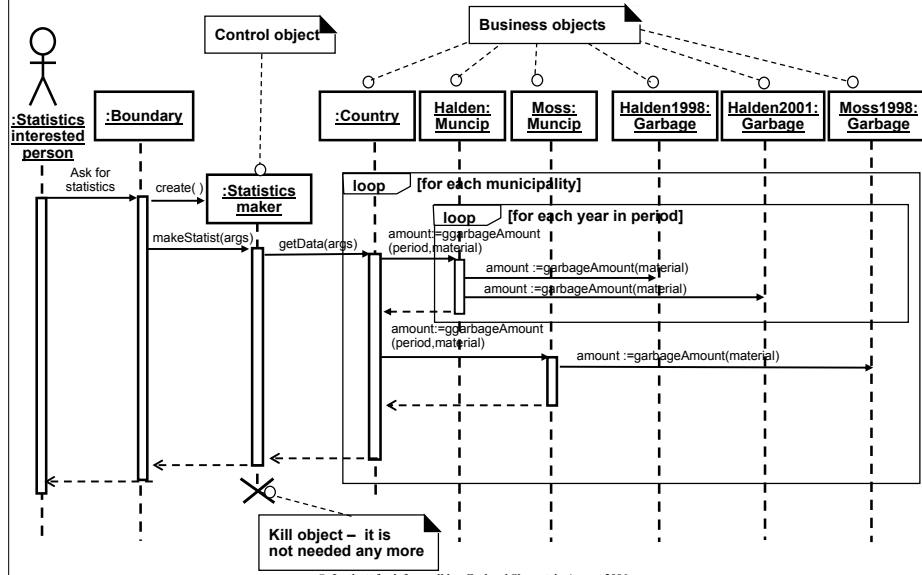
Used time ("Mean Effort") and quality ("% correct solutions") were measured  
Only the experienced designers seem to solve the task better with a DC design

\* Erik Arisholm and Dag Sjøberg, "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software," *IEEE Transactions on Software Engineering*, 2004

## Forest simulator UML Sequence diagram



## Sequence diagram for giving garbage-collection statistics



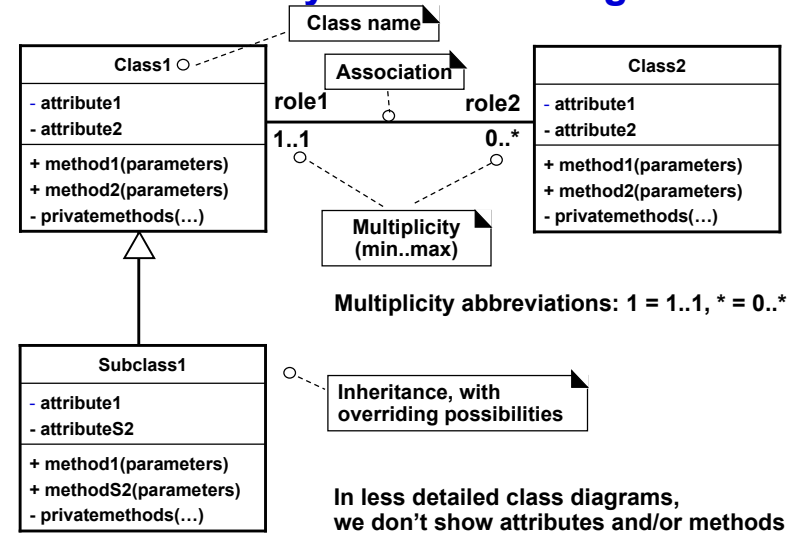
## From Use case to Sequence diagram

- For each Use case, make a sequence diagram for the normal flow of events
- For each variation you may choose to make a new sequence diagram.
  - It is important to make sequence diagrams for variations that have great impact on the design – will the variation demand new objects or new methods?
- UML 2.0 sequence diagrams encompasses constructions for including normal flow and variations in one diagram – the diagram gets more complicated, but everything is in one place, and the MDA-people are happy...

## Method for responsibility driven Object oriented analysis and design

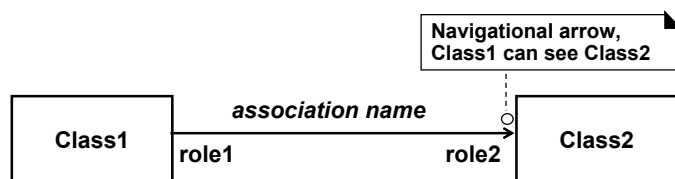
- Analysis of requirements
  1. Identify actors and their goals
  2. Make a high level Use case diagram
  3. Make a textual specification of each Use case with normal flow of events and variations
- Object design
  4. Identify objects and assign responsibilities to them (CRC)
  5. Make Sequence diagrams for important Use cases
  - ➔ 6. Make Class diagram for each Use case
  7. Make an integrated Class diagram for the system

## Anatomy of a Class diagram

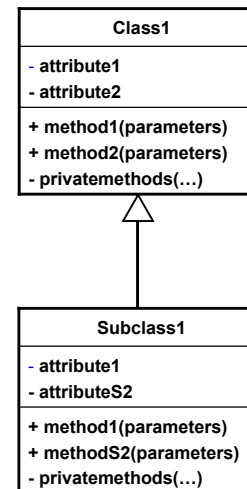


## The associations

- For an object to be able to send a message to another object, it must know about it.
- Therefore, we have associations between the objects.
- The most common associations are those between an object and the object that created it.
- We may want to show navigability by putting arrows on the association line.



## Subtyping and inheritance



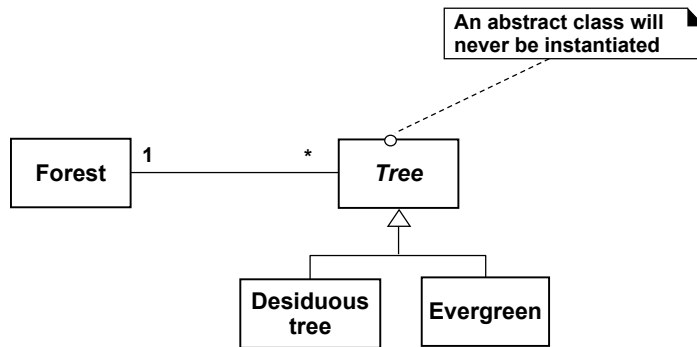
The subclass will inherit all attributes and methods from the superclass.

Subclassing has two main purposes:

- To ease the writing of a class that is a variation of another class  
We may add new attributes and methods, or override (redefine) attributes and methods.
- To create a flexible typing system, in that a subclass can be used wherever its superclass is expected.  
In case of method overriding, the method of the actual object will be used (if not explicitly stated otherwise).

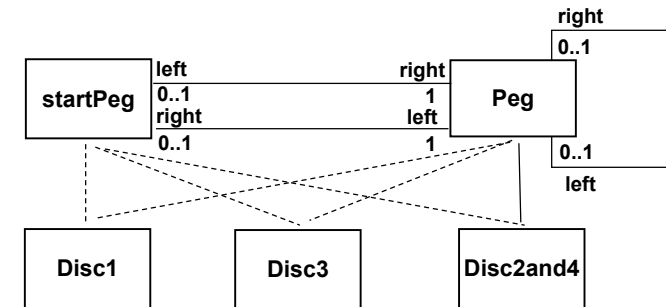
**Polymorphism:** The same method call may bind to different methods and cause different behaviours.

## Example – Class diagram



## The Towers of Hanoi – Class diagram (first attempt)

Usage	Use-Case view	Logical view	Component view	Concurrency view	Deployment view
Diagram Diagram					
Class/object diagram					
Sequence diagram					
Collaboration diagram					
State diagram					
Activity diagram					
Component diagram					
Deployment diagram					

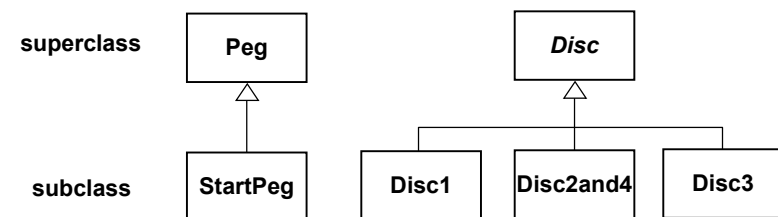


❑ Oh – is this complicated...

## Some observations

- ❑ All pegs have a peg to the left and a peg to the right
- ❑ All pegs sends the message "move" to the top disc
- ❑ All pegs have the method *moveDiscs*
- ❑ All discs sends the message "moveDiscs" to some peg
- ❑ All discs have the method *move*
- ❑ ... and then there are some slight differences

## Inheritance

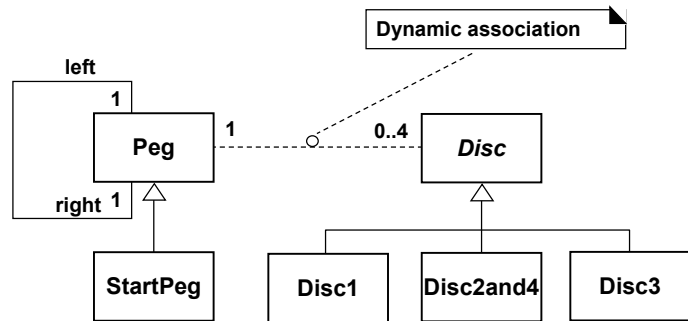


❑ Remember:

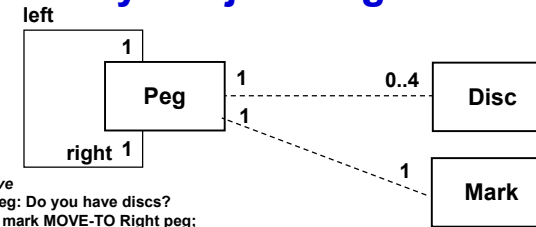
- The subclass will inherit all the status variables and all the methods from the superclass
- In addition, we may in the subclass add status variables and methods, and *override* the inherited methods
- Abstract classes are never instantiated

Usage	Use-Case view	Logical view	Component view	Concurrency view	Deployment view
Diagram type					
Class/object diagram					
Sequence diagram					
Collaboration diagram					
State diagram					
Activity diagram					
Component diagram					
Deployment diagram					

## Tower of Hanoi – Class diagram



## Why not just Peg and Disc?



### Method move

Ask LEFT peg: Do you have discs?

No --> Tell mark MOVE-TO Right peg;

Move to the Left peg;

Tell the "old peg" MOVED •

Yes --> Ask the peg for the size of the top-disc:

Bigger than me -->

Tell mark MOVE-TO Right peg;

Move to the Left peg;

Tell "old peg" MOVED •

Smaller than me -->

Ask Right peg: Do you have discs ?

No --> Tell mark MOVE-TO Left peg;

Move to the Right peg;

Tell the "old peg" MOVED •

Yes --> Ask the peg for the size of the top-disc:

Bigger than me -->

Tell mark Left peg;

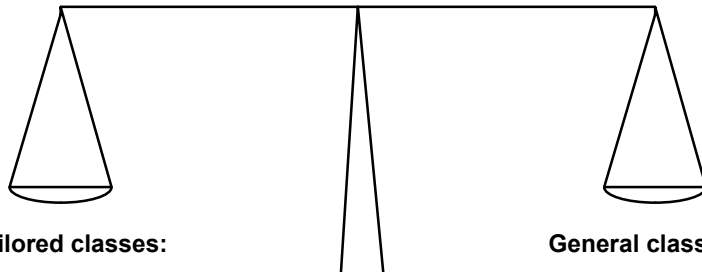
Move to the Right peg;

Tell the "old peg" MOVED •

Smaller than me --> Tell the "old peg" CAN'T MOVE;

and so on...

## Tailored or general classes?



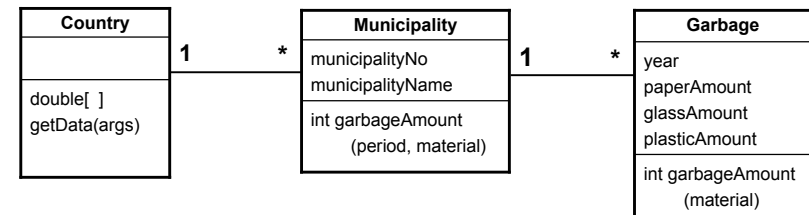
**Tailored classes:**

**Simpler classes**  
**Avoids a lot of if-then-else**  
**Avoids halting problems**

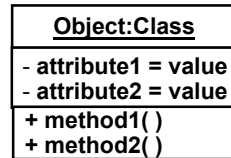
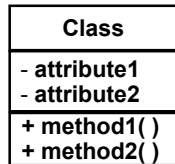
**General classes:**

**Fewer classes**

## Class diagram for the garbage statistics system



## Classes vs. objects

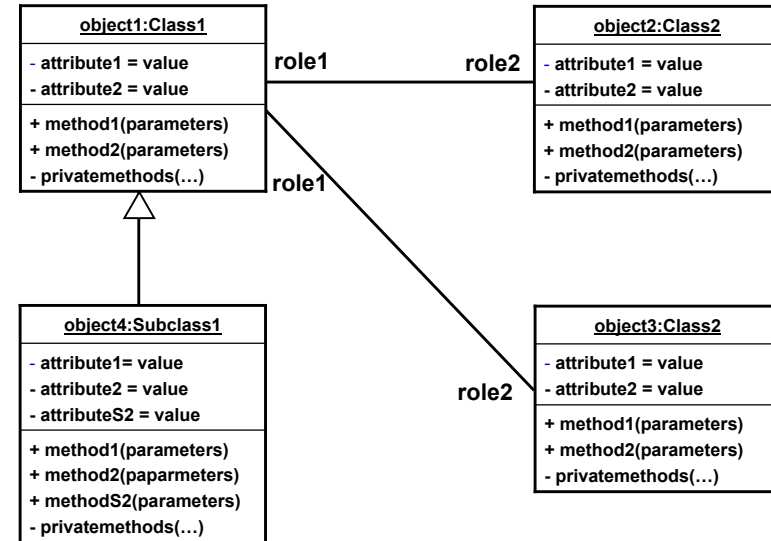


+ means "public"  
 - means "private"  
 # means "protected"

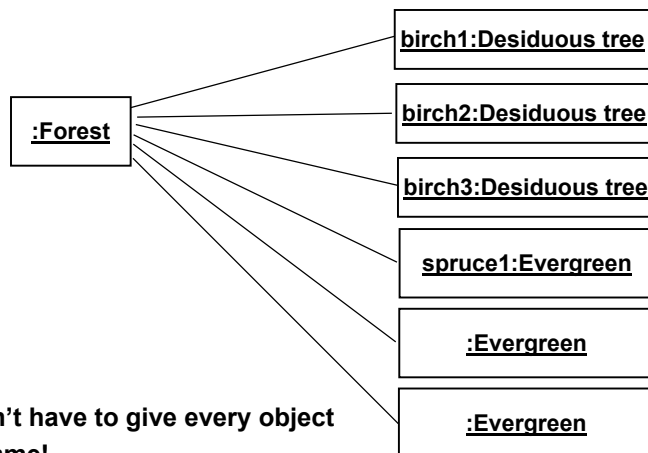
An object is an instantiation of a class



## Anatomy of an Object diagram



## Example – Object diagram

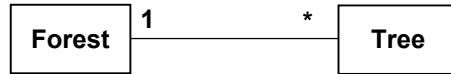


We don't have to give every object a name!

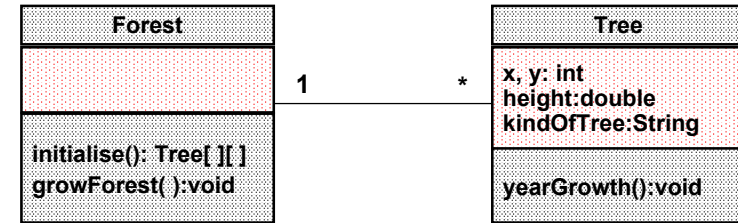
## From Sequence diagram to Class diagram

- Start with the Sequence diagram for normal flow of events:
  - Create classes for all objects.
  - Every unique message in the sequence diagram must be handled by a method in the class for the object.
  - Add the attributes needed by the methods.
  - If an object want to send a message to another object, it must know about it. Make the necessary associations for carrying this knowledge. (... but not for the knowledge acquired dynamically)
  - Note similarities between classes and create superclass/subclass hierarchies if appropriate
- For each variation of each sequence diagram:
  - add new classes, attributes and methods as needed by the messages for the variation.

## Forest simulator UML Class diagram



## Forest simulator UML Class diagram (detailed)



## Method for responsibility driven Object oriented analysis and design

- ❑ Analysis of requirements
  1. Identify actors and their goals
  2. Make a high level Use case diagram
  3. Make a textual specification of each Use case with normal flow of events and variations
- ❑ Object design
  4. Identify objects and assign responsibilities to them (CRC)
  5. Make Sequence diagrams for important Use cases
  6. Make Class diagram for each Use case
  - ➔ 7. Make an integrated Class diagram for the system

## Make an integrated Class diagram for the system

- ❑ Just take all the class diagrams derived from the sequence diagrams, and integrate them!
- ❑ Or – perhaps better – start with the most central class diagram and add the others incrementally!

## Design patterns

- ❑ A design pattern is a named, well documented design principle, a solution to common design problems, independent of any Universe of Discourse or application area
- ❑ Knowledge of patterns
  - avoiding “reinventing the wheel”
  - ease the communications and the discussions in the design process
- ❑ See [http://en.wikipedia.org/wiki/Design\\_pattern\\_%28computer\\_science%29](http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29)
- ❑ The most known book: Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (“the Gang of four”). *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA, Addison-Wesley 1995.

## The layout of a pattern

- ❑ Name (short, but descriptive)
- ❑ Abstract – a brief overview of the pattern
- ❑ Context – in what situations can this problem arise
- ❑ Problem – what is the problem that arises in this context?
- ❑ Solution – including explanatory text, models, CRC-cards...
- ❑ Consequences – good and bad things about what happens if you use the pattern

The general idea of a pattern is taken from the American architect Christopher Alexander’s architectural design patterns (1975 – 1980).

A commonly used format is the one used by the [Gang of Four](#). It contains the following sections:

- ❑ **Pattern Name and Classification:** Every pattern should have a descriptive and unique name that helps in identifying and referring to it. Additionally, the pattern should be classified according to a classification such as the one described earlier. This classification helps in identifying the use of the pattern.
- ❑ **Intent:** This section should describe the goal behind the pattern and the reason for using it. It resembles the problem part of the pattern.
- ❑ **Also Known As:** A pattern could have more than one name. These names should be documented in this section.
- ❑ **Motivation:** This section provides a scenario consisting of a problem and a context in which this pattern can be used. By relating the problem and the context, this section shows when this pattern is used.
- ❑ **Applicability:** This section includes situations in which this pattern is usable. It represents the context part of the pattern.
- ❑ **Structure:** A graphical representation of the pattern. [Class diagrams](#) and [Interaction diagrams](#) can be used for this purpose.
- ❑ **Participants:** A listing of the classes and objects used in this pattern and their roles in the design.
- ❑ **Collaboration:** Describes how classes and objects used in the pattern interact with each other.
- ❑ **Consequences:** This section describes the results, side effects, and trade offs caused by using this pattern.
- ❑ **Implementation:** This section describes the implementation of the pattern, and represents the solution part of the pattern. It provides the techniques used in implementing this pattern, and suggests ways for this implementation.
- ❑ **Sample Code:** An illustration of how this pattern can be used in a programming language
- ❑ **Known Uses:** This section includes examples of real usages of this pattern.
- ❑ **Related Patterns:** This section includes other patterns that have some relation with this pattern, so that they can be used along with this pattern, or instead of this pattern. It also includes the differences this pattern has with similar patterns. (From [http://en.wikipedia.org/wiki/Design\\_pattern\\_%28computer\\_science%29](http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29))

## The GRASP patterns – in short

GRASP: General Responsibility Assignment Software Patterns

- ❑ **Information Expert**  
Let the object that has the knowledge (i.e. the data) also handle the data. (But... what about the other way around?)
- ❑ **Creator**  
Whenever a new object should be created, let the object that has to know about the new object, create it.
- ❑ **High cohesion**  
Don’t let an object have responsibility for a lot of different, unrelated things.
- ❑ **Low coupling**  
Let each object collaborate with relatively few other objects. However, do not go to the extreme, so that the system consist of only one object!

continued...

## The GRASP patterns – in short (cont.)

- ❑ **Controller**  
Let each Use case has a control object that control the execution of the Use case. If the Use cases are small and few, the whole system may have a single control object – this is however contradictory to the principle of high cohesion.
  - ❑ **Polymorphism**  
Create specialised objects by means of subclassing instead of general objects, which repeatedly must ask themselves about what they are to regulate their behaviour.
  - ❑ **Pure fabrication**  
Sometimes it is necessary to invent an artificial (fabricated) object with no counterpart in the real world in order to get a neat design.
  - ❑ **Indirection**  
Sometimes it is convenient to decouple objects by means of an intermediate object (Pure fabrication to achieve Low coupling).
- continue...

## The GRASP patterns – in short (cont.)

- ❑ **Protected variations**  
Try to hide objects that are in an unstable design situation behind a stable interface.
  - ❑ **A special case:**  
**Don't talk to strangers (Law of Demeter)**  
An object should only send messages to
    - itself
    - an object that is directly accessible through an association
    - a parameter of the method
    - an object that is created within the method
- If you violate this rule, you make your design too dependent of the overall structure, which is likely to change.

## More specific patterns Classification

Design patterns can be classified in terms of the underlying problem they solve. Examples of problem-based pattern classifications include:

- ❑ **Fundamental patterns** (basic stuff, generally applicable)
- ❑ **Creational patterns** (object creation mechanisms)
- ❑ **Structural patterns** (how to realise relationships)
- ❑ **Behavioral patterns** (how to realise behaviour)
- ❑ **Concurrency patterns** (how to realise parallell execution)
- ❑ **Architectural patterns** (higher level design, like the Model-View-Controller paradigm)

## Some creational patterns

- ❑ Creational design patterns deal with **object creation** mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.
- ❑ Some examples of creational design patterns include:
  - ❑ **Abstract factory pattern**: centralize decision of what **factory** to instantiate
  - ❑ **Factory method pattern**: centralize creation of an object of a specific type choosing one of several implementations
  - ❑ **Anonymous subroutine objects pattern**
  - ❑ **Builder pattern**
  - ❑ **Lazy initialization pattern**
  - ❑ **Prototype pattern**
  - ❑ **Singleton pattern**

## Singleton pattern

- ❑ The singleton **design pattern** is used to restrict instantiation of a class to one (or a few) **objects**. This is useful when exactly one object is needed to coordinate actions across the system. Before designing a class as a singleton, it is wise to consider whether it would be enough to design a normal class and just use one instance.
- ❑ The singleton pattern is implemented by creating a **class** with a method that creates a new instance of the object if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the **constructor** is made either private or protected. Note the distinction between a simple static instance of a class and a singleton.

## Singleton pattern in Java

See Barnes & Kölling page 407

```
class Parser {  
  
    private static Parser instance = new Parser();  
  
    public static Parser getInstance() {  
  
        return instance;  
  
    }  
  
    private Parser() {  
  
        ...  
  
    }  
  
}
```

## Some structural patterns

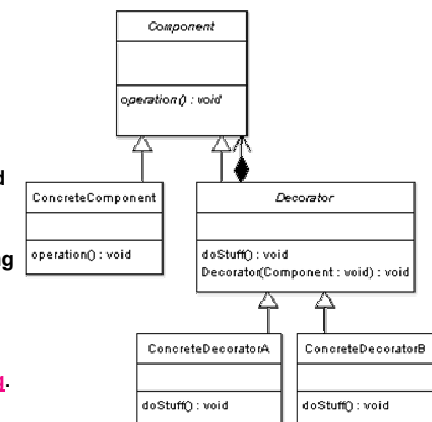
Structural Design Patterns ease the design by identifying a simple way to realize relationships between entities.

Examples of Structural Patterns include:

- ❑ **Adapter pattern**: 'adapts' one interface for a class into one that a client expects
- ❑ **Aggregate pattern**: a version of the **Composite pattern** with methods for aggregation of children
- ❑ **Bridge pattern**: decouple an abstraction from its implementation so that the two can vary independently
- ❑ **Composite pattern**: a tree structure of objects where every object has the same interface
- ❑ **Container pattern**: create objects for the sole purpose of holding other objects and managing them
- ❑ **Decorator pattern**: add additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes
- ❑ **Extensibility pattern**: aka. Framework - hide complex code behind a simple interface
- ❑ **Facade pattern**: create a simplified interface of an existing interface to ease usage for common tasks
- ❑ **Flyweight pattern**: a high quantity of objects share a common properties object to save space
- ❑ **Proxy pattern**: a class functioning as an interface to another thing
- ❑ **Pipes and filters**: a chain of processes where the output of each process is the input of the next
- ❑ **Private class data pattern**: restrict accessor/mutator access

## Decorator pattern

- ❑ The decorator pattern is a **design pattern** that allows new/additional behavior to be added to an existing method of an object dynamically.
- ❑ The decorator pattern works by **wrapping** the new "decorator" object around the original object, which is typically achieved by passing the original object as a parameter to the constructor of the decorator, with the decorator implementing the new functionality. The interface of the original object needs to be maintained by the decorator.
- ❑ Decorators are alternatives to **subclassing**. Subclassing adds behaviour at **compile time** whereas decorators provide a new behaviour at **runtime**.



## Façade pattern

A façade is an object that provides a simplified interface to a larger body of code, such as a [class library](#). A façade can:

- ❑ make a [software library](#) easier to use and understand, since the façade has convenient methods for common tasks;
- ❑ make code that uses the library more readable, for the same reason;
- ❑ reduce dependencies of outside code on the inner workings of a library, since most code uses the façade, thus allowing more flexibility in developing the system;
- ❑ wrap a poorly designed collection of APIs with a single well-designed API.
- ❑ Façades are very common in [object-oriented design](#). For example, the Java standard library contains dozens of classes for parsing font files and rendering text into geometric outlines and ultimately into pixels.

## Some behavioral patterns

Behavioral design patterns identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Examples of this type of design pattern include:

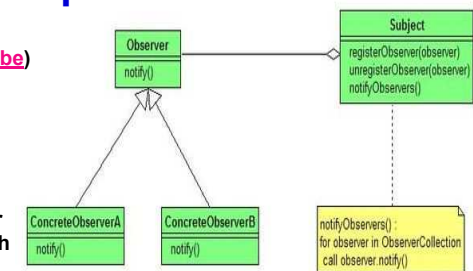
- ❑ [Chain of responsibility pattern](#)
- ❑ [Command pattern](#)
- ❑ [Event listener](#)
- ❑ [Interpreter pattern](#)
- ❑ [Iterator pattern](#)
- ❑ [Mediator pattern](#)
- ❑ [Memento pattern](#)
- ❑ [Observer pattern](#)
- ❑ [State pattern](#)
- ❑ [Strategy pattern](#)
- ❑ [Template method pattern](#)
- ❑ [Visitor pattern](#)
- ❑ [Single-serving visitor pattern](#)
- ❑ [Hierarchical visitor pattern](#)

## Memento pattern

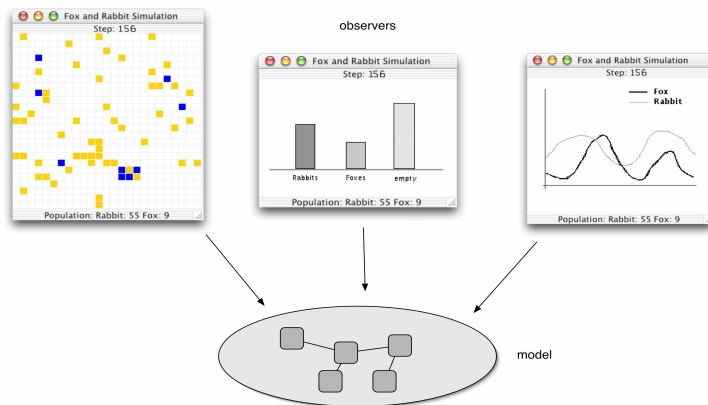
- ❑ The memento pattern provides the ability to restore an object to its previous state (undo by rollback).
- ❑ The memento pattern is used by two objects: the *originator* and a *caretaker*. The originator is some object that has an internal state. The caretaker is going to do something to the originator, but wants to be able to undo the change. The caretaker first asks the originator for a memento object. Then it does whatever operation (or sequence of operations) it was going to do. To rollback to the state before the operations, it returns the memento object to the originator. The memento object itself is an [opaque object](#) (one which the caretaker can not, or should not, change). When using this pattern, care should be taken if the originator may change other objects or resources - the memento pattern operates on a single object.
- ❑ Classic examples of the memento pattern include the seed of a [pseudorandom number generator](#) and the state in a [finite state machine](#).

## Observer pattern

- ❑ The observer pattern (sometimes known as [publish/subscribe](#)) is a [design pattern](#) used in computer programming to observe the state of an [object](#) in a [program](#).
- ❑ The essence of this pattern is that one or more objects (called observers or listeners) are registered (or register themselves) to *observe* an [event](#) which may be raised by the observed object (the subject). (The object which may raise an event generally maintains a collection of the observers.)  
...the Hollywood principle: “Don’t call us – we will call you”
- ❑ When the event is raised each observer receives a [callback](#) (called *notify* on the diagram) . The notify function may also be passed some parameters (generally information about the event that is occurring) which can be used by the observer.
- ❑ Each concrete observer implements the notify function and as a consequence defines its own behavior when the notification occurs.

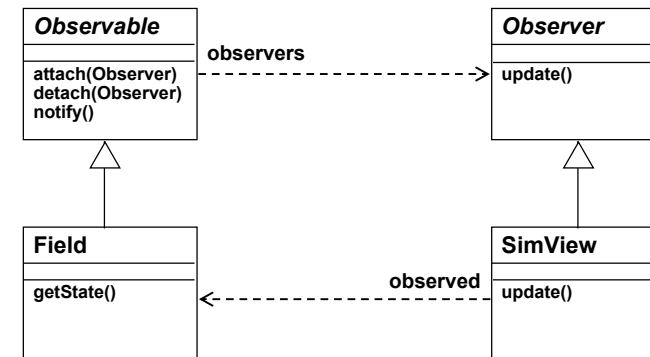


## Observers



## Structure of the Observer pattern

See Barnes&Kölling page 409



## Visitor pattern

- The visitor **design pattern** is a way of separating an **algorithm** from an object structure. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures.
- The idea is to use a structure of element classes, each of which has an accept method that takes a visitor object as an argument. The visitor is an **interface** that has a visit() method for each element class. The accept() method of an element class calls back the visit() method for its class. Separate concrete visitor classes can then be written that perform some particular operations.

## Visitor pattern – example

```
interface Visitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}

interface Visitable {
    public void accept(Visitor visitor);
}

class Wheel implements Visitable {
    private String name;
    Wheel(String name) {
        this.name = name;
    }
    String getName() {
        return this.name;
    }
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Engine implements Visitable{
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Body implements Visitable{
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Car implements Visitable {
    private Engine engine = new Engine();
    private Body body = new Body();
    private Wheel[] wheels
        = { new Wheel("front left"), new Wheel("front
            right"),
            new Wheel("back left"), new Wheel("back
            right") };
    public void accept(Visitor visitor) {
        visitor.visit(this);
        engine.accept( visitor );
        body.accept( visitor );
        for(int i=0; i<wheels.length; ++i)
            wheels[i].accept( visitor );
    }
}

class PrintVisitor implements Visitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting
            "+wheel.getName()
                +" wheel");
    }
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
    ...
}

Car car = new Car();
Visitor visitor = new PrintVisitor();
car.accept(visitor);
}
```

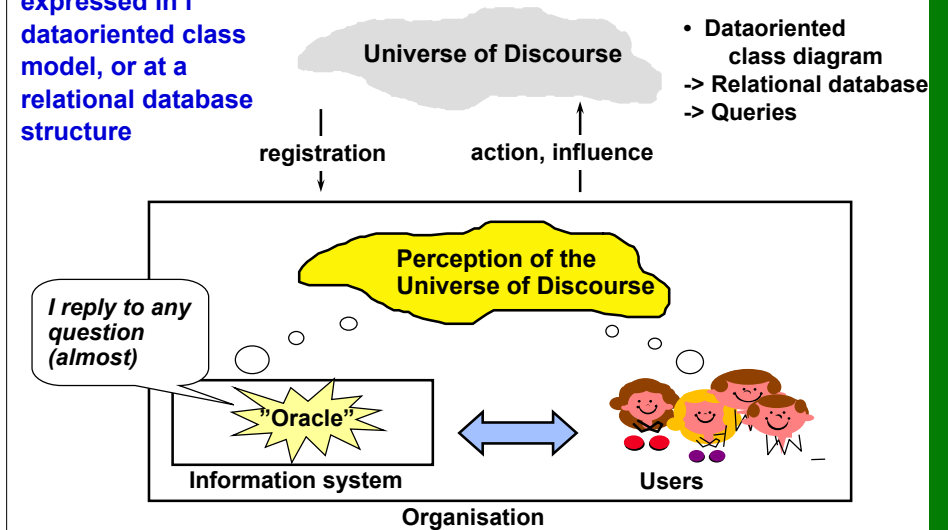
# And then we have the anti-patterns...

...describing what you should *not* do...

See <http://en.wikipedia.org/wiki/Anti-pattern>

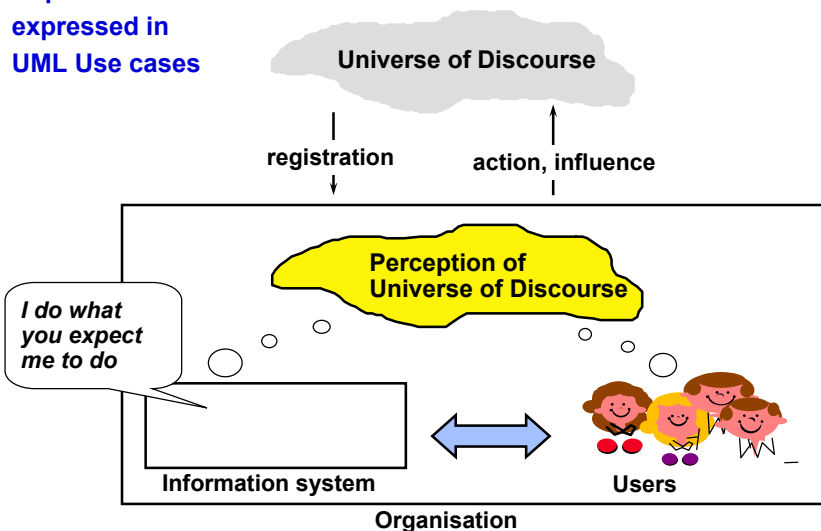
What the Oracle should know is expressed in i dataoriented class model, or at a relational database structure

# From the kernel and outwards

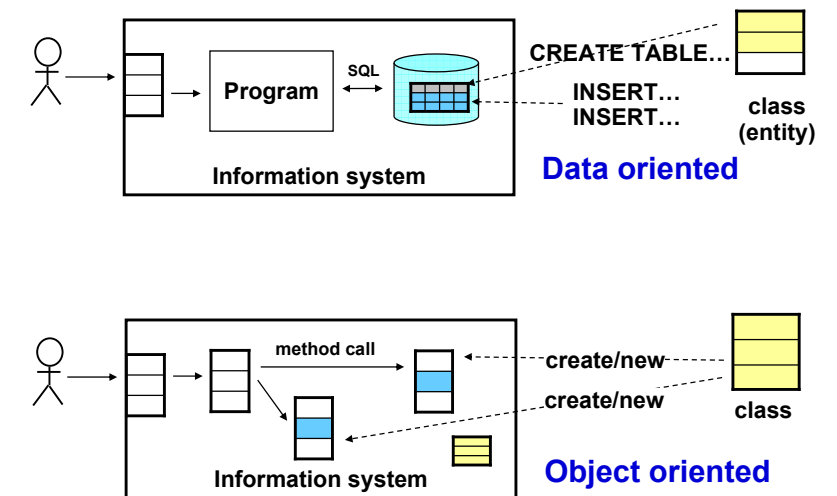


The functional requirements are expressed in UML Use cases

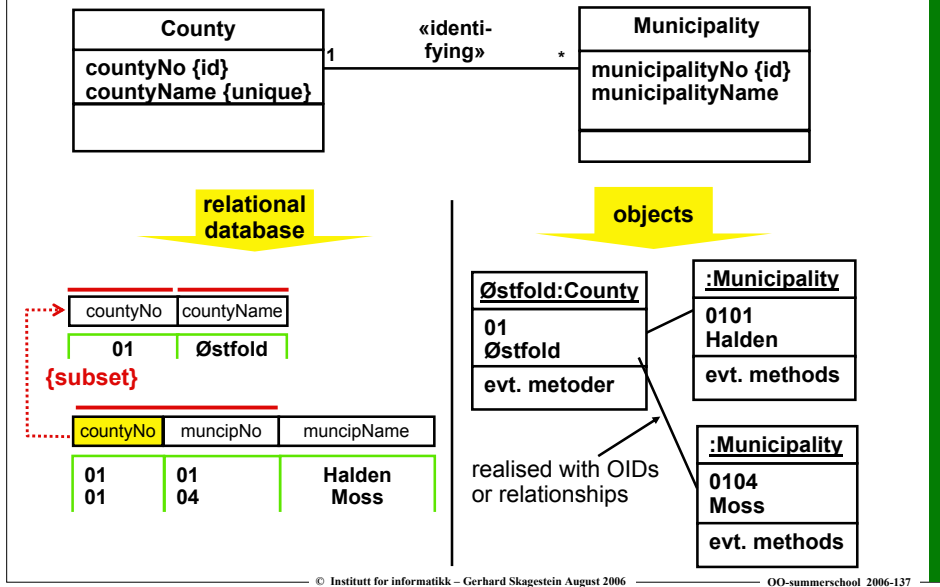
# From the peel and inwards



# Data oriented vs. object oriented design



## Realising a class diagram



## Development directions and designs

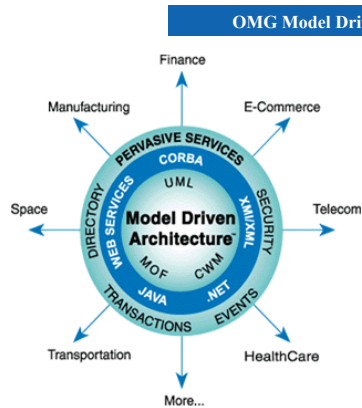
	From the kernel and outwards	From the peel and inwards
Data oriented architecture	😊	?
Object oriented architecture	?	😊

Are the yellow squares of interest?



## The MDA-vision – do it automatically 😊

<http://www.omg.org/mda/>



### How Systems Will Be Built

MDA<sup>®</sup> provides an open, vendor-neutral approach to the challenge of business and technology change. Based firmly upon OMG's established standards\*, MDA aims to separate business or application logic from underlying platform technology. Platform-independent applications built using MDA and associated standards can be realized on a range of open and proprietary platforms, including CORBA<sup>®</sup>, J2EE, .NET, and Web Services or other Web-based platforms. Fully-specified platform-independent models (including behavior) can enable intellectual property to move away from technology-specific code, helping to insulate business applications from technology evolution, and further enable interoperability. In addition, business applications, freed from technology specifics, will be more able to evolve at the different pace of business evolution.

\* Key standards that make up the MDA suite of standards include Unified Modeling Language (UML); Meta-Object Facility (MOF); XML Meta-Data Interchange (XMI); and Common Warehouse Meta-model (CWM).