INTRODUCTION TO SOFTWARE ENGINEERING

Software Engineering

Chapter 1

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

etsinf

Goals

- Introduce Software Engineering and explain its importance for software development
- Answer main questions related to Software Engineering
- Introduce Software Process

Contents

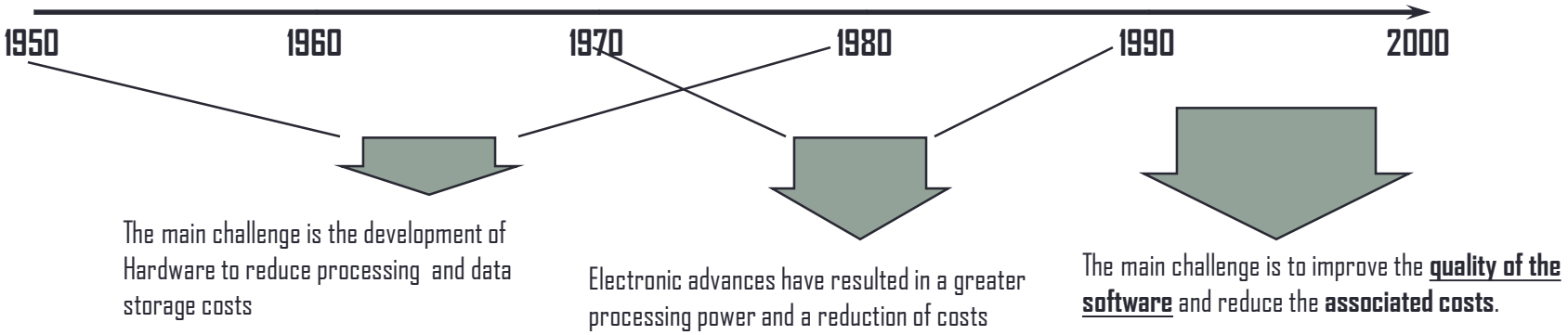
References

- Sommerville, I. Software Engineering. Addison-Wesley, 2008.
- Pressman, R., Software Engineering: A Practitioner's Approach. McGraw-Hill, 2005.
- Weitzenfeld, A., Object Oriented Software Engineering with UML, Java and Internet. Thomson, 2005

INTRODUCTION

Software makes a difference

In the latest decades software has overcome hardware as a critical factor for success

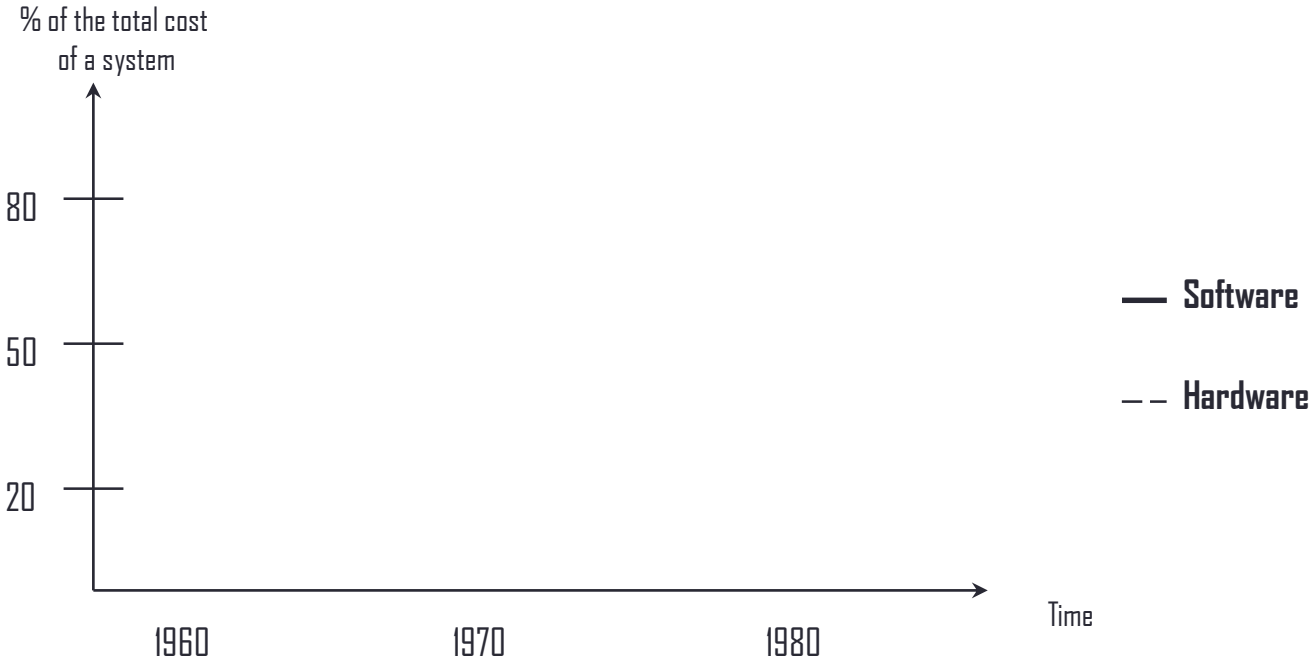


Software makes a difference

- In the last decade, as a result of the sucess of the Web as a platform and the use of mobile devices, the software industry has experienced a revolution
  - New languages
  - New HTML versions
  - New devices
  - New development methods!!

Software is more expensive…

- Evolution of the total cost of a system in terms of the percentage invested in software and in hardware



…and not just money!

- http://www5.in.tum.de/persons/huckle/bugse.html
- http://www.pcmag.com/article2/0,1759,1636333,00.asp
- http://www.microsiervos.com/archivo/ordenadores/10-peores-bugs.html
- http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all
- http://www.taringa.net/posts/info/3469982/Los-20-desastres-mas-famosos-de-la-historia-del-software.html

- http://catless.ncl.ac.uk/Risks

THE SOFTWARE

- ✓ Characteristics
- ✓ Software crisis
- ✓ Quality Software
- ✓ Industry Problems

What is software?

- Instructions that provide an expected function and behavior when executed
- Data structures that allow programs to adequately manipulate information
- Documents that describe the operation and use of programs

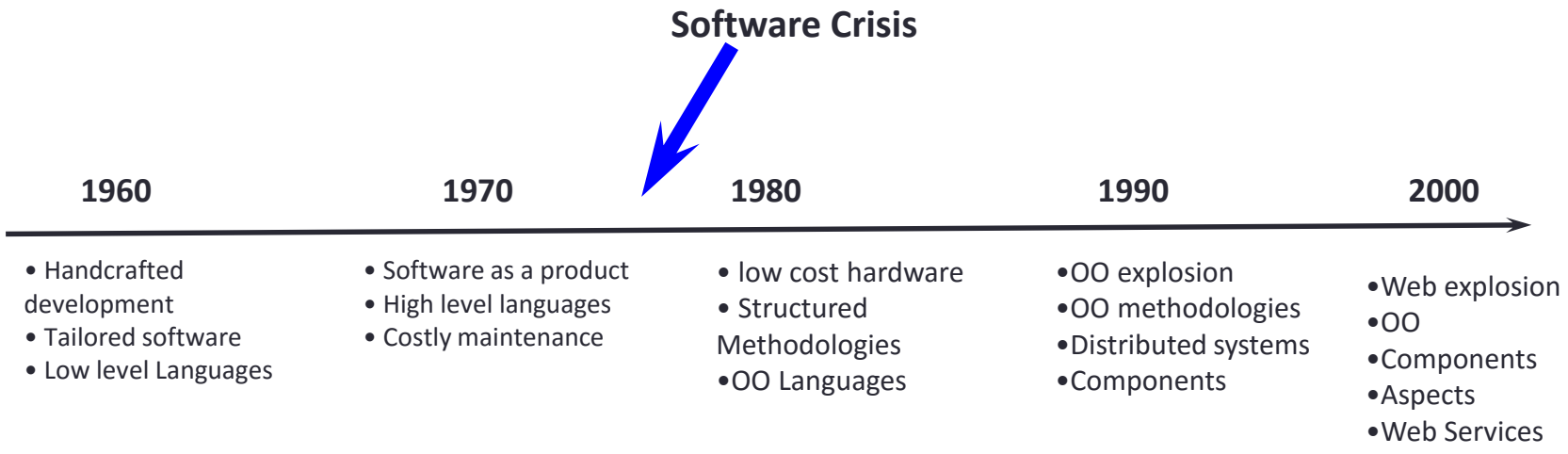
Characteristics of software

Software is a logical element:

- It is developed, not manufactured in the classical sense
- It doesn’t break down, it deteriorates as a result of changes
- Most of it is tailored for specific purposes instead of being assembled from existing components

Software Evolution

- The **context** in which software is developed is strongly related to the development of computing systems



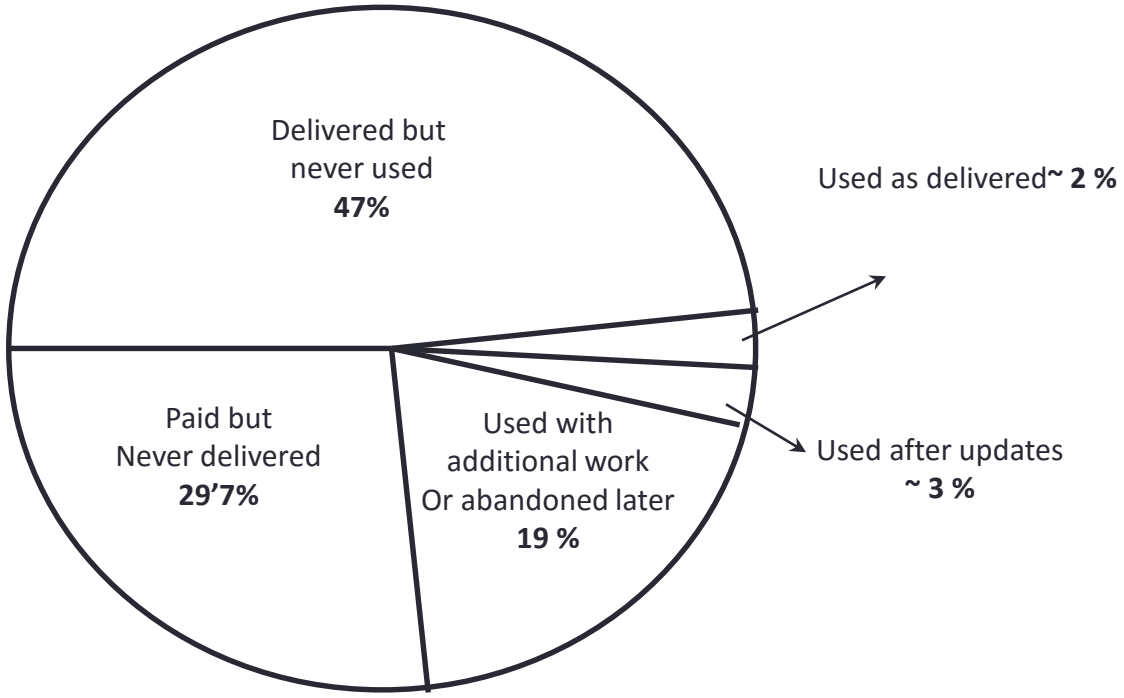
Software Crisis

- Costs are higher than planned
- Delivery dates delays
- Bad Performance
- Impossible maintenance
- High cost of Updates
- Unreliable products

Low quality Software!

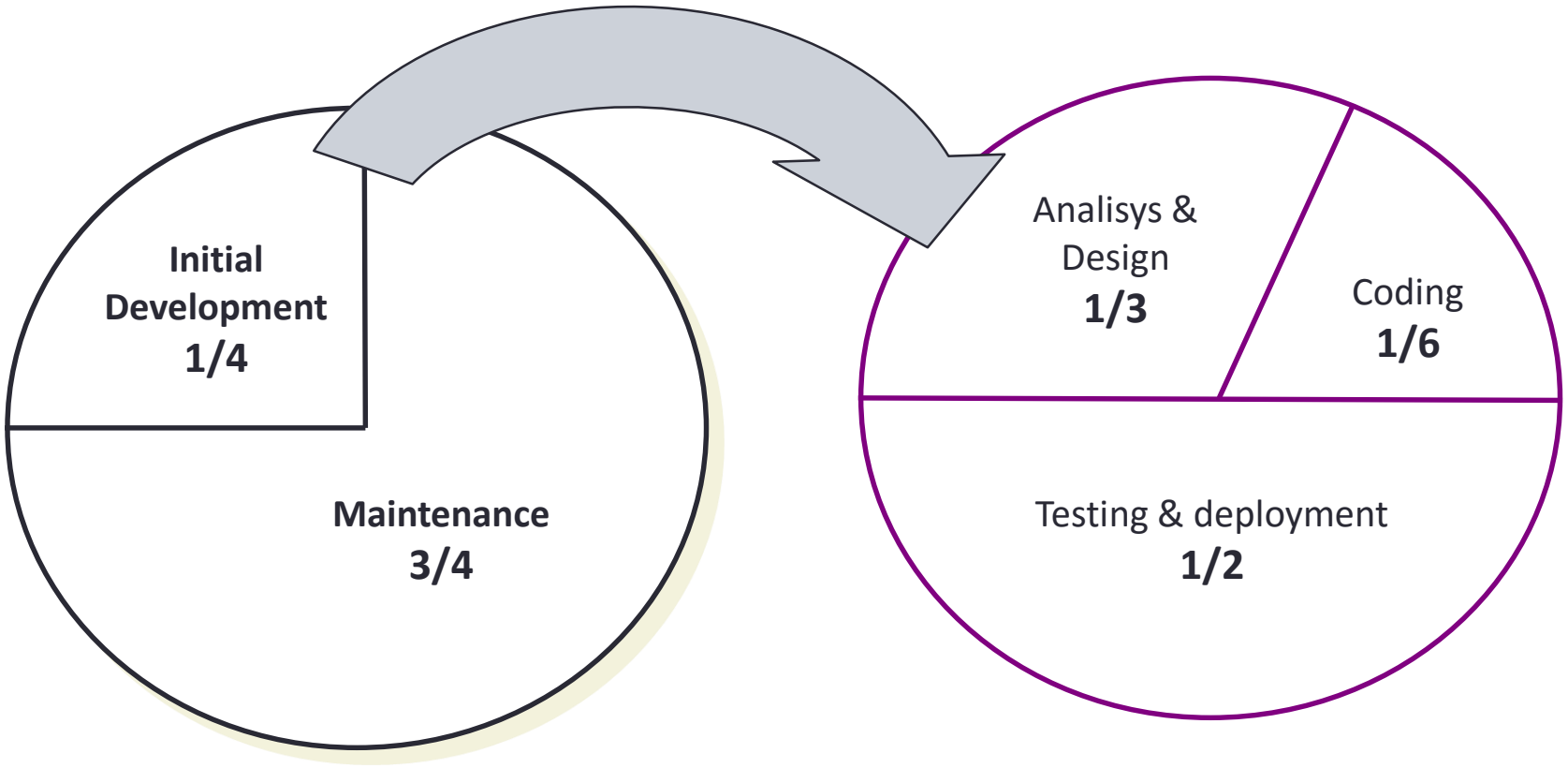
Software Crisis

- **Investment** in Software development
  - Year 1979 (Total: $ 6.8 million)



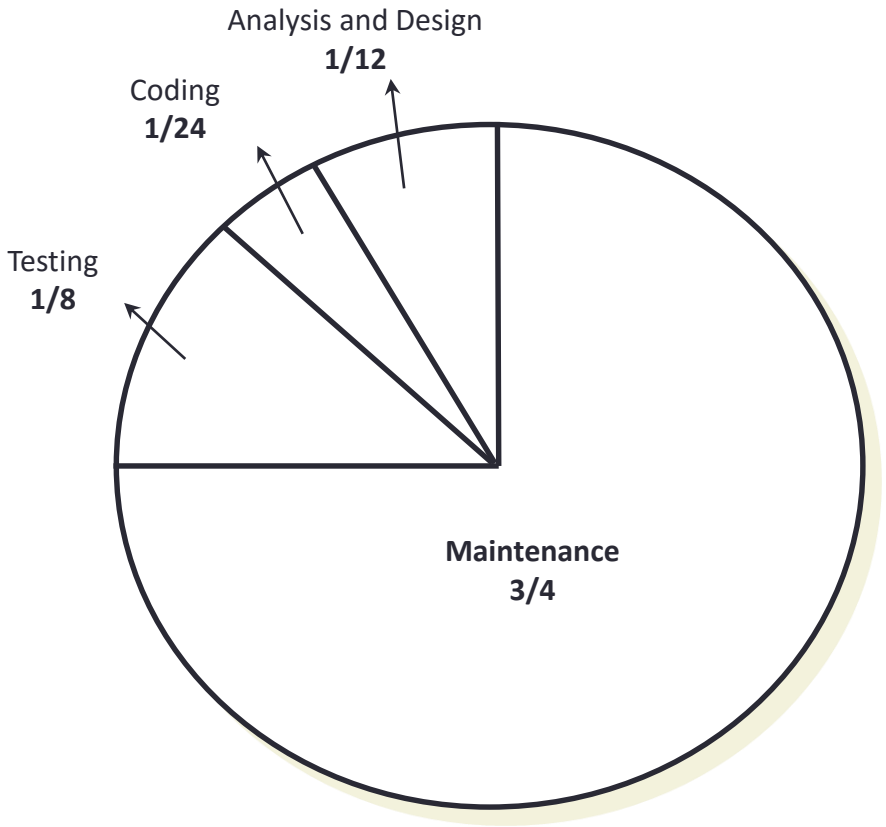
Software Crisis

- **Investment** in software development (by development phase):



Software Crisis

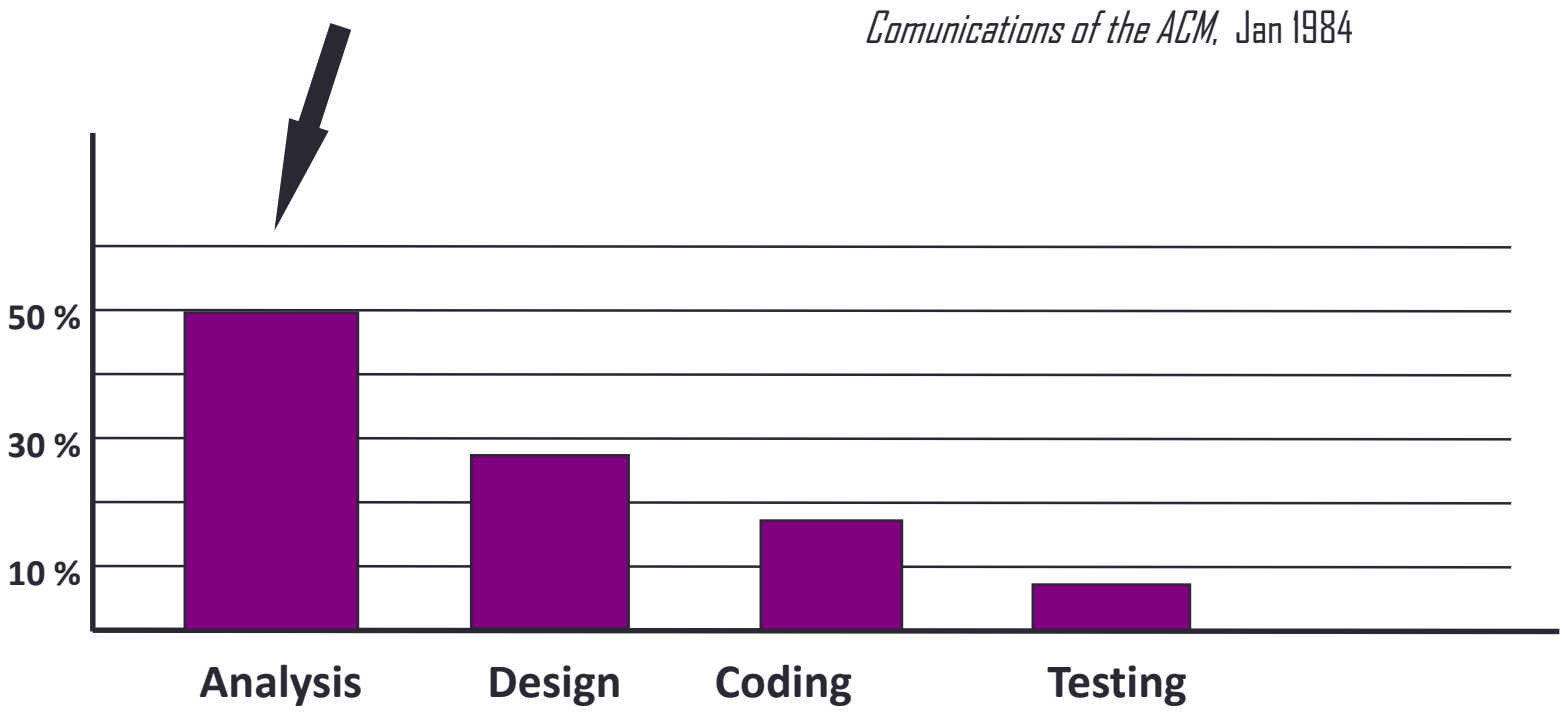
- Summary of **investment**:



- **Analysis & Design** ⟹ **8 %**
- **Coding** ⟹ **4 %**
- **Testing and Maintenance** ⟹ **88 %**

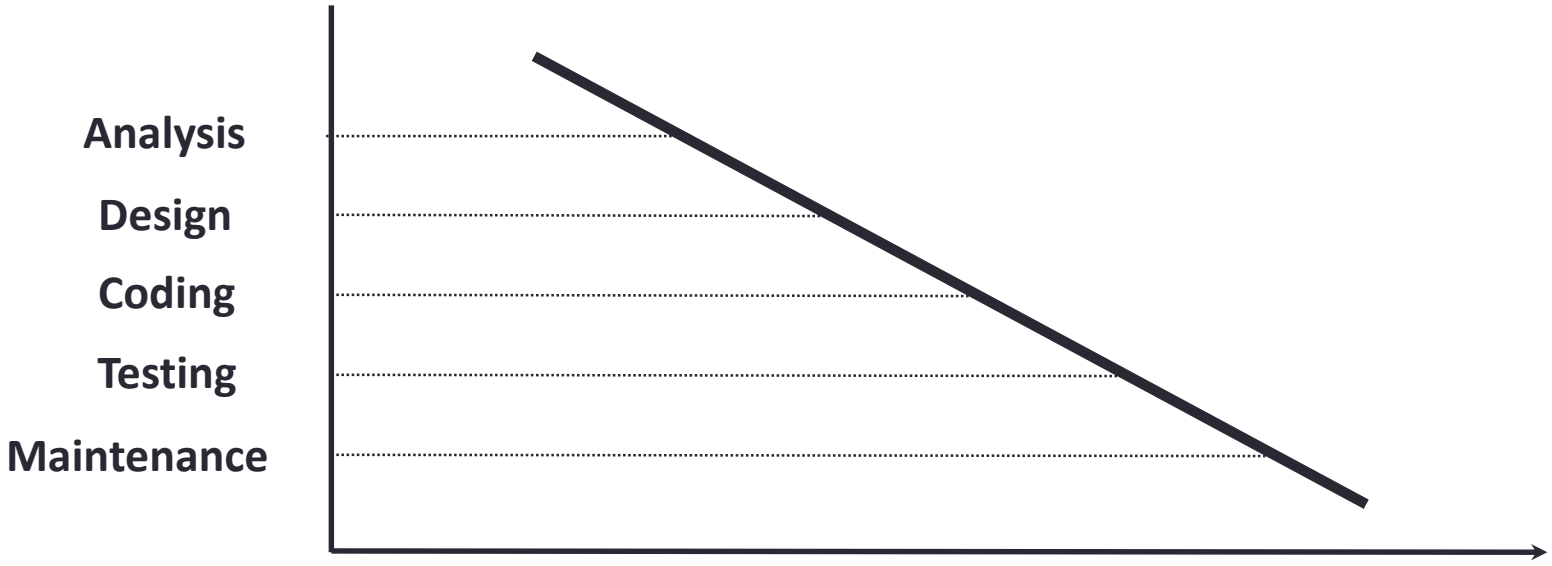
Software crisis

- **Errors** during software development (by development phase) :



Software Crisis

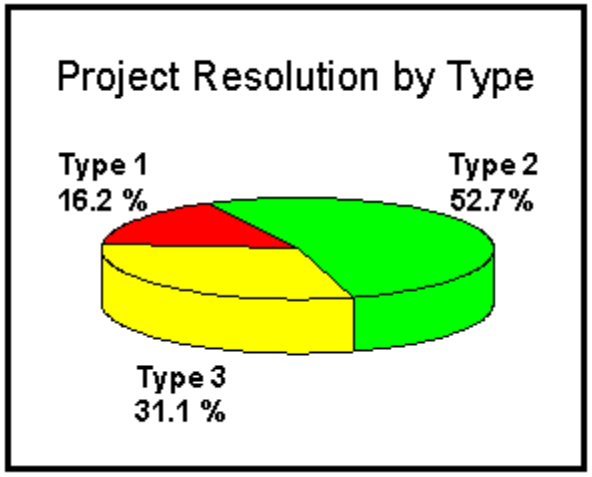
- **Cost** of fixing errors:



Software crisis

Investment in software development.

- Year 1994 (Total: $ 250 Kmillions/year -- 175.000 projects)



- ***Type 1 (Success).*** The project is Ended on time and within butget with all initially planned features and functionalities.

- ***Type 2 (Updates).*** The project is Is ended late and at a higher cost It has less features and functionalities than were specified.

- ***Type 3 (Cancelled).*** The project is cancelled during its development.

http://www.standishgroup.com/sample_research/chaos_1994_1.php

Software crisis

CHAOS Report…

RESOLUTION

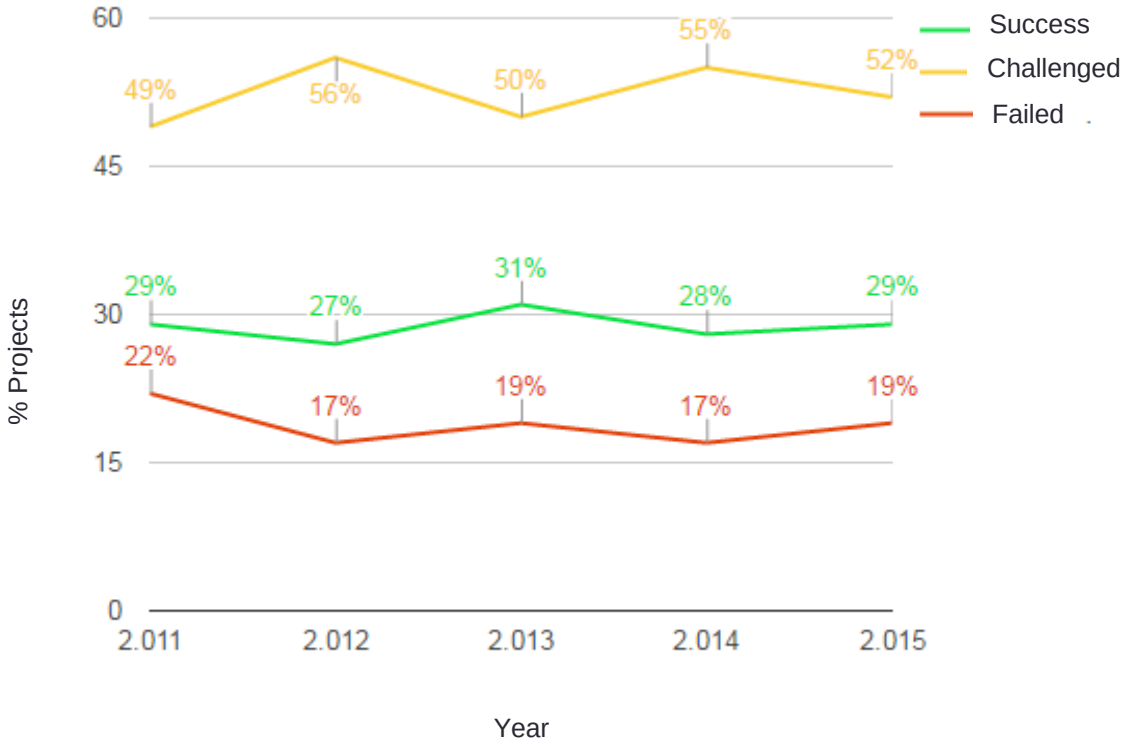
	2004	2006	2008	2010	2012
Successful	29%	35%	32%	37%	39%
Failed	18%	19%	24%	21%	18%
Challenged	53%	46%	44%	42%	43%

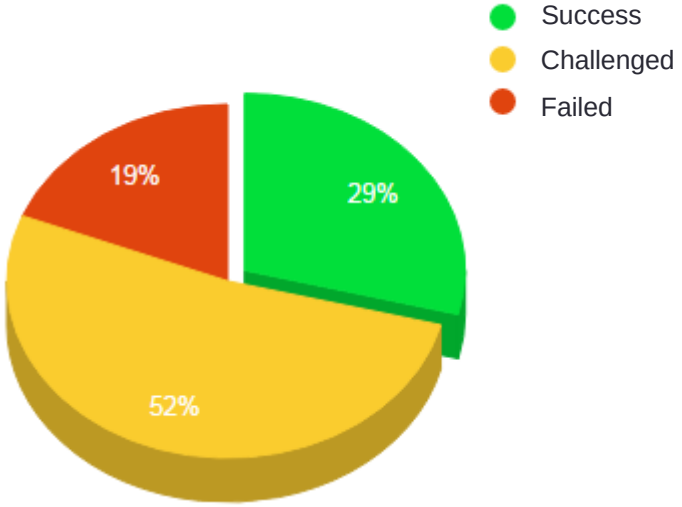
Project resolution results from CHAOS research for years 2004 to 2012.

http://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf

Software crisis

CHAOS Report 2015



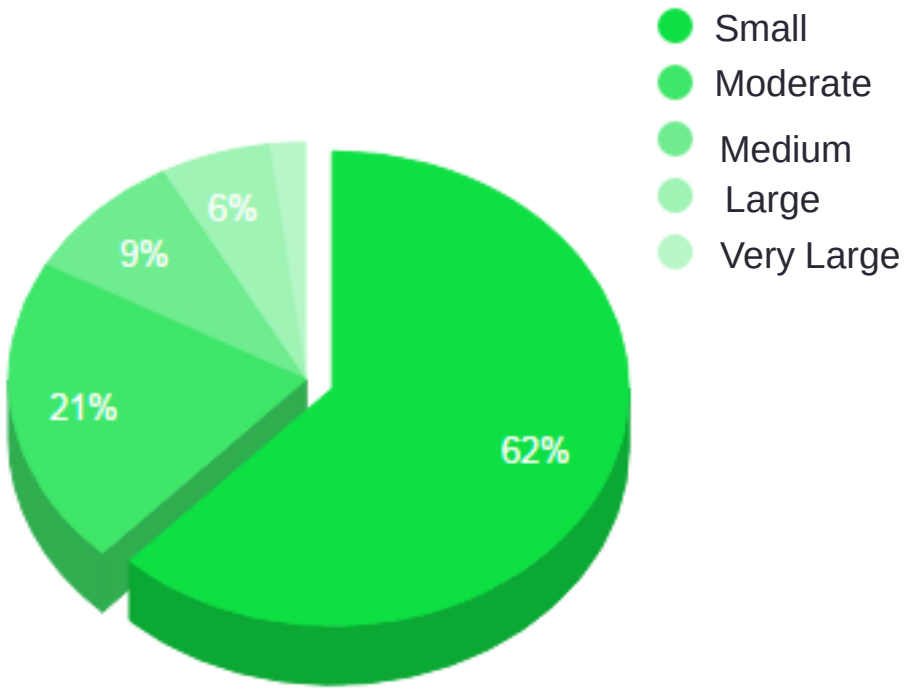


Software crisis

CHAOS Report 2015

Is the size of a software project relevant?

% over successful projects
2011-2015



Quality software

- The end goal is to produce high quality software

What is high quality software?

Agreement with:

- Functional and non functional requirements
- The documented development standards
- The expected features exhibited by any software developed profesionally

Quality factors

- The classification of the quality factors of software takes into account three importact aspects of a software product

  1. Its operational features
  2. Its capability to support updates
  3. Its adaptability to new environments

- These must be measured direct or indirectly during the whole development process

Quality factors

Correctness Does it do what I want?

Reliability Is it reliable all the time?

Efficiency Will it run in the HW platform efficiently?

Integrity Is it safe?

Usability Is it designed to be used?

Operational features

Maintainability May it be corrected?

Flexibility May it be easily changed?

Testability May it be verified?

Supporting Updates

Reusability Is the software reusable?

Portability May it be used in another HW or OS?

Interoperability May it interact with another system?

Adaptability to New environments

Software Industry Problems

- Products are low quality
- High maintenance and development costs
- Delivery Delays

Reasons:

- ✓*Little investment and effort in the analysis and specification phases*
- ✓*Use of informal and inadequate models*
- ✓*Non physical nature of programming*
- ✓*Poor theorerical foundations*
- ✓*Products already in the market make it difficult innovation*
- ✓*High levels of hand crafting*
- ✓*Groupwork*
- ✓*Comumunication with users*
- ✓*Project management by non computing engineers*

Solutions

- Education:
  - Formal methods (executable formal languages: logic + algebra)
  - New development methods and new lifecycles
- Difusion of technological advancements
  - New programming paradigms
  - Architectures, protocols, computation models
- Tools investment
  - Modern development environments
  - Documentation generation engines

SOFTWARE ENGINEERING

- ✓ Definitions
- ✓ The Software Developmet Process

Definitions

- B. Boehm:
  - *"SE is the practical* application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them
- R. Pressman:
  - *"SE is a discipline that integrates methods, tools and procedures for the development of Software"*.
- A. Davis:
  - *"SE is the application of scientific principles for: (1) the transformation of a problem into a SW solution and (2) its maintenance during all its life".*
- I. Sommerville:
  - *"SE is an engineering approach covering all aspects of software production"*

A little bit of history

- http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/Web/History/

Summarizing…

- SE is more than just coding
- The SE process starts well before writing lines of code and it continues after the first version of the product has been completed
- Key tasks are planning and rigurous control of software projects

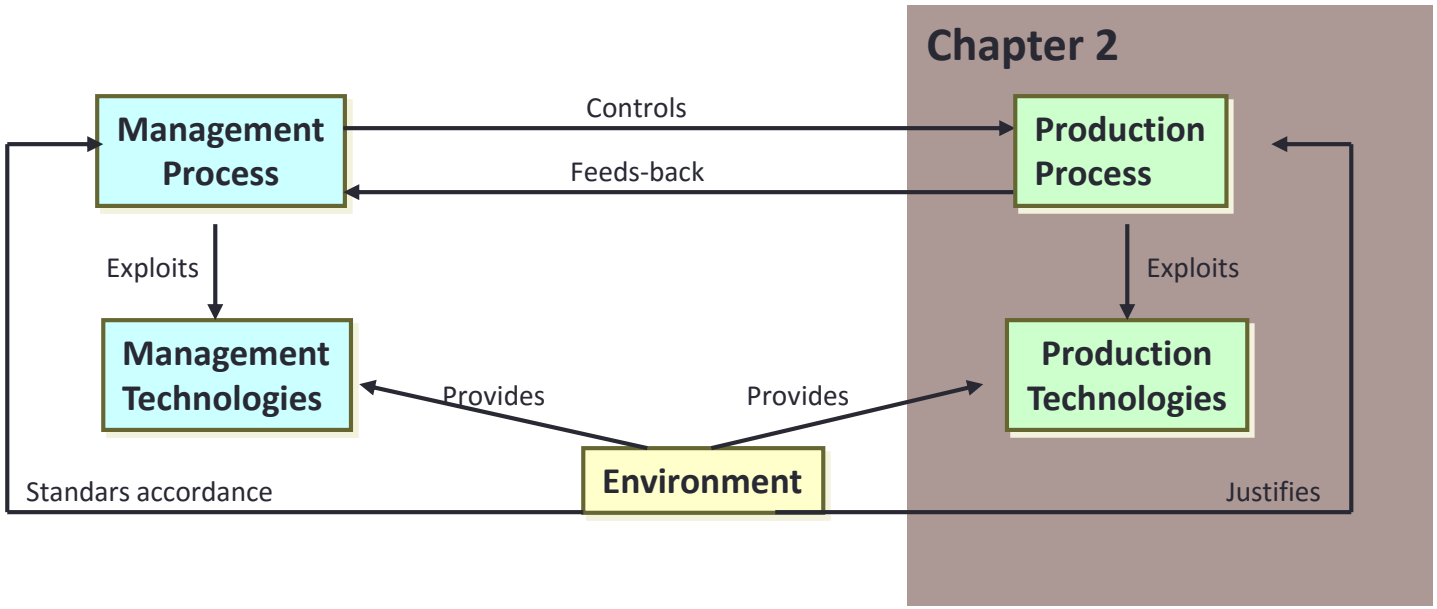
The importance of the process

- The new challenges to be faced when developing software require fast and effective answers to changing requirements.

- The specification of a development process and the use of tools for its execution and monitoring are mandatory

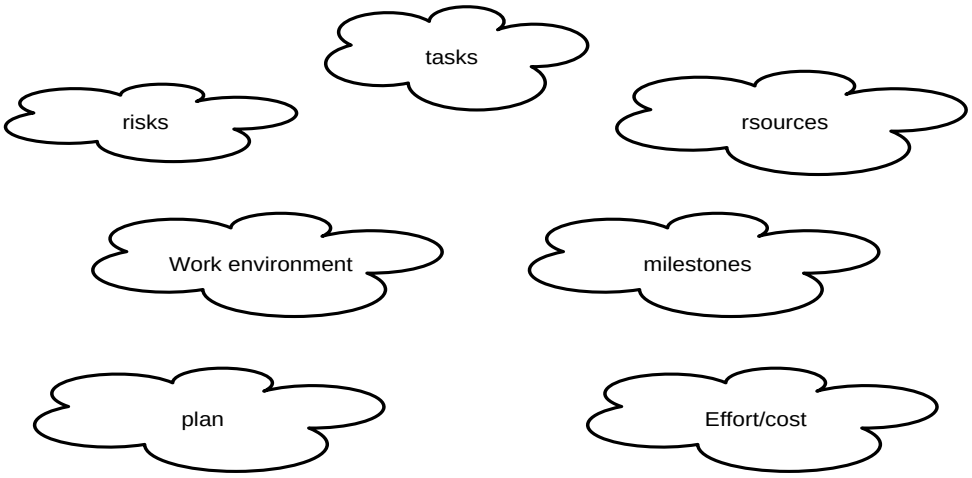
The Software process

- It is a framework for the development of software



Software projects management

- The management of a software project is the first level of a software development process and it covers all the development process



Software engineering vs Engineering

- Similarities
  - Activities to be done are not specific of software projects
  - Many common management techniques
  - Many similar problems (time, resources, changing specifications…)
- Differences
  - The product (software) is not tangible and flexible
  - The software process is not standard. Several alternatives exist
  - Many times software projects are “unique”

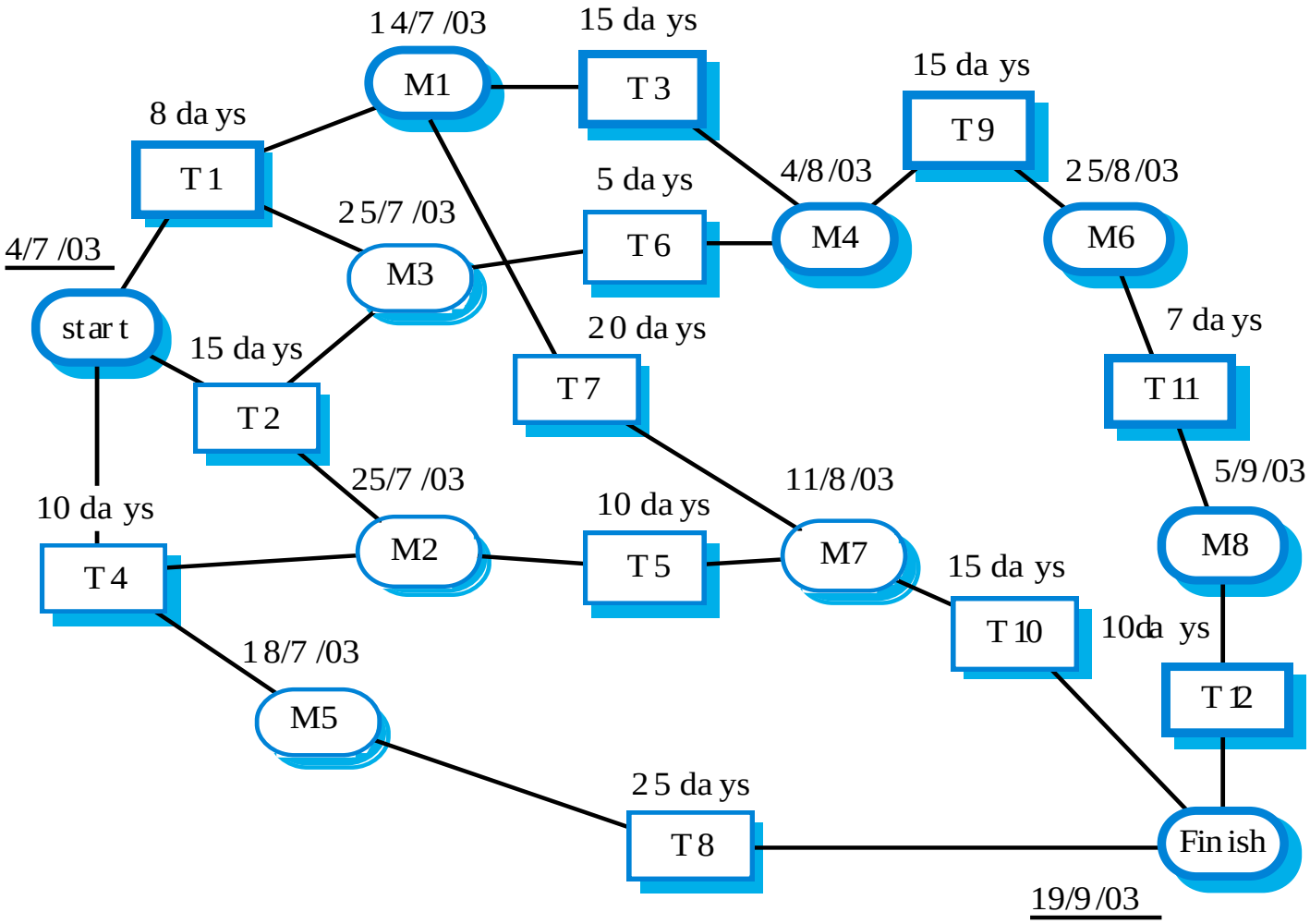
Software projects management

- Activities involved:
  - Writing proposal
  - Project planning
  - Cost estimation
  - Selection and evaluation of human resources
  - Project control
  - Writing and presenting reports

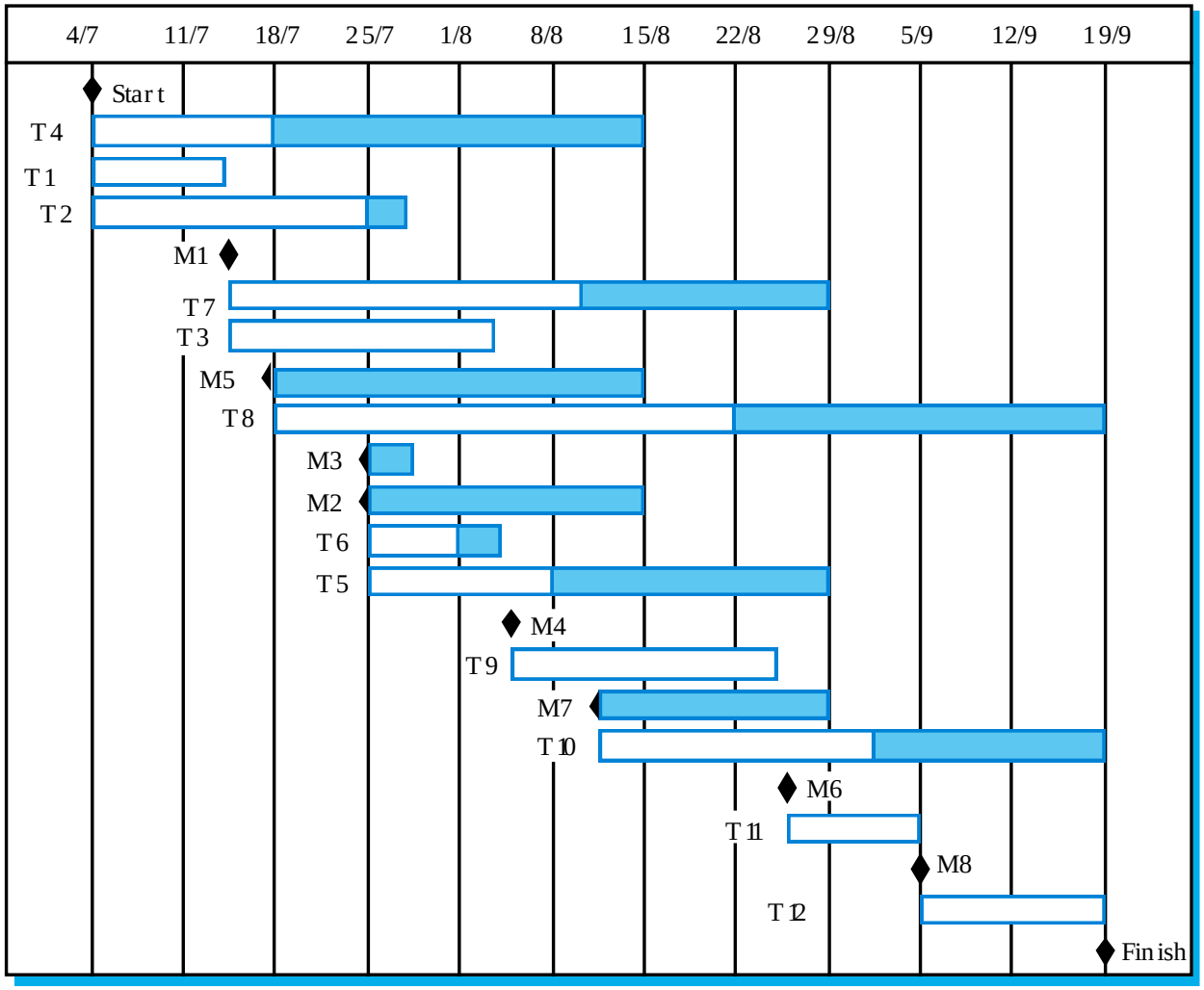
Project Plan - Structure

1. **Introduction** — Goals and restrictions (budget, time…)
2. **Project organization** — Team Organization (personnel-roles)
3. **Risks analysis** — Risks, probabilities and estrategies
4. **Software and Hardware requirements** — Acquisitions, prices, delivery dates …
5. **Division of labour** — Activities, milestones and products to deliver
6. **Planning** — Dependencies between activities; Estimated time, assigning resources
7. **Supervision and reporting**

Tasks networks (PERT diagrams)



Gantt diagram



Seminar

SeC2

Chapter 2.
The Software Process

Software Engineering
Computer Science School
DSIC – UPV

Goals

- Team work (4 persons) related to:
  - "Software Process"
  - Process development models or lifeccycles
  - Methodologies

Agile Methods

- Each Team must nominate an expert on:
  - XP (eXtreme Programming)
  - SCRUM
  - CRYSTAL
  - AGILE UNIFIED PROCESS

Agile Methods

- Phase I: Methodology-Driven Experts Meetings
  - SCRUM Experts Meeting
  - XP Experts Meeting
  - AUP Meeting
  - Crystal Meeting
  - Discover the main features of the given methodology
  - Use Discovery Template

- Phase II: Best Methodology Selection Meeting
  - Each expert of the group presents his(her) methodology to other team members
  - Decide and propose best agile methodology
  - Ellaborate a presentation including 1 slide per methodology and 1 slide to justify your selection.

Questions

Section 1. Process Models

Indicate whether the following statements are true or false. Justify your answers in any case.

1. The goal of Software Engineering is to deal with the implementation of software systems using object oriented languages.
2. The writing process of a project plan is iterative.
3. The quality of a software preoduct is measured once it is finished, just before it is delivered to the customer, verifying whether the predefined quality factors are fulfilled.
4. The participation of the customer in evolutionary process models is minimal, just at the beginning of the process and at the end in the acceptance tests.
5. In the classical model with prototyping, the prototype is generated by means of an automatic procedure. The subsequent development is manually made.

Questions

Section 1. . . .

6. A prototype is a software system with excellent operational features (efficient, robust, etc.)
7. In the automatic software programming paradigm, the prototype is either the specification or it is derived from the specification. However, the maintenance is performed on the code.
8. The software quality factors are focused on the correctness, the ease of maintenance and the portability.
9. What is a prototype and When is it used in the development of a software system?
10. The project plan just includes the planning of the project.

Questions

Section 1. . . .

11. The classical model with prototyping is an evolutionary model because each new prototype is a new version of the product to release to the customer.
12. In the automatic software development paradigm tests are performed on the formal specification.
13. The cost of a project is hard to estimate, mainly due to the cost associated to software engineers.
14. In the administration of a software project, the management of risks involves identifying the risks and assigning to each risk its estimated occurrence probability.
15. The maintenance phase is not considered to be part of any software process model, it is not even considered as a software quality factor.

Questions

Section 2. Methodologies

1. Indicate which ones are the essential elements of a methodology and their relationshio.
2. What is the difference between a process model and a methodology.
3. What two dimensions are defined in RUP. Briefly explain each one of them.
4. How does RUP answer the questions related to a software process: "A software development process defines who does what, how and when"?
5. Summarize the development process in RUP
6. Among the principles of agile methodologies there are some referring to the customer, others to the development team, and others to the process followed. Acording to this categorization, What principles do lie within each category? Justify your answer.
7. Indicate 4 favourable conditions for the application of an agile methodology
8. Summarize the development process in XP

Additional Questions

1. Comment on this statement: "Software engineering is more than coding..."
2. What caused the need for an engineering approach to software development?
3. Define the term "Software Process". Define the term "Software Process Model".
4. Is there any process model that integrates prototyping? Which one? Which one is its goal?
5. What does it mean that a software development process is iterative, evolutionary and interactive?
6. ¿What inconvenients does it have the fall development lifecycle? When is it appropriate to use it?
7. Explain four relevant features of the automatic prototyping approach.
8. Explain the main differences between the incremental lifecycle and the spiral one.
9. What is the difference between the following quality factors: correctness, reliability, integrity?
10. Explain why the writing process of a project plan is iterative and must be reviewed continuously during the project.
11. In the administration of a software project, what is a milestone? Give an example.
12. Identify four possible risks that may arise ina software project. Explain how would they affect to a project.
13. Explain at least four causes for the following situation: "Software products are low quality, costs are high and deliveries are late"
14. What do we mean with the term "high quality software"?

Additional Work

Methodologies.....

- Methodologies
- Comparison between methodologies
- Aplication of methodologies
- ..... .

(Agree the work to be done with your professor)

THE SOFTWARE PROCESS

Chapter 2

Software Engineering
Computer Science Engineering School
DSIC – UPV

Goals

- Define term "Software Process"
- Present main development process models that have been proposed
- Introduce the notion of methodology, presenting RUP and the main features of agile methodologies.

Contents

References

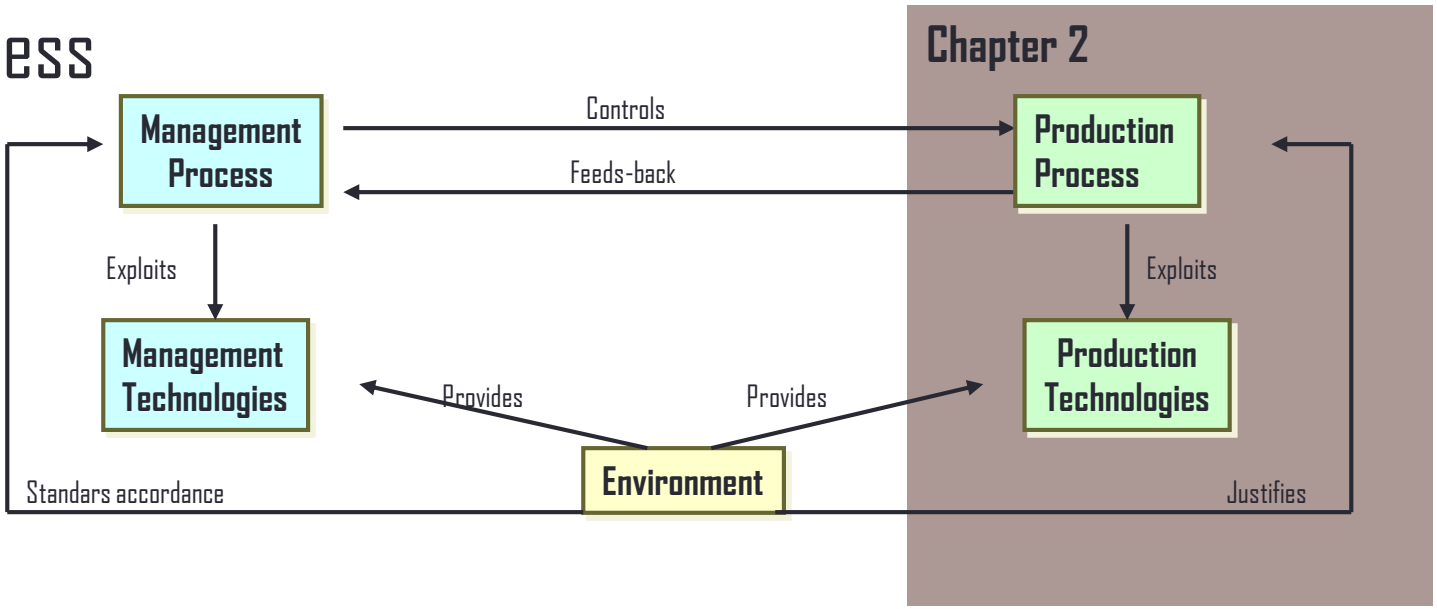
- Sommerville, I. Ingeniería del Software. (8ª ed.). Addison-Wesley, 2008
- Presman, R.S., Ingeniería del Software: un enfoque práctico (6ª ed.), McGraw-Hill, 2005
- Royce, W.W., Managing the Development of Large Software Systems: Concepts and Techniques. Proc WESCON, 1970
- Agresti, W.W. Tutorial: New Paradigms for Software Development. IEEE Computer Society Press, 1986
- Balzer, R., Cheatman, T.E. and Green, C., Software Technology in the 1990's: Using a New Paradigm, IEEE Computer, Nov. 1983, pp. 39-45
- McDermid, J. And Rook, P., Software Development Process Models. Software Engineer's Reference Book, CRC Press, 1993
- Boehm, B.W.. A Spiral Model of Software Development and Enhancement, IEEE Computer, pages 61-72, May 1988.

References

- Krutchen, P., *The Rational Unified Process- An Introduction*. Addison –Wesley, 1998
- Jacobson, G. Booch and J. Rumbaugh., *The Unified Software Development Process*, Addison-Wesley, 1999
- Beck, K., Extreme Programming Explained: Embrace Change. The XP Series. Addison-Wesley, 2000

The Software process

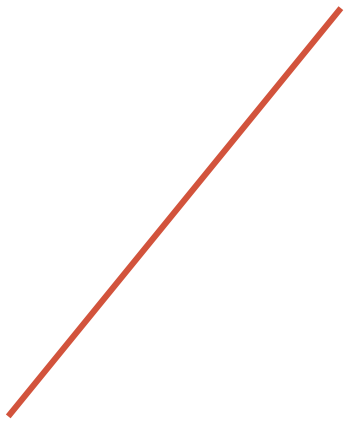
- It is a framework for the development of software
- In general the term "Software Process" is associated to the production process... but it includes the management process



The Development Process

- Collection of activities towards the development or evolution of software
- Also known as **Lifecycle**

- **Generic Activities** that are always carried out:
  - Specification
  - Development
  - Validation
  - Evolution

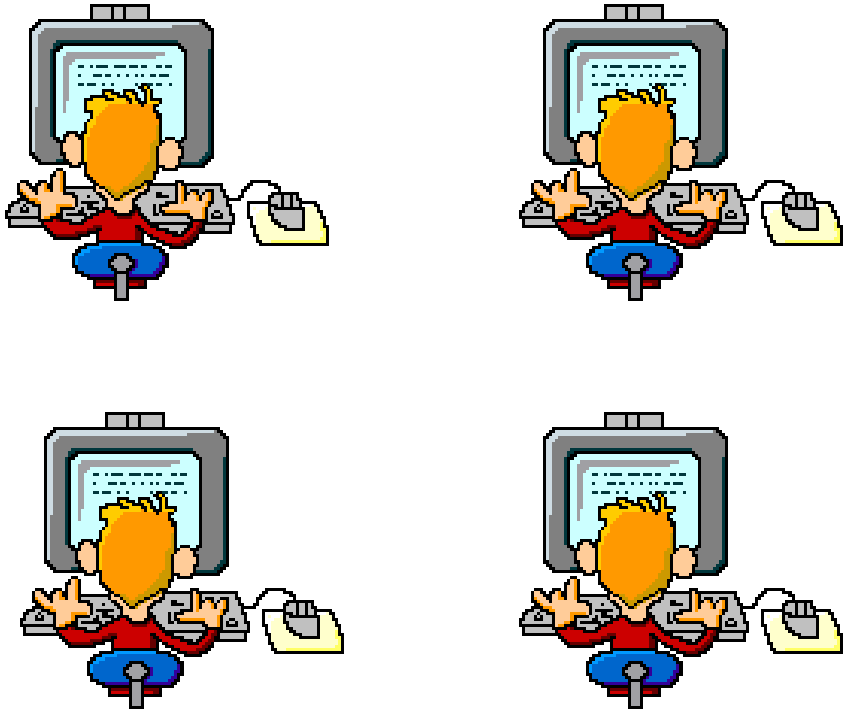


- Analysis
- Design
- Implementation
- Testing
- Maintenance

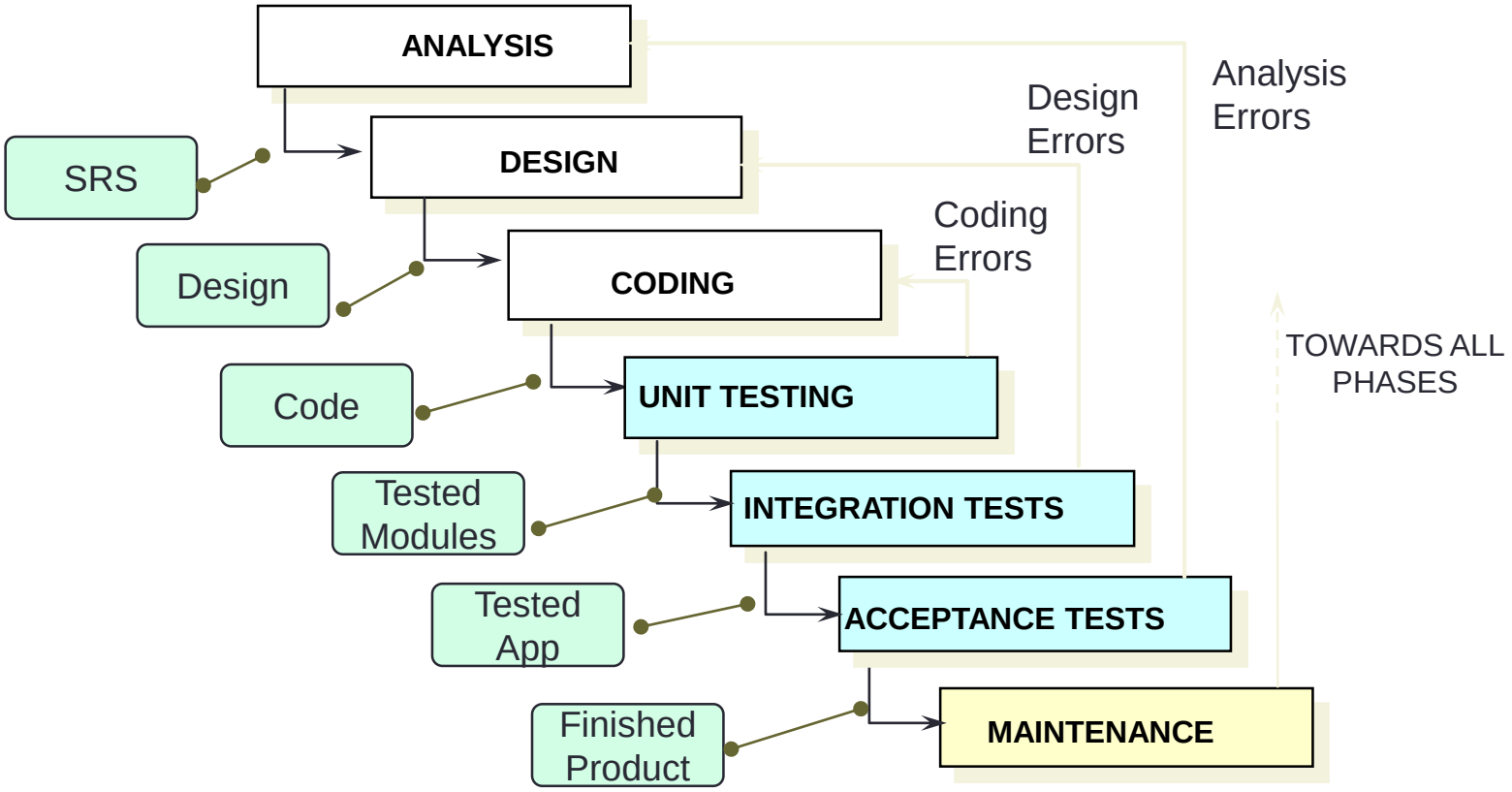
Lifecycle Models

- *Code-and-fix*
- Classic or Waterfall
- Classic with prototyping
- Automatic Code Generation
- Evolutionary Models:
  - Incremental
  - Spiral

Code and Fix

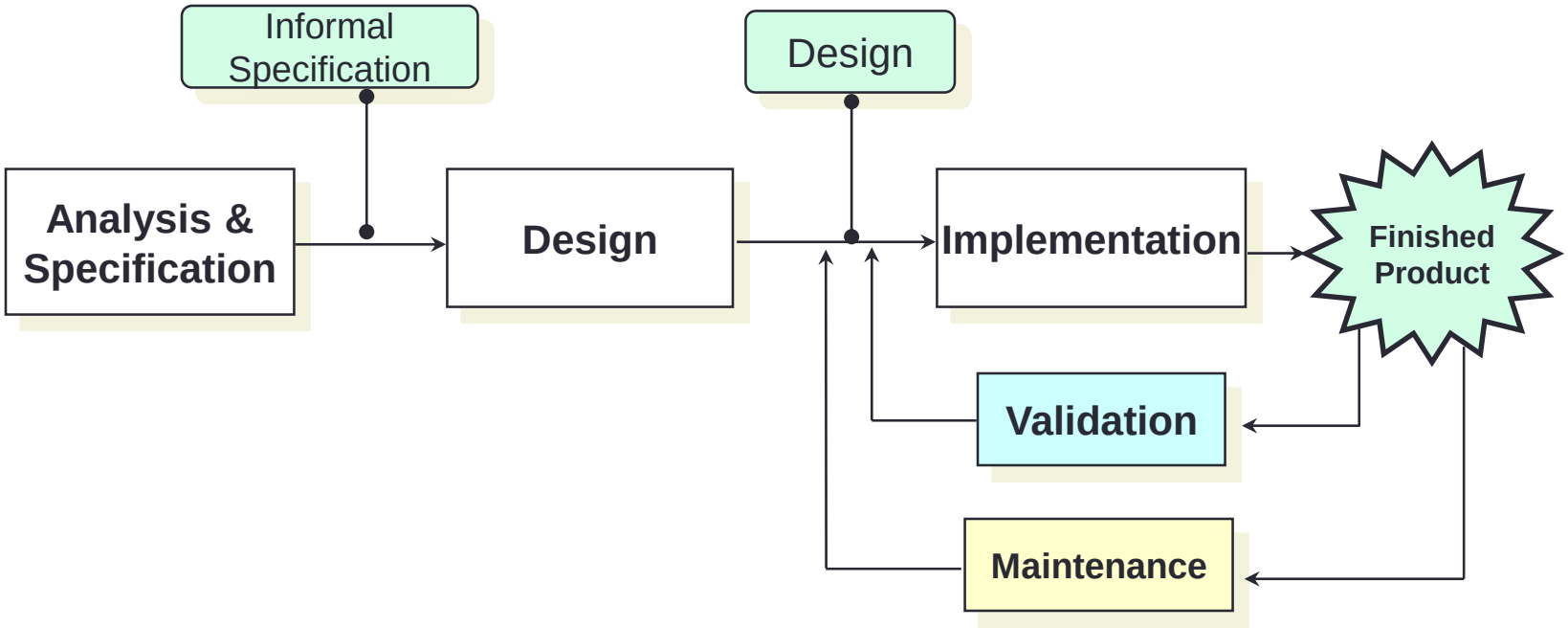


Classic or Waterfall



Classic or Waterfall

- In practice this model is "distorted" and all the validation and maintenance is performed on the source code.

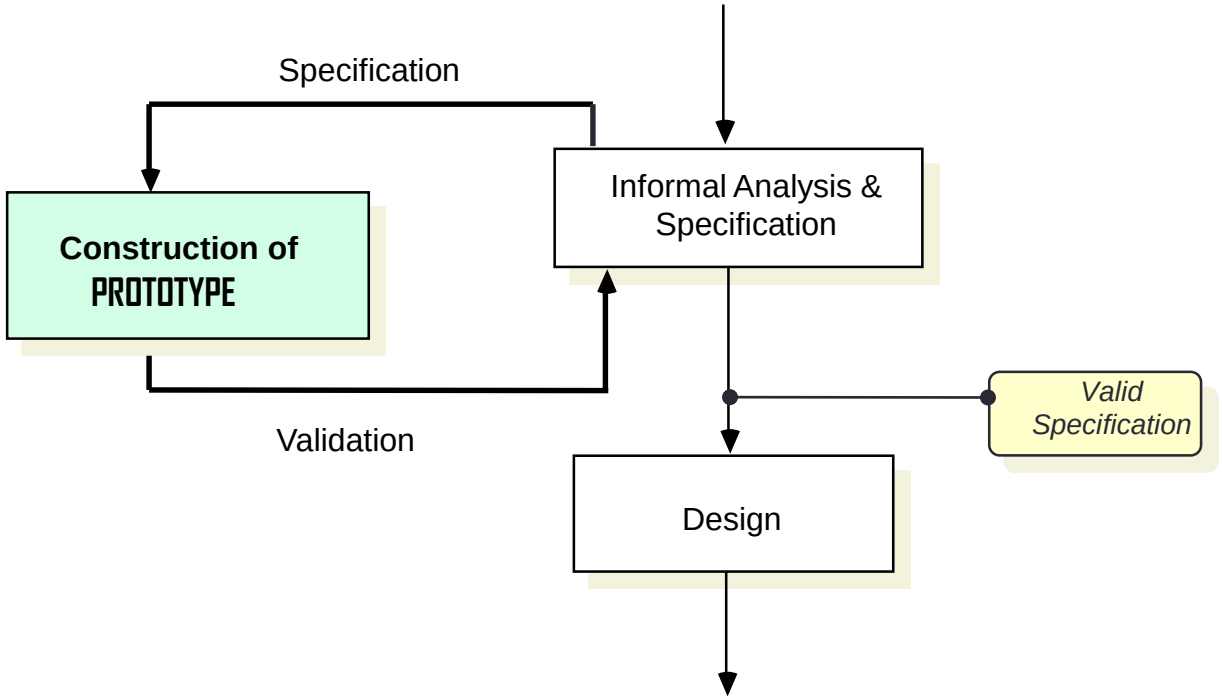


Classic Model with Prototyping

- **Prototype:** First version of a product in which only some features are integrated or all of them are featured but unfinished

- Types of prototypes:
  - Vertical: some functionality of the system is fully developed.
  - Horizontal: all views of the system are shown (simulated)

Classic Model with Prototyping



- It helps customers to clearly establish the requirements
- It helps developers to improve their products

Classic Model with Prototyping

- Criticism:
  - ☺ It reduces the risk of patching on the final product (code maintenance is not avoided)
  - ☺ It helps both customers and developers to understand the requirements
  - ☹ The customer sees a version of the final product (not assuming it is not robust and incomplete)
  - ☹ It requires an additional investment (the invested time may result in loosing market opportunity)
  - ☹ Bad decisions taken during a rapid development of the prototype are usually transferred to the final product

Automatic Code Generation

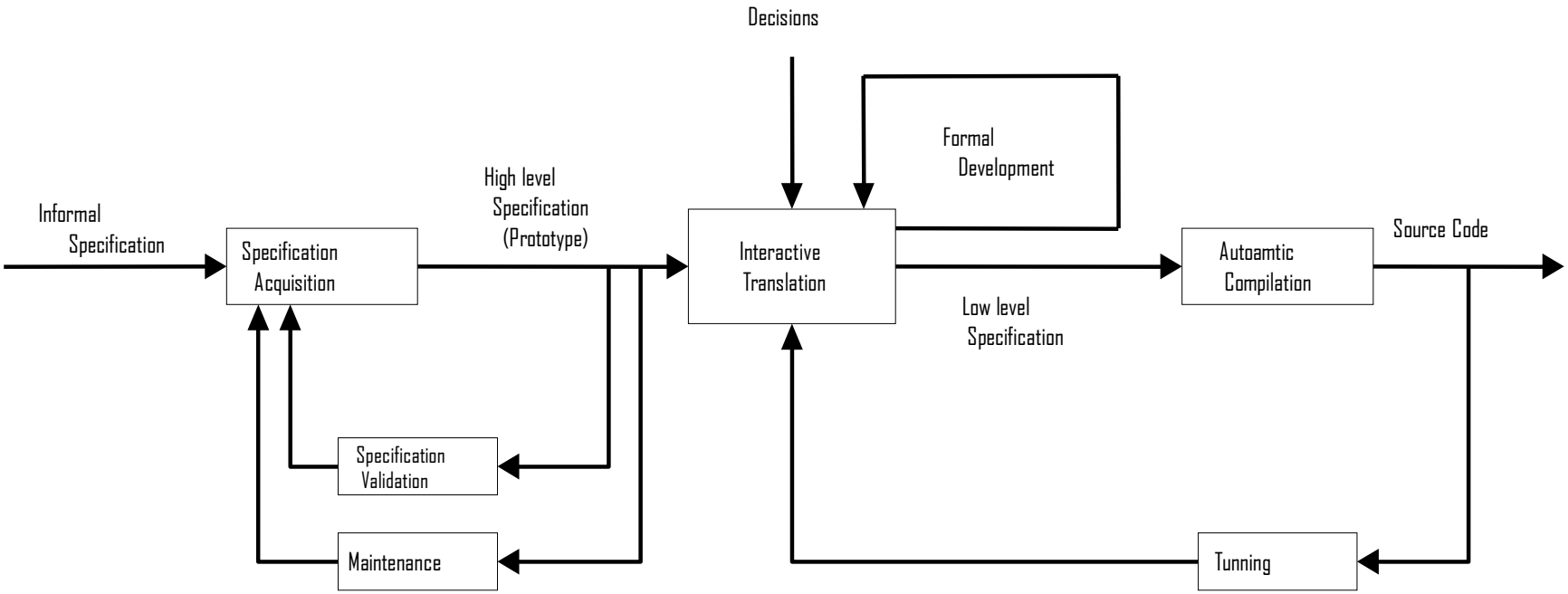
(R. Balzer, 1983)

- Goal

  Automatize the software development process

- Basic Features:
  - ✓ Use of formal specification languages
  - ✓ The specification is a prototype of the product
  - ✓ The requirements are discussed by running the specification
  - ✓ The application is derived semi-automatically

Automatic Code Generation Model



Automatic Code Generation Model

- Comparison

CLASSIC Prototyping

- Informal Specification
- Non standard prototype
- Prototype manually built
- Prototype discarded
- Manual implementation
- Code must be tested
- Maintenance on the code

AUTOMATIC GENERAT.

- Formal specification
- Standard prototype
- The specification is the prototype
- It evolves towards the final product
- Automatic Implementation
- No testing
- Maintenance on the specification

Automatic Code Generation

- Criticism

  ☺ It helps reducing human errors

  ☺ It reduces development costs

  ☹ It is difficult to use formal languages

➡ *It is the predecessor of MDE/MDA*

Evolutionary Development

- Adaptable to changing requirements

- More ellaborated versions are built at each iteration

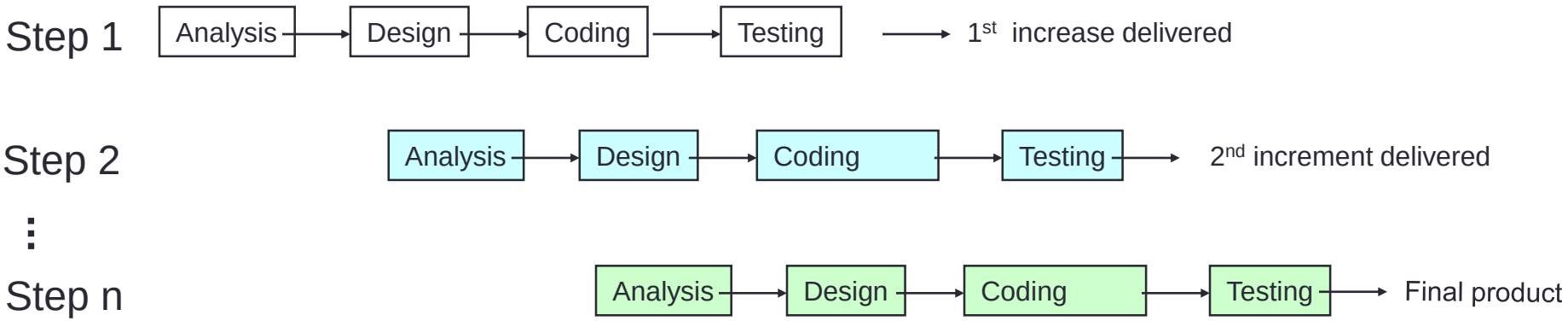
➡ Incremental Model

➡ Spiral Model

Incremental Model

(McDermind, 1983)

- Sequence of applications of the classical model
- Each iteration produces a delta of the product
- It ends when the final product is delivered



Incremental Model

- Criticism

☺Useful when not enough human resources for a complete deliverable

☺ Each deliverable may be evaluated by the customer ➔highly interactive

☹ Difficult to know the required increase for each iteration

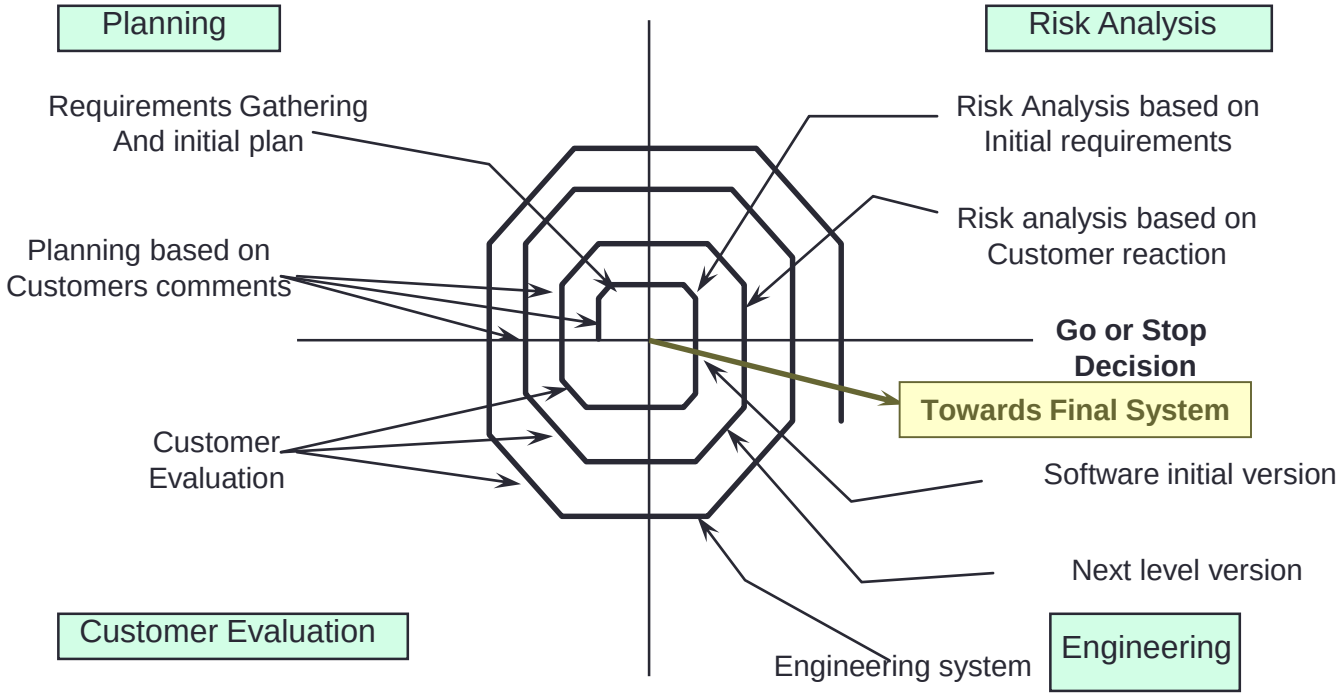
Spiral Model

(B. Boehm, 1988)

- Approach:
  - Iterative.
  - Interactive.
  - Evolutive

- It introduces risks analysis in the development process

Spiral Model

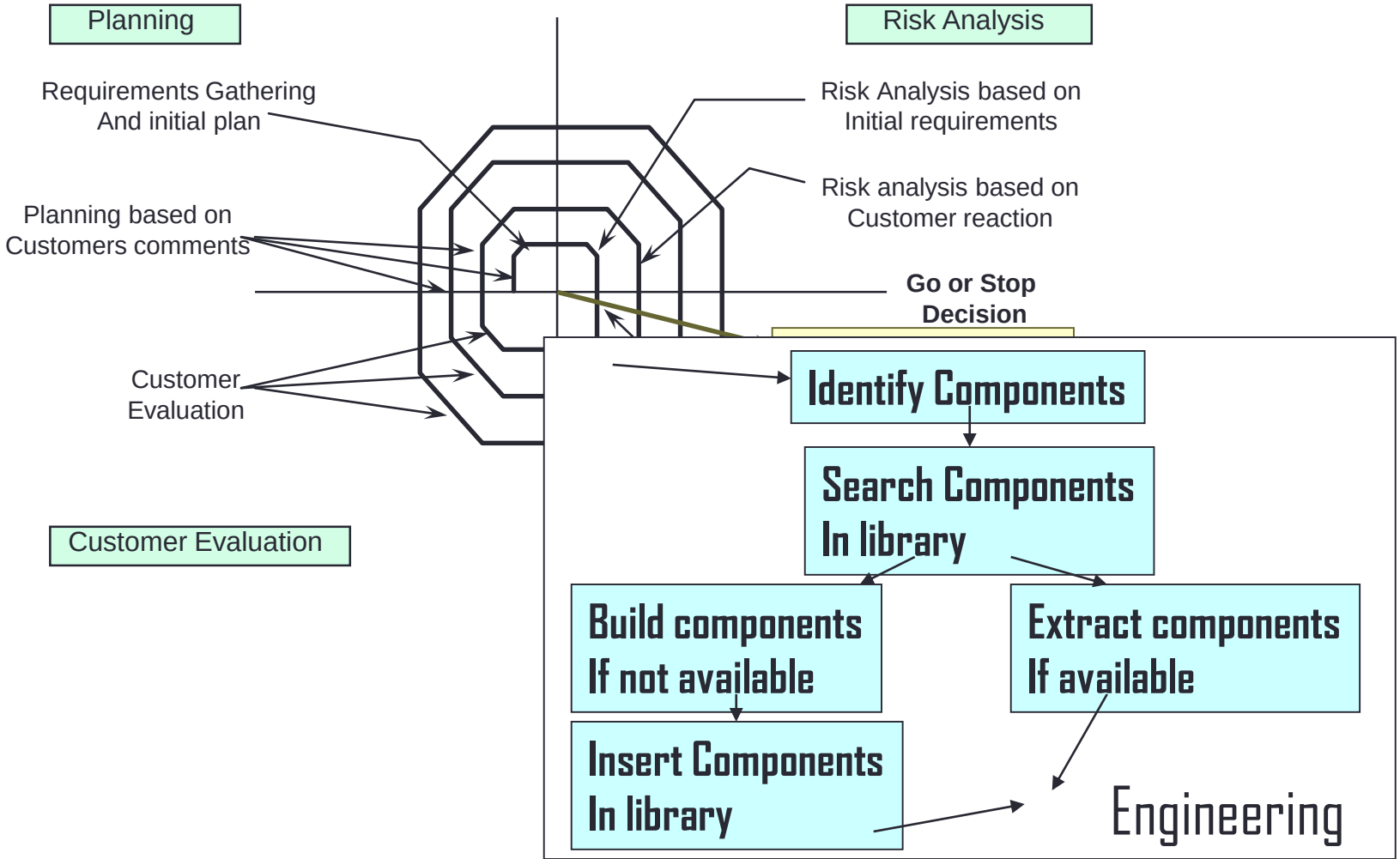


Spiral Model

- Criticism
  - ☺ Each time more complete versions of the product are obtained.
  - ☺ Each version is evaluated by the customer ➔ Highly interactive
  - ☹ It is difficult to assess risks
  - ☹ Hard to guarantee path towards the final product

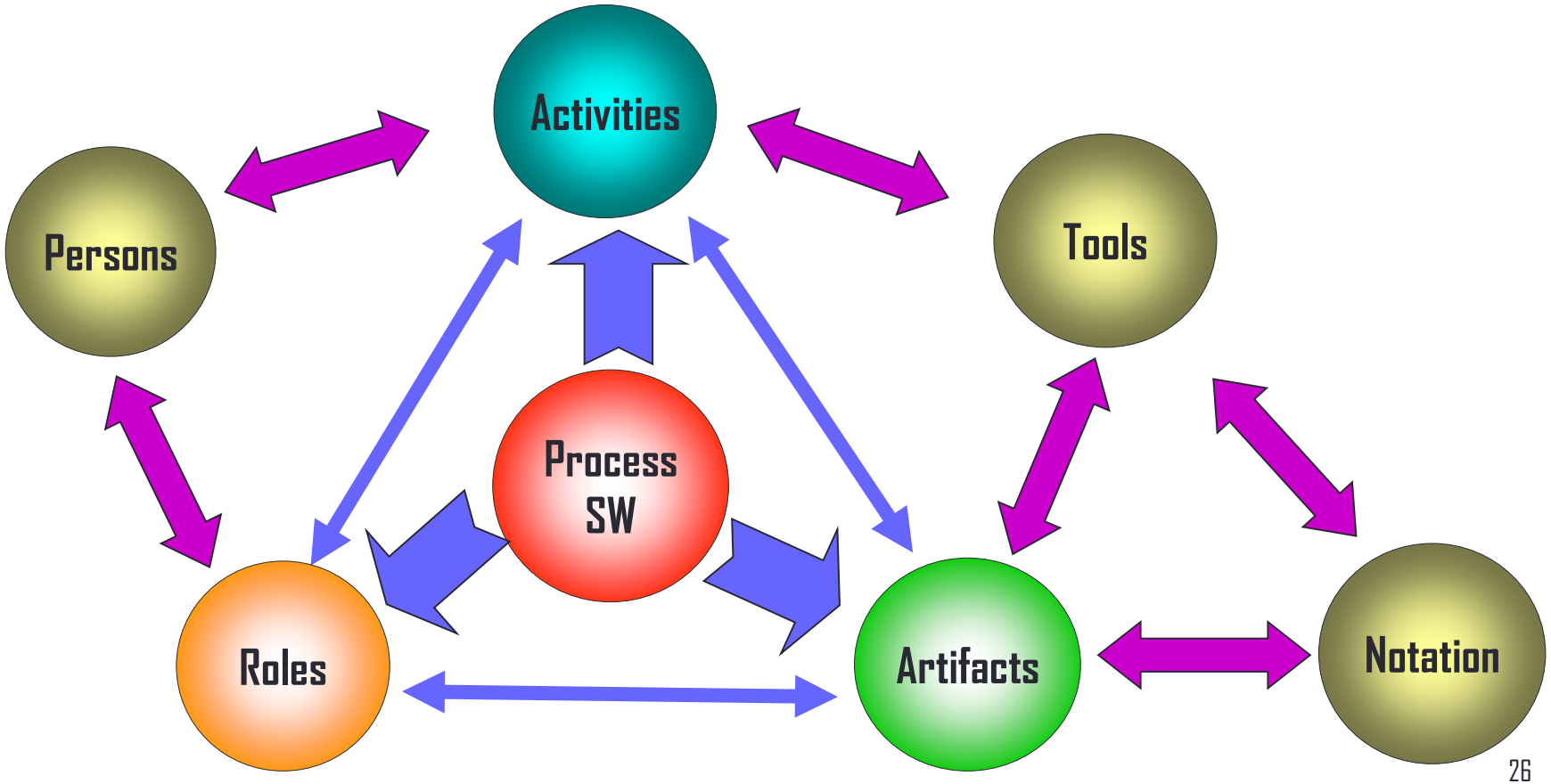
Components Assembly Model

- The engineering phase may be adapted to new requirements



Methodology

- In a software development project, the methodology defines: Who/ What / How / When



Methodology

- Defines an explicit process of software development

  (its goal is the formalization of activities related with the elaboration of information systems)

- This process must be:
  - Reproducible
  - Defined
  - Measurable with respect to performance
  - Subject to Optimizations
  - ...

Methodology

There is no universal software methodology.

Structured methodologies

Object oriented methodologies **RUP**

Traditional methodologies ***vs.*** *Agile methodologies*

RUP **XP**

Agile Methodologies

Agile Methodologies appreciate:

- The individual and the interactions within the development team more than the activities and the tools
- The development of software that works rather than obtaining a good documentation $\Longrightarrow$ Minimalistic approach wrt modelling and documentation of the system
- The collaboration with the customer rather than the negotiation of a contract
- The fast response to changes rather than following a strict planning

http:\\www.agilealliance.com

Agile Methodologies

Principles of Agile Methodologies (1/2)

1.- The main priority is to satisfy the customer with early and continuous releases of usable software.

2.- Welcome changes. Agile processes apply updates for the customer to remain competitive.

3.- Release the developed software frequently and with the shortest possible interval of time between releases

4.- Business people and developers work together as a team in a project

5.- Build project driven by personal motivations. Provide the environment that people need and trust them.

Agile Methodologies

Principles of Agile Methodologies (2/2)

6.- Face to face dialogue is the most efficient and effective method to communicate information within a development team

7.- Developed software is the first metric of progress

8.- Agile processes promote a bearable development. Funding entities, developers and users are capable of keeping a peaceful ambient

9.- The continuous attention to technical quality and good design increases agility

10.- Simplicity is key

11.- The best architectures, requirements and designs arise from the organziation of the team

12.- At regular intervals, the team reflects about how to be more effective and how to synchronize and adjust their work.

Agile Methodologies

- Comparative

Agile Methodology	Non Agile Methodology
The customer is part of the Development team (*on-site*)	The customer intereacts with the team By means of meetings
Small teams (< 10 members) Working at the same place	Large teams
Few artifacts	More artifacts
Few roles	More roles
Less emphasis on the architecture	The architecture is essential

Agile Methodologies

- Comparative

Agile Methodology	Non Agile Methodology
Heuristics	Rigurous
Tolerant with updates	Resistant to updates
Internally imposed (by the team)	Externally imposed
Less controlled process, with Few principles	Highly controlled process with many Policies and norms
No traditional contract or at least very flexible	There is a prefixed contract

Main Agile methodologies

- Extreme Programming (XP) http://www.extremeprogramming.org
- SCRUM http://www.controlchaos.com
- Crystal Methods http://alistair.cockburn.us/Crystal+methodologies
- Adaptive Development Software (ADS) http://www.adaptivesd.com
- Dynamic Systems Development Method (DSDM) http://www.dsdm.org
- Feature-Driven Development (FDD) http://www.featuredrivendevelopment.com
- Lean Development (LD) http://www.poppendieck.com

Extreme Programming (XP)

Kent Beck, Ward Cunningham y Ron Jeffries

www.extremeprogramming.org

www.xprogramming.com

- Design for dynamic environments
- Ideal for small teams (<= 10 coders)
- Strongly oriented towards coding
- Emphasis on informal and verbal communication
- Other values: simplicity, feedback and courage

XP

Development Cycle

Stories, Iterations, Versions, Tasks and test cases

✓ The customer selects the **next version** to be built, choosing the **functional features** that he considers more valuable (known as **Stories**) from a set of possible stories, being informed about *costs* and the required *time* of their implementation.

✓ Coders **convert stories** into **tasks to be done** and then convert **tasks** into a **set of test cases** to demonstrate that the tasks have been completed.

✓ Working with a teammate, the coder **runs the test cases** and **updates the design (evolution)** trying to keep it simple.

XP

Planning

Small deliverables

40 hours weeks

Simple design

Coding in pairs

Laboratory

tests

Collective ownership

Metaphore

Refactoring

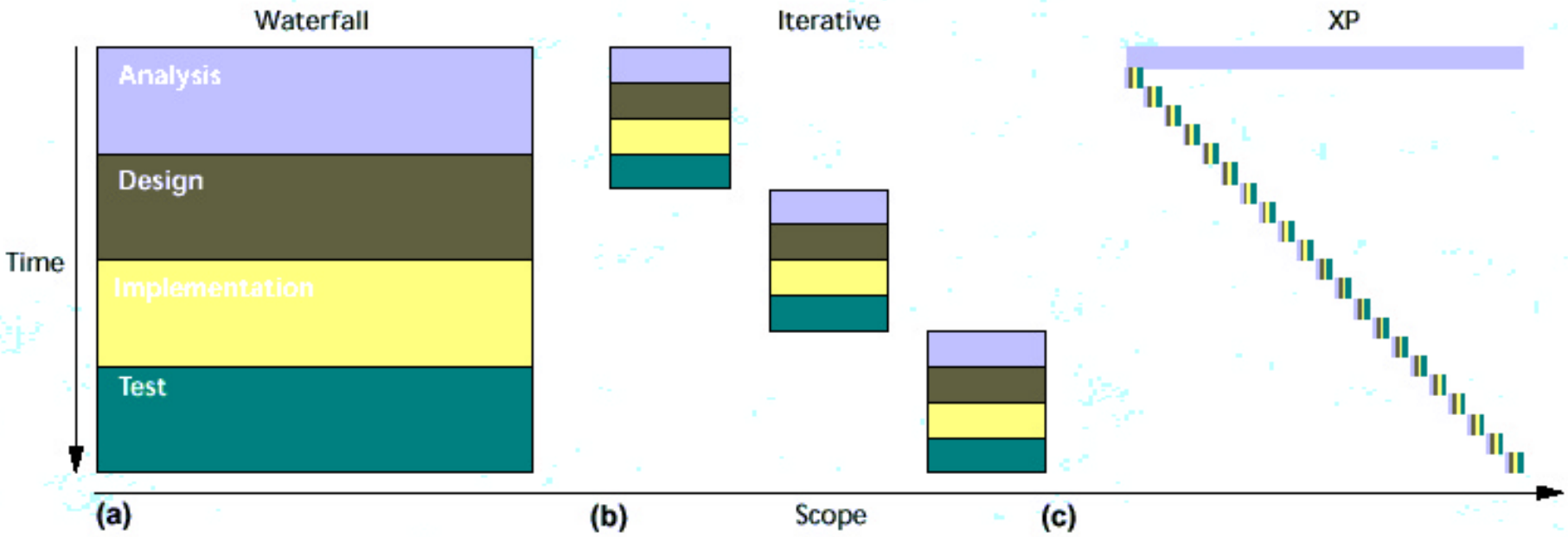
The customer always with the coder

Continuous integration

Coding standards

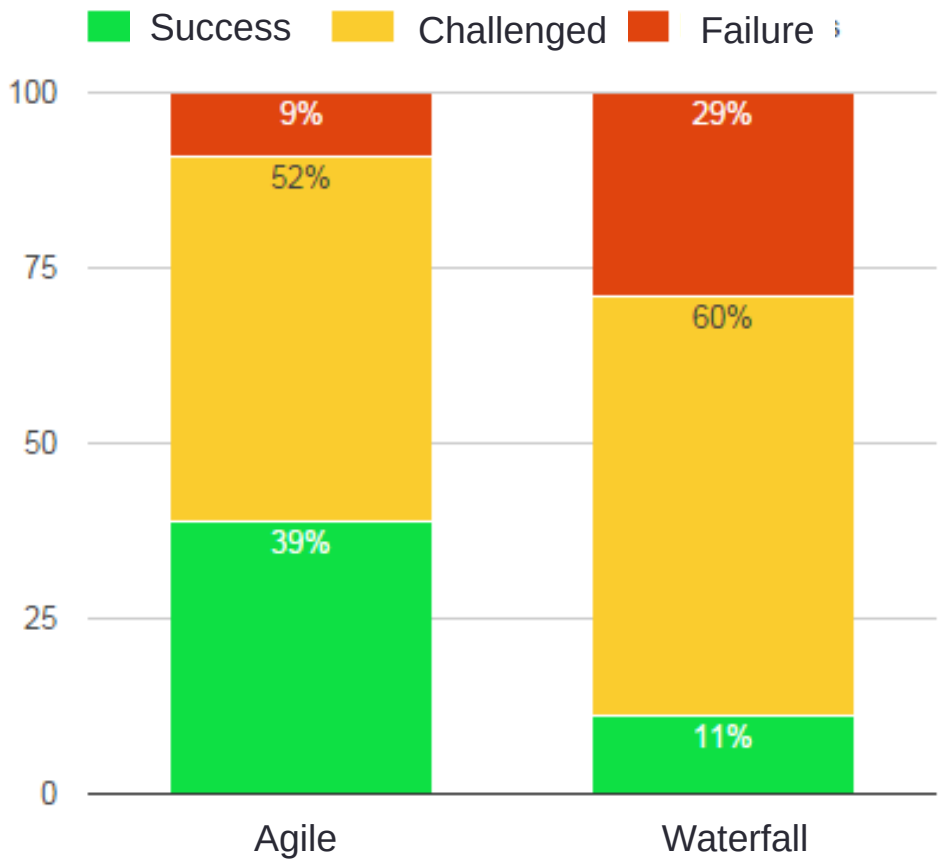
XP

Comparative



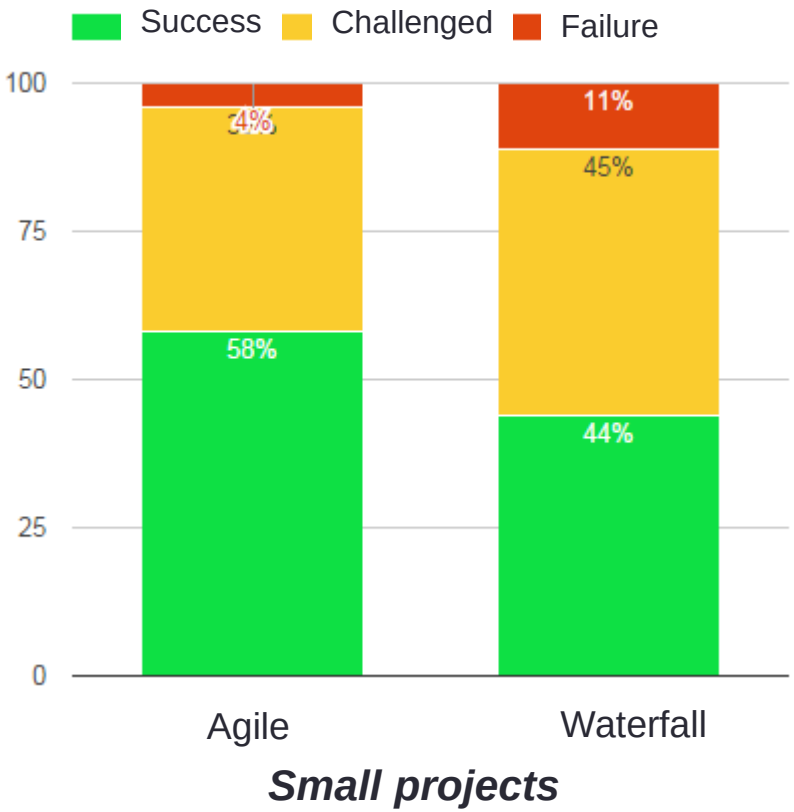
Agile vs. Waterfall

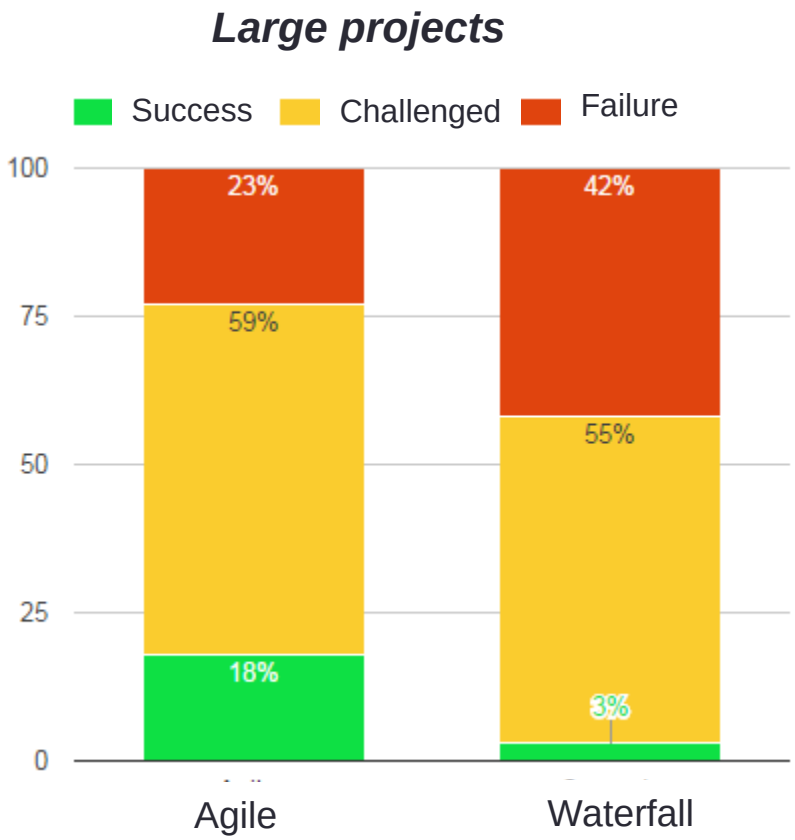
Success based on methodology
2011-2015



http://www.laboratorioti.com/2016/05/16/informe-del-caos-2015-chaos-report-2015-bien-mal-fueron-los-proyectos-ano-2015/

Agile vs. Waterfall





http://www.laboratorioti.com/2016/05/16/informe-del-caos-2015-chaos-report-2015-bien-mal-fueron-los-proyectos-ano-2015/

ANNEX - Rational Unified Process (RUP)

Software development process
(Rational – IBM)

Uses UML as modelling language

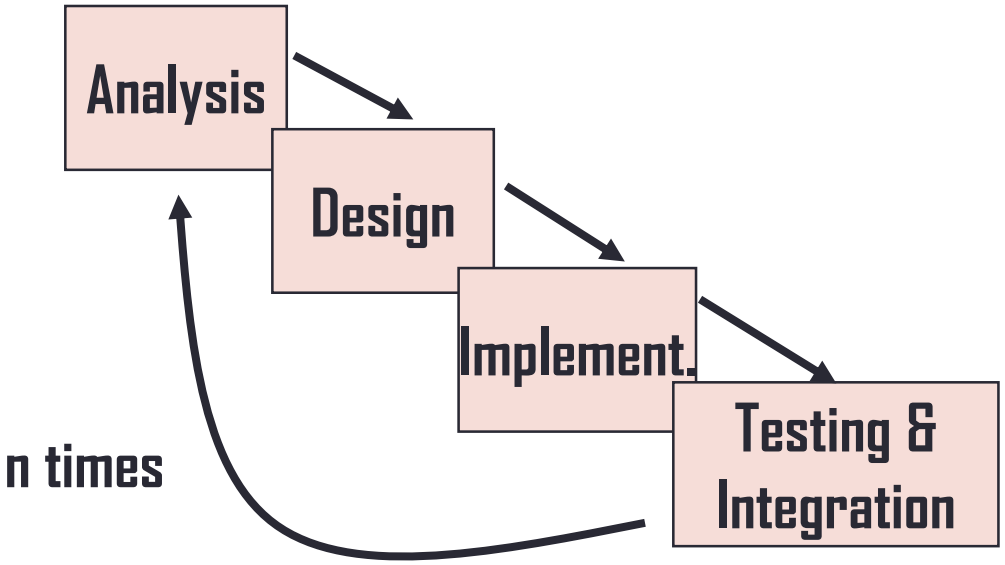
Features:

- *Use cases driven process*: from specification to maintenace
- *Iterative and incremental process:* iterations depending on the importance of use cases and the study of risks.
- *Architecture centered process*: reusable and serving as a guide towards the solution

RUP

- Iterative and Incremental

  Activities are performed in a mini-fall with a limited scope (the goals of the iteration)

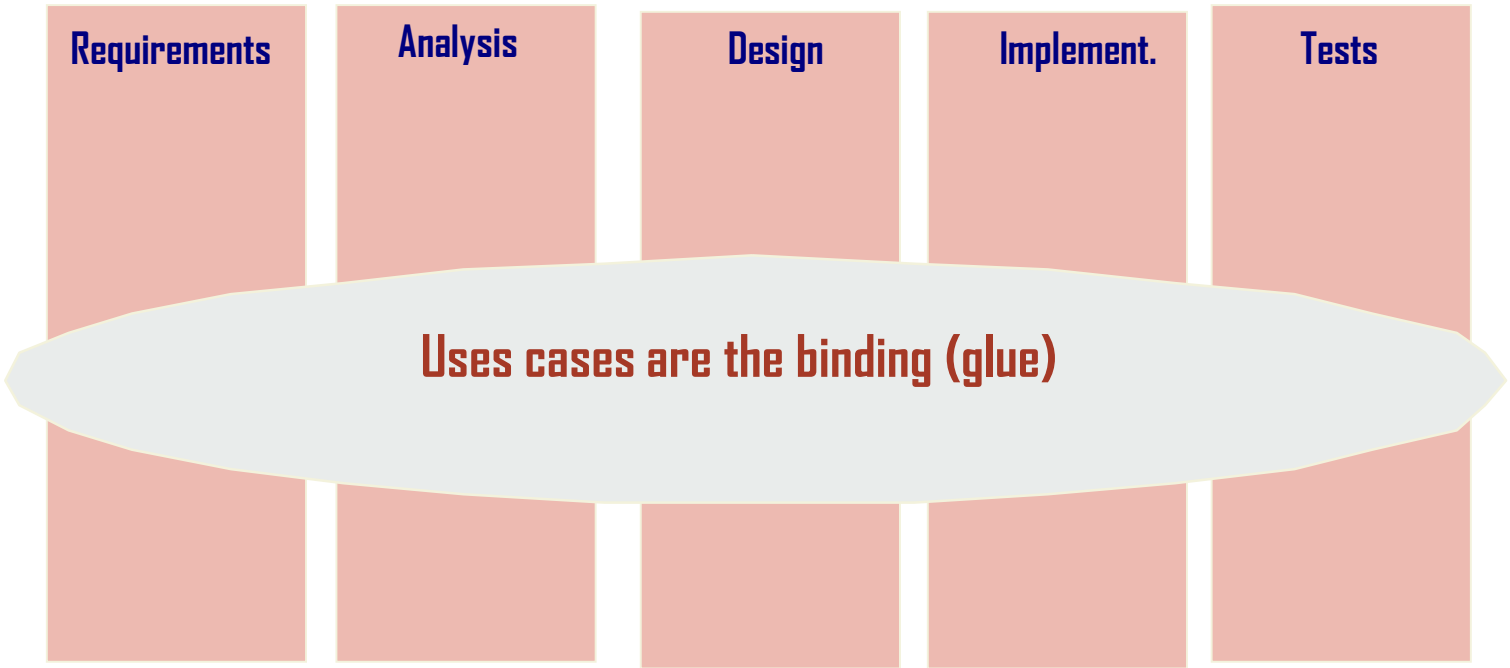


ACTIVITIES OF THE ITERATION

- Plan iteration (risks)
- Analysis of Use cases and Scenarios
- Design of Architectural choices
- Implementation
- Tests
- Integration
- Evaluation of release
- Preparation of release

RUP

- Use cases driven



Capture, clarify & validate use cases

Realize use cases

Verify whether use cases are satisfied

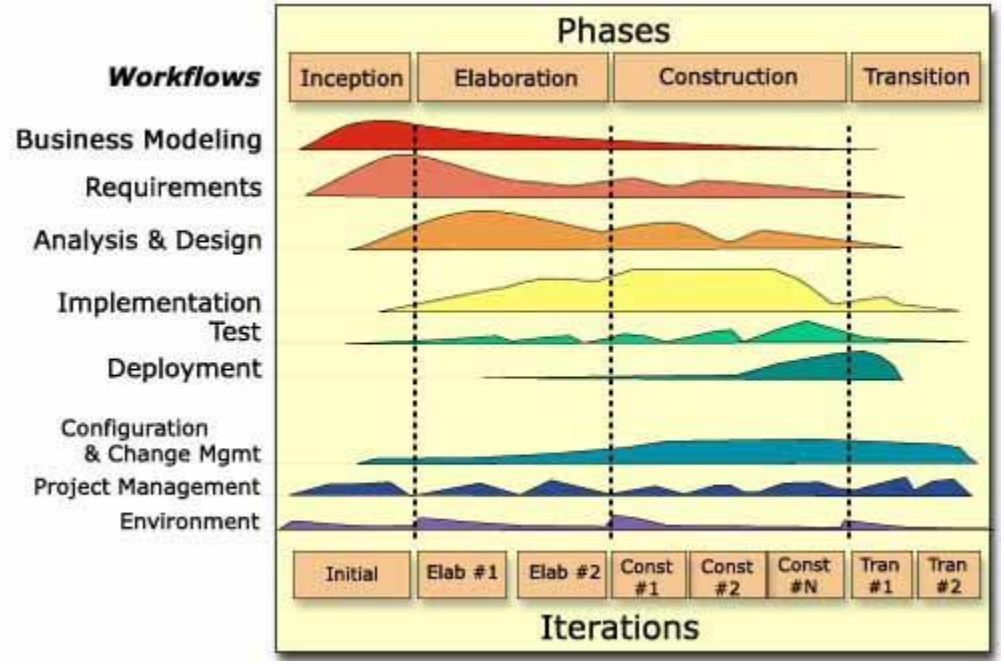
RUP

Dynamic View

Horizontal Axis: Time oriented organization

Static View

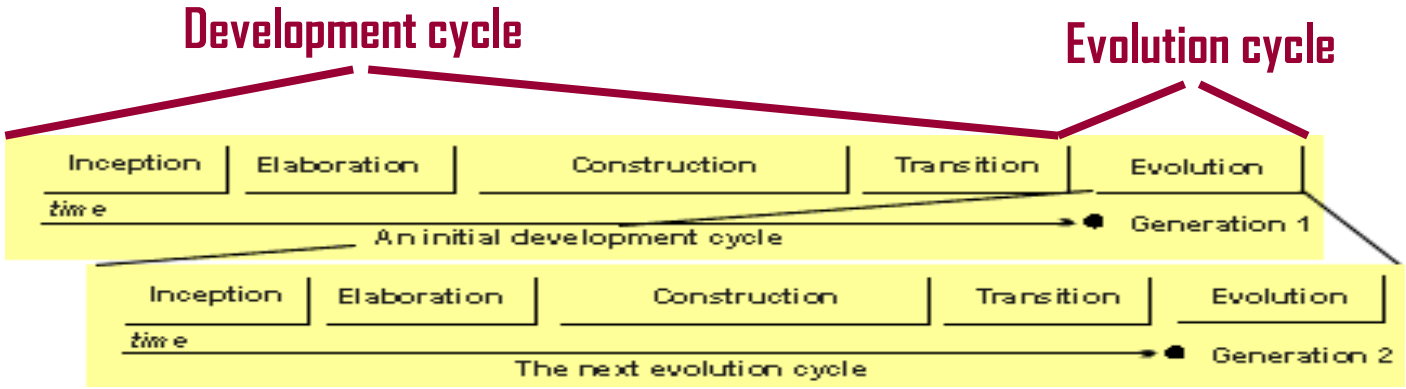
Vertical Axis:
Content oriented
organization

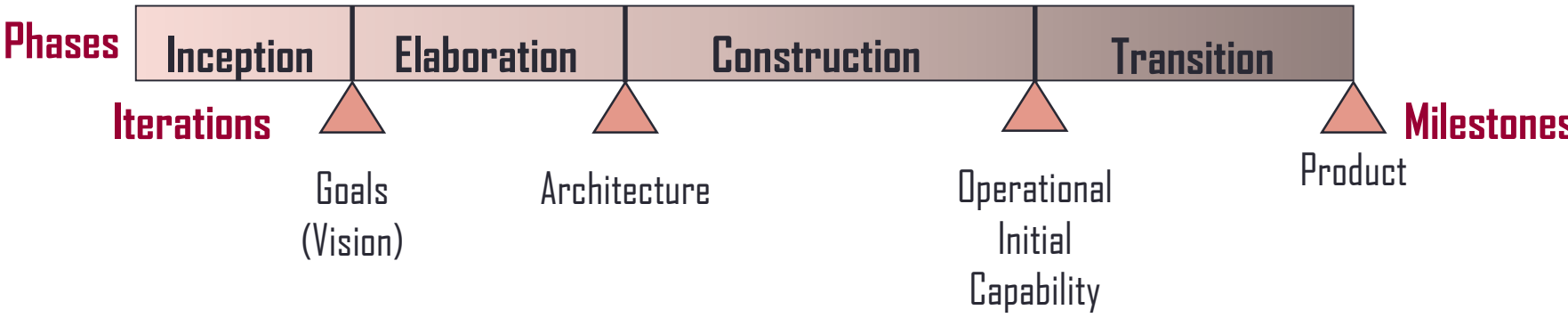


RUP

Dynamic View

- Cycles, Phases, Iterations and Milestones





RUP

Dynamic View

- Phases
  - *Inception(Opportunities Study)*
    - The scope and goals of the project are defined
    - The functionality and capabilities of the product are defined
  - *Elaboration*
    - The problem domain and the desired functionality are studied in depth
    - The basic architecture is defined
    - The project plan is defined according to the available resources

RUP

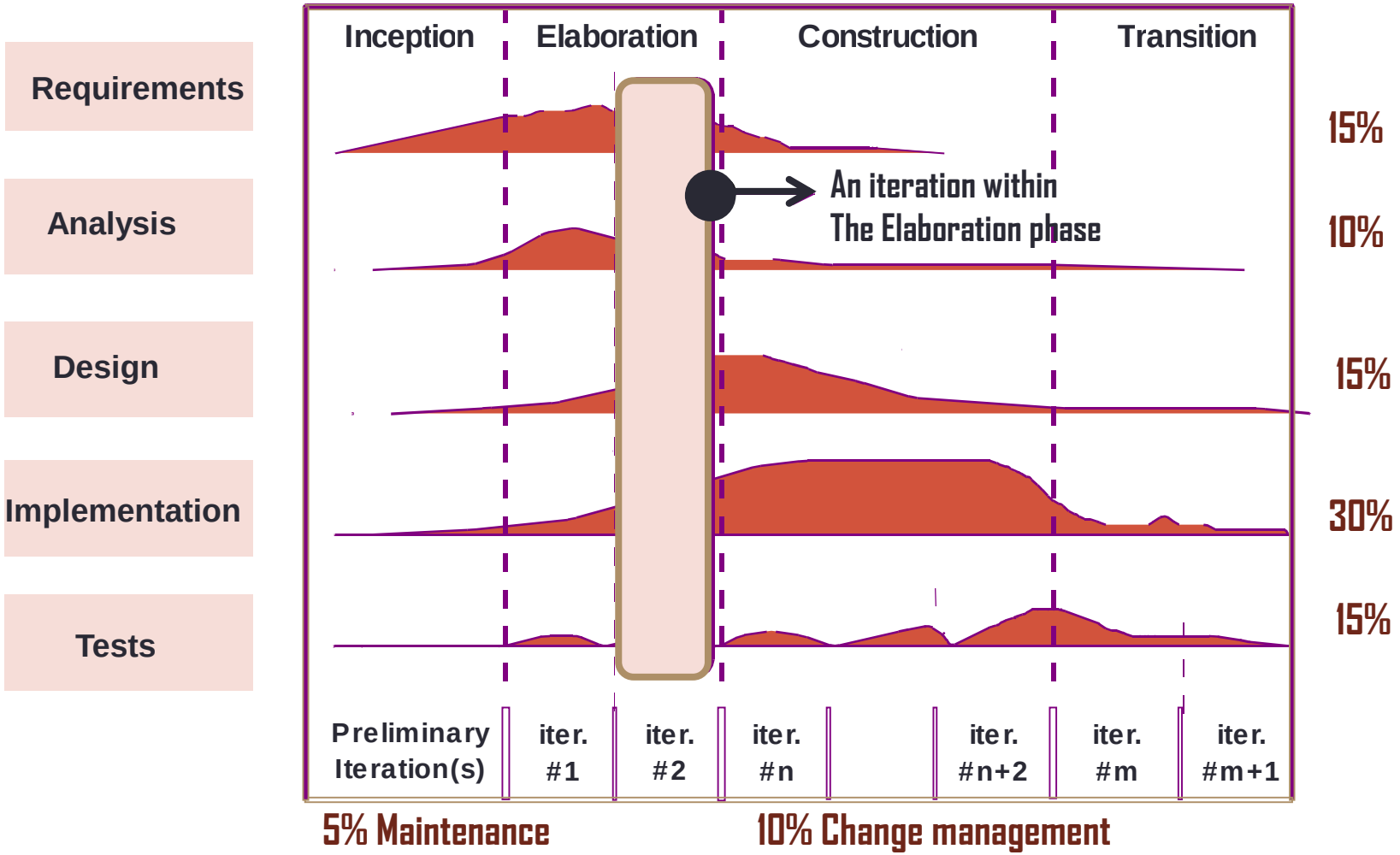
Dynamic View

- *Construction*
  - On each iteration analysis, design and implementation tasks are performed
  - The architecture is refined
  - An important part of the work is dedicated to coding and testing
  - The system and its use is documented
  - This phase provides a built product and a documentation

- *Transition*
  - The product is delivered to the user for its use
  - Marketing, packaging, installation, configuration, training, support and maintenance, ...
  - User, installation,... guides are completed and refined

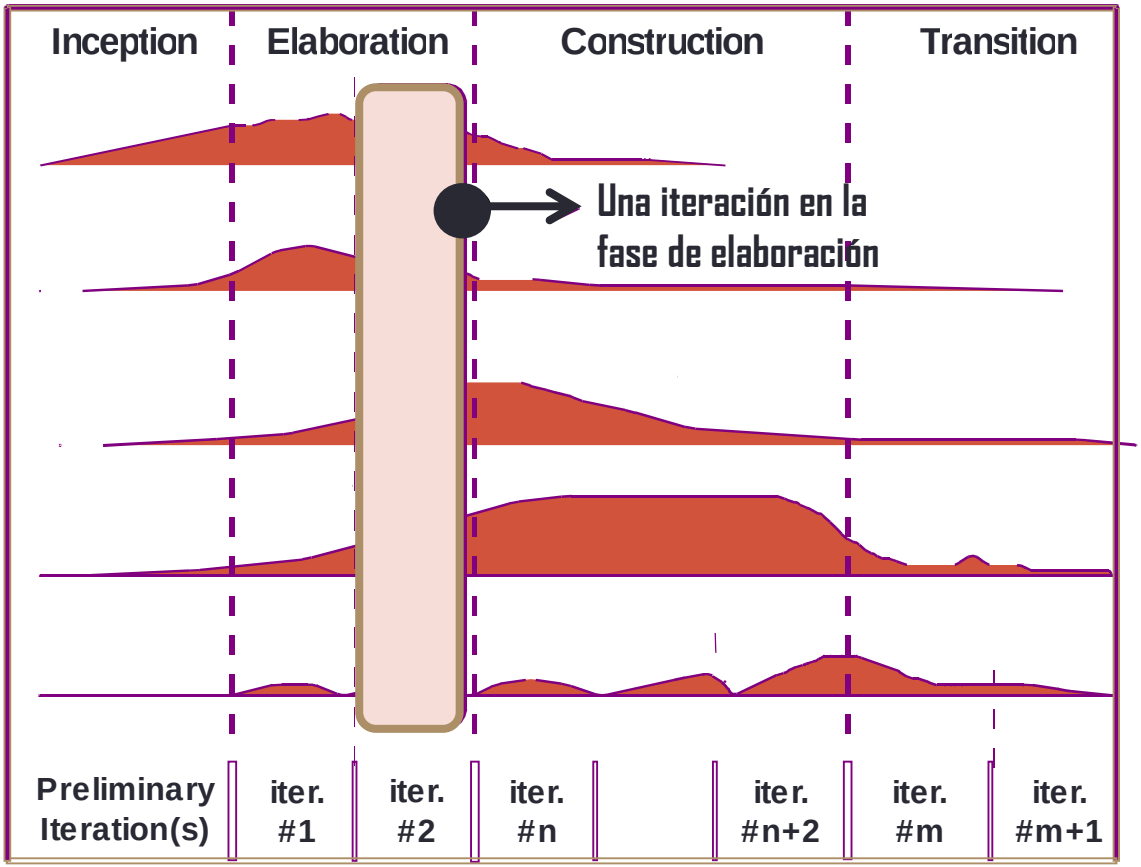
Dynamic View

RUP - *Distribution of **effort** with respect to activities*



Dynamic View

RUP - *Distribution of **effort** wrt phases*



Effort:	5%	20%	65%	10%
Duration:	10%	30%	50%	10%

RUP

Static View

- Workflows

Workflow	*Description*
Business Modelling	Business processes are modelled using business use cases
Requirements	Actors are defined that interact with the system and use cases are developed to model the requirements of the system
Analysis & Design	A design model is created using architectural models, component models, object models and interacion models.
Implementation	The different components of the system are structured and implemented. The automatic generation of code helps to speed up this process.
Tests	Testing is an iterative process that takes place simultaneously with the implementation. As soon as the implementation is finished the integration tests take place.
Deployment	A *release* (version) of the product is created, distributed to the users and installed in their workplace.

RUP

Static View

- Workflows

Workflow	*Description*
Configuration and Change Management	To manage changes in the system
Project Management	To manage the development of the system
Environments	Development of appropriate software development tools for development teams.

SOFTWARE ARCHITECTURES

Chapter 3

Software Engineering
Computer Science School
DSIC – UPV

Goals

- Introduce the concept of Software Architecture
- Describe the main features of distributed systems, in particular the multi-layered architecture.

Contents

1. INTRODUCTION

Programming in the small/medium/large

Modules

Classes

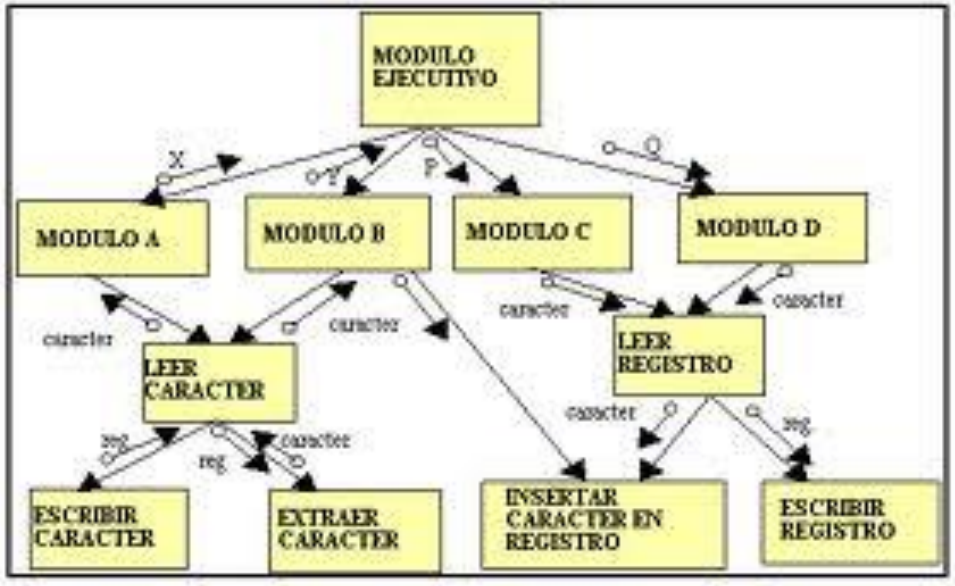
Programming in the small/medium/large

- When systems grow in size it is required an organization in terms of subsystems so that they are manageable

- Throughout history of software development different strategies to manage complexity, usually related with design at different levels of abstraction, have been used

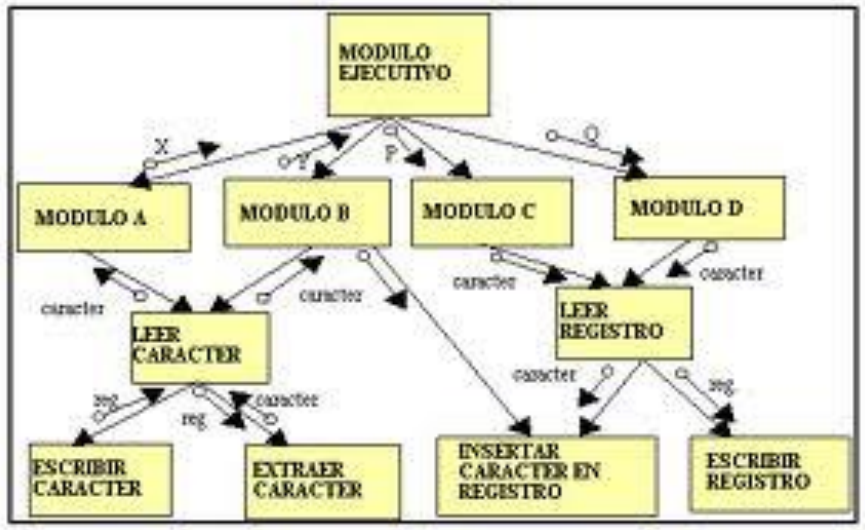
Structured Methods

- Structure diagram
  - Based on the notion of module (Parnas, 1972)
  - A system is partitioned in modules that invoque or provide service to other modules, possibly with data passing in both directions



Module

- Part of a program that implements part of the functionality
- Characteristics:
  - White/Black box
  - Encapsulation



- A module may be decomposed in terms of other modules of a lower level

Modular Architecture

- Preliminary Design: Structuring the system in terms of modules
  - Building structure diagram
  - Module = black box
- Detailed design:
  - Description of process that are implemented by modules
  - Module= white box

Object Oriented Architetures

- Classes as decomposition units
  - Structure + behavior in a module
- The structure of classes is propagated to the code
  - New classes are incorporated when lowering the abstraction level
- Packages as a way of grouping classes in self-contained componets
  - A Java package is a mechanism to organize classes that may be reused if a '.jar' file is created that may be imported in another project. It is called a Java component and it may contain classes and other Java components or libraries

Problems

- Approaches based on modules and objects are low level ones.
- They do not divide the application in terms of functional blocks but they are mere groupings of code

- A more abstract mechanism is necessary to clearly detect the aspects that are present in most software systems

SOFTWARE ARCHITECTURES

Client-Server

Multi-Layered

What do we mean with "Software Architecture"?

The ***software architecture***, has to do with the design and implementation of high level structures. It is the outcome after assembling a number of different architectural elements in order to adequately satisfy both functional and non functional requirements such as trustability, scalability, portability and availability.

Kruchten, Philippe

Software architecture is important

- In the description phase of the **Software Architecture** the system must be organized in terms of **subsystems.**

- Many times the architecture is based on other similar previously developed systems by means of **architectonic patterns**.

- Some interesting patterns in information systems are: interactive systems, multi-layered systems, **distributed systems**, real time systems, etc.

Types of Systems (non exhaustive list...)

- **Distributed Systems**:

  A software system in which information processing is distributed among different computing nodes.

- **Personal Systems**:

  Non distributed systems that are designed to be run in a personal computer or workstation.

- **Embedded Systems**:

  Information systems (hardware + software), usually real-time ones integrated in a more general engineering system that perform functions of control, processing and/or monitoring.

Distributed Systems Architectures (non exhaustive)

- **Multi-processing architectures:**

  The system consists of multiple processes that may or may not be run in different processors.

- **Client/Server architectures**:

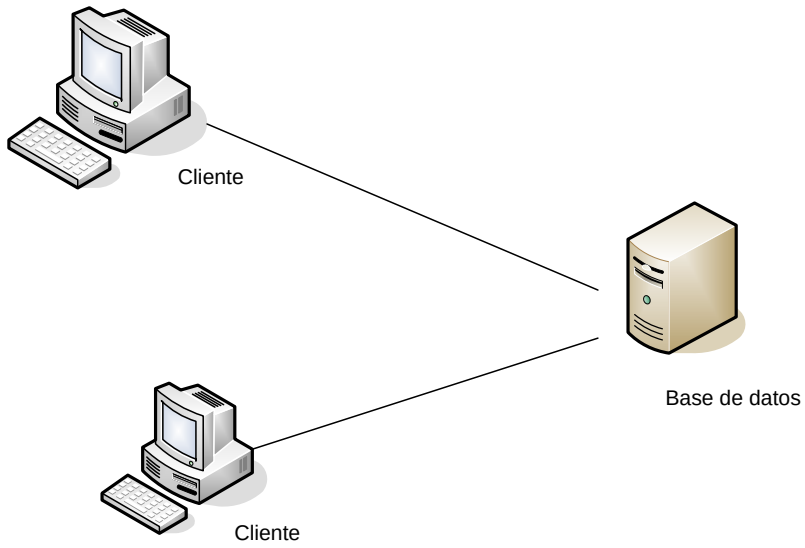
  The system is seen as a set of services that are provided to client applications by server applications. Client and server applications are handled separately.

- **Distributed objects architectures**:

  The system is seen as a set of interacting objects whose location is not relevant. There is no distinction between a provider of a service and a consumer.

Client Server Architecture

- C/S divides an application into 2 components which are run in 1 or more devices:
  - The server (S) is a service provider.
  - The client (C) is a consumer of services.
- C and S interact by means of a message passing mechanism:
  - Service request.
  - Answer.



Multi-Layered Architecture

A **layered system** is a sorted set of subsystems each one defined in terms of the ones located below them and providing the implementation base of the systems above.

- The objects in each layer may be independent (recommended) although there use to be some dependencies between objects of different layers.

- There is a relationship **client/server** between the lower layers (providing services) and the upper layers (using those services).

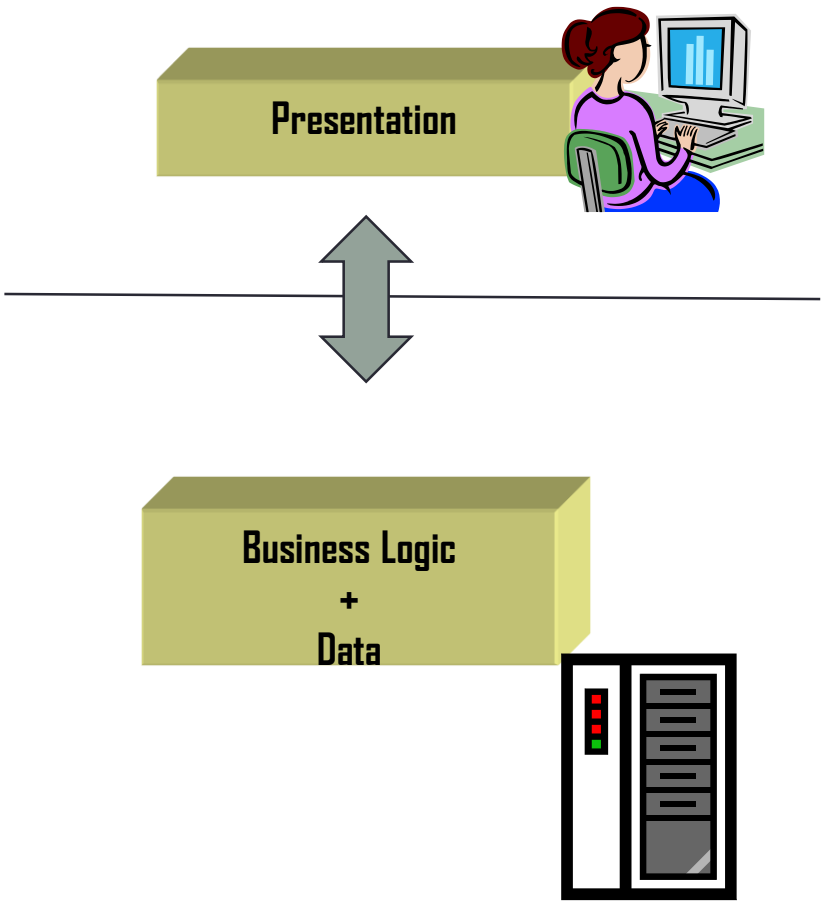
Multi-Layered Architecture

Layered architectures may be **open** o **closed** depending on the dependencies between layers.

- **open**: a layer may use characteristics of any layer.
- **closed**: a layer may only use characteristics of its adjacent lower layer.

It is recommended to use **closed** architectures, because there are fewer dependencies between layers and because it is easier to apply changes because the interface of a layer only affects to its immediate upper layer.

2 – layers Architectures: Thin clients

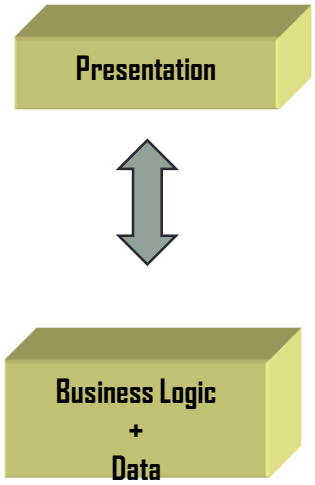


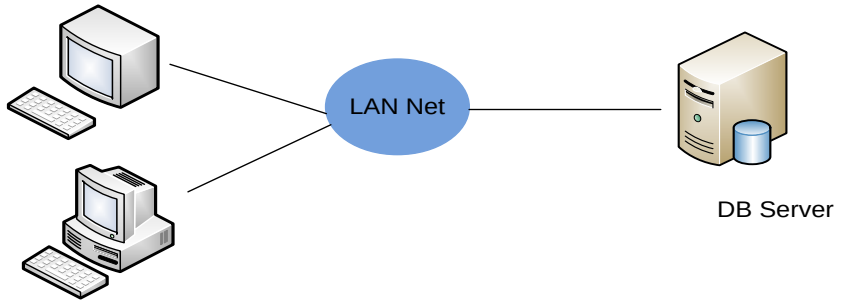
Useful for:

- Legacy systems in which the separation between processes and data management is not feasible
- Data intensive applications (queries and navigation on a DB) with little processing

Layers versus Tiers

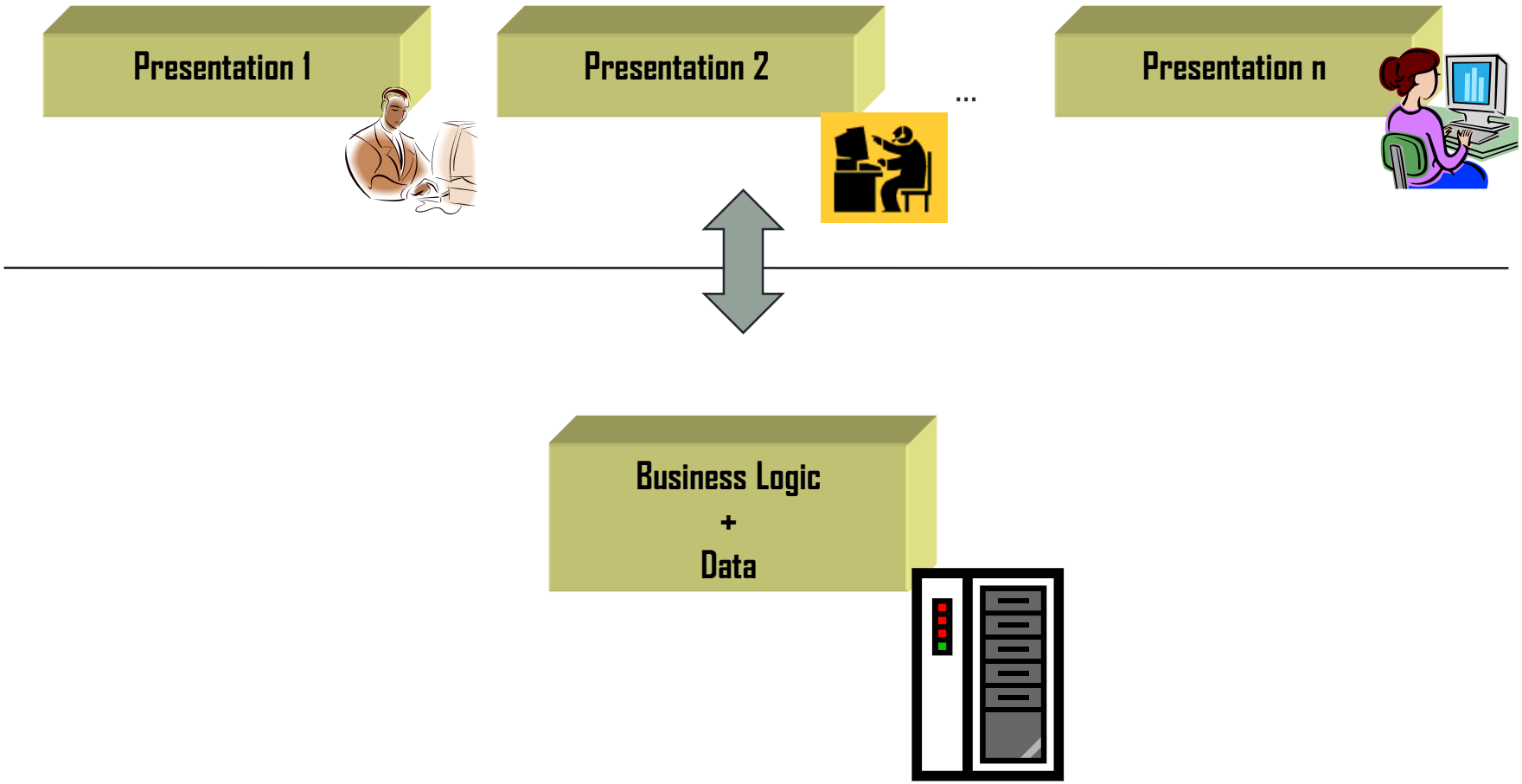
- Layer refers to a logical segmentation of the solution whereas tier refers to a physical segmentation or location.





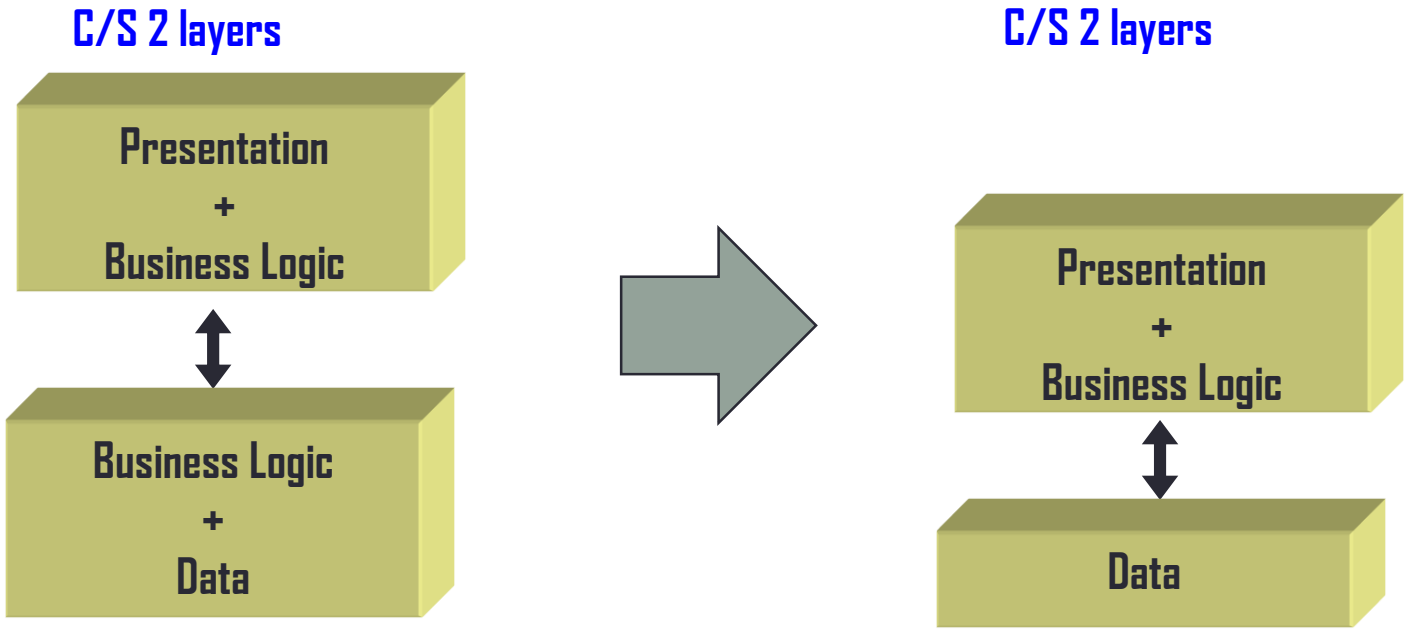
Architecture with 2 layers: Thin clients

1 Application – N platforms:



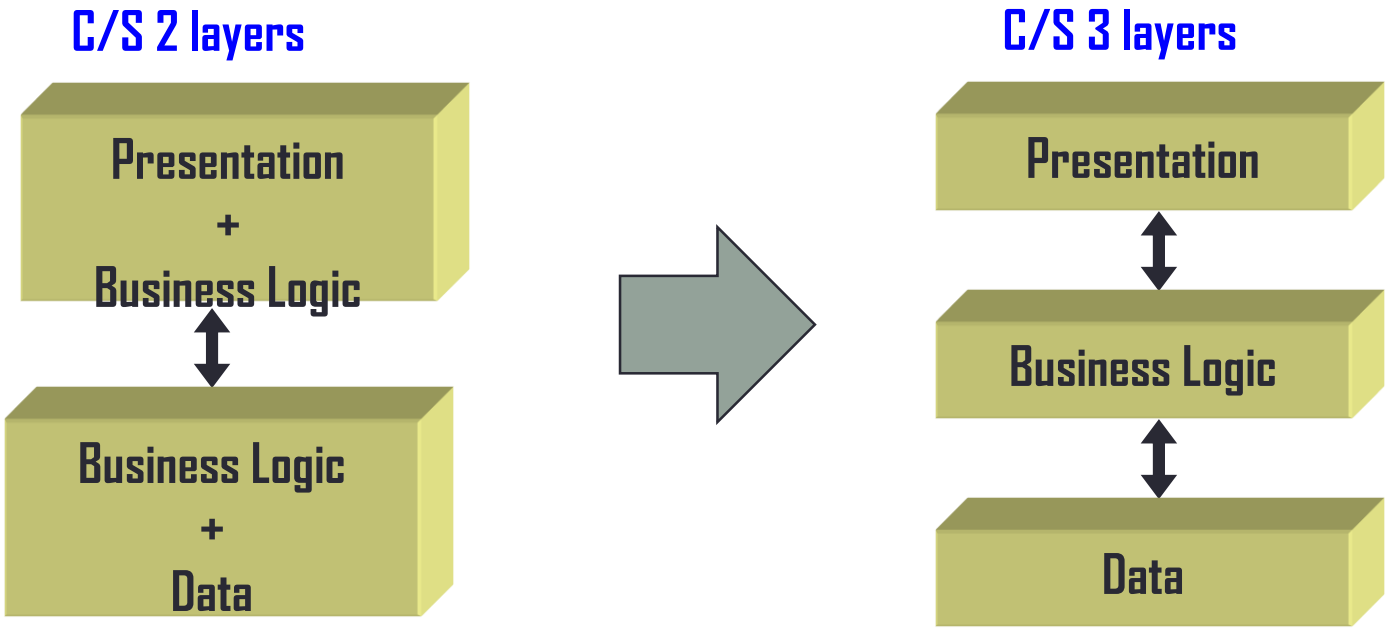
Architecture with 2 layers: Fat clients

Part of the logic (e.g. validations, business rules) is moved to the client



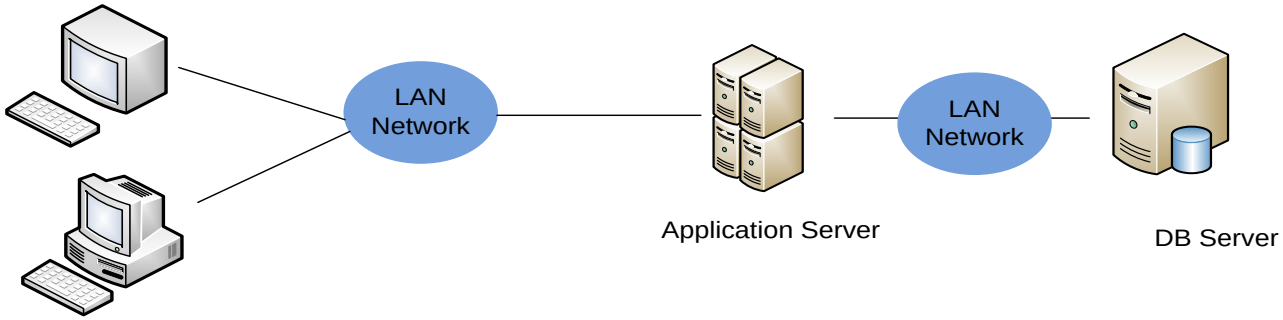
(Excel + Access)

Solution: 3 layers

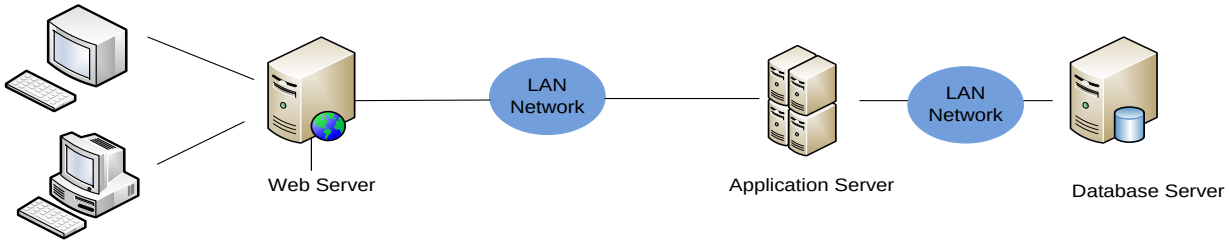


Client Server 3/N-tiers

- Architectures 3-tiers

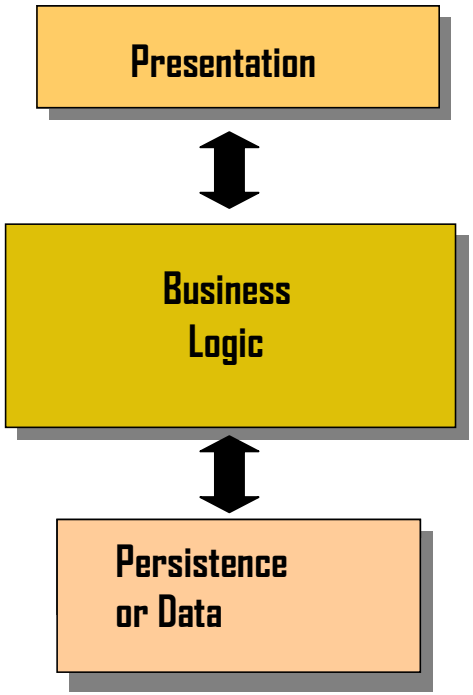


- Architectures 4-tiers



3-layered Architecture

(Generic)

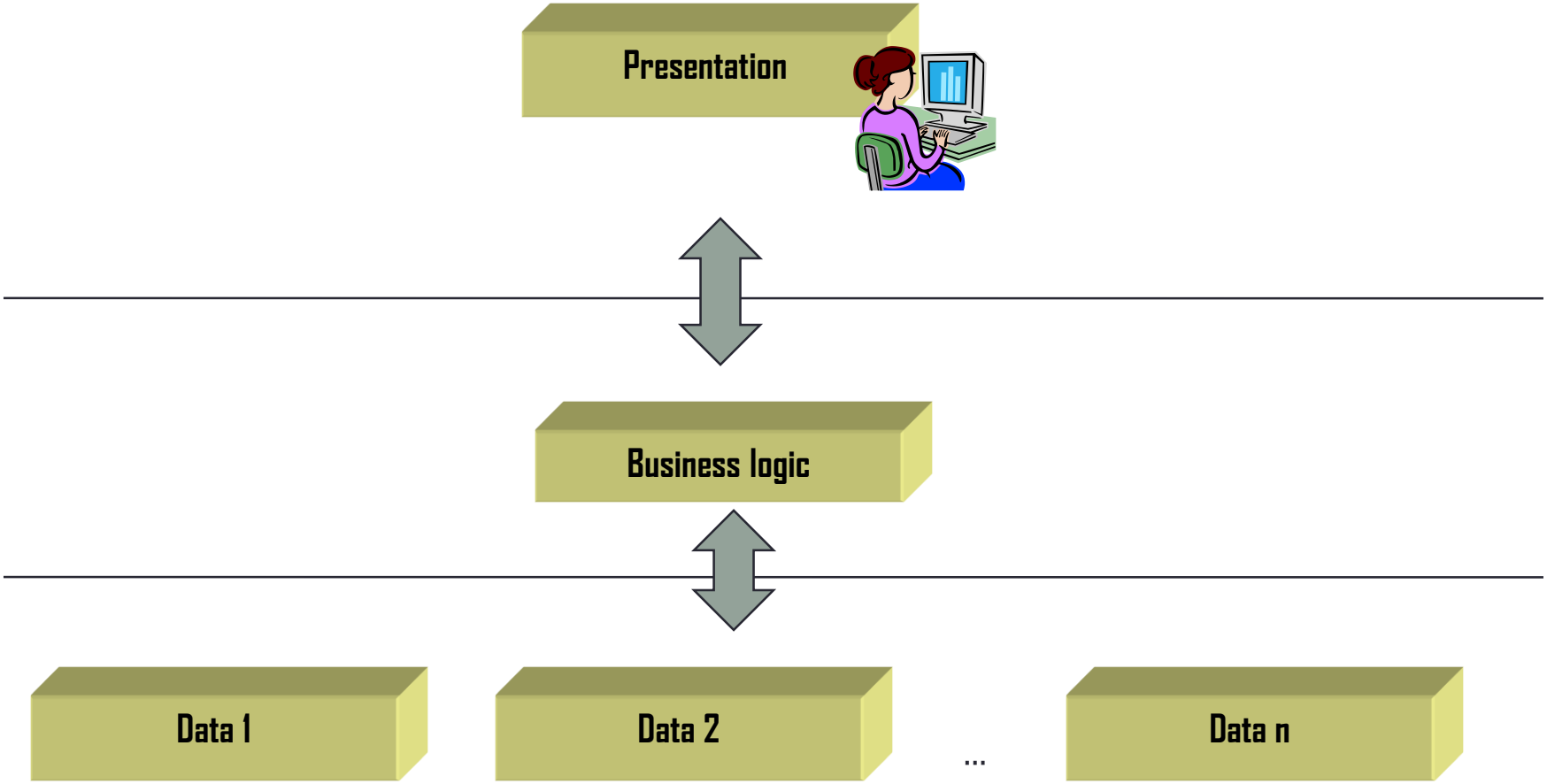


- **Presentation**
  - Presentation of computation results to the user and user input detection.
- **Business Logic**
  - Provide the functionality of the application
- **Data**
  - Provide persistence to data by means of databases or files...

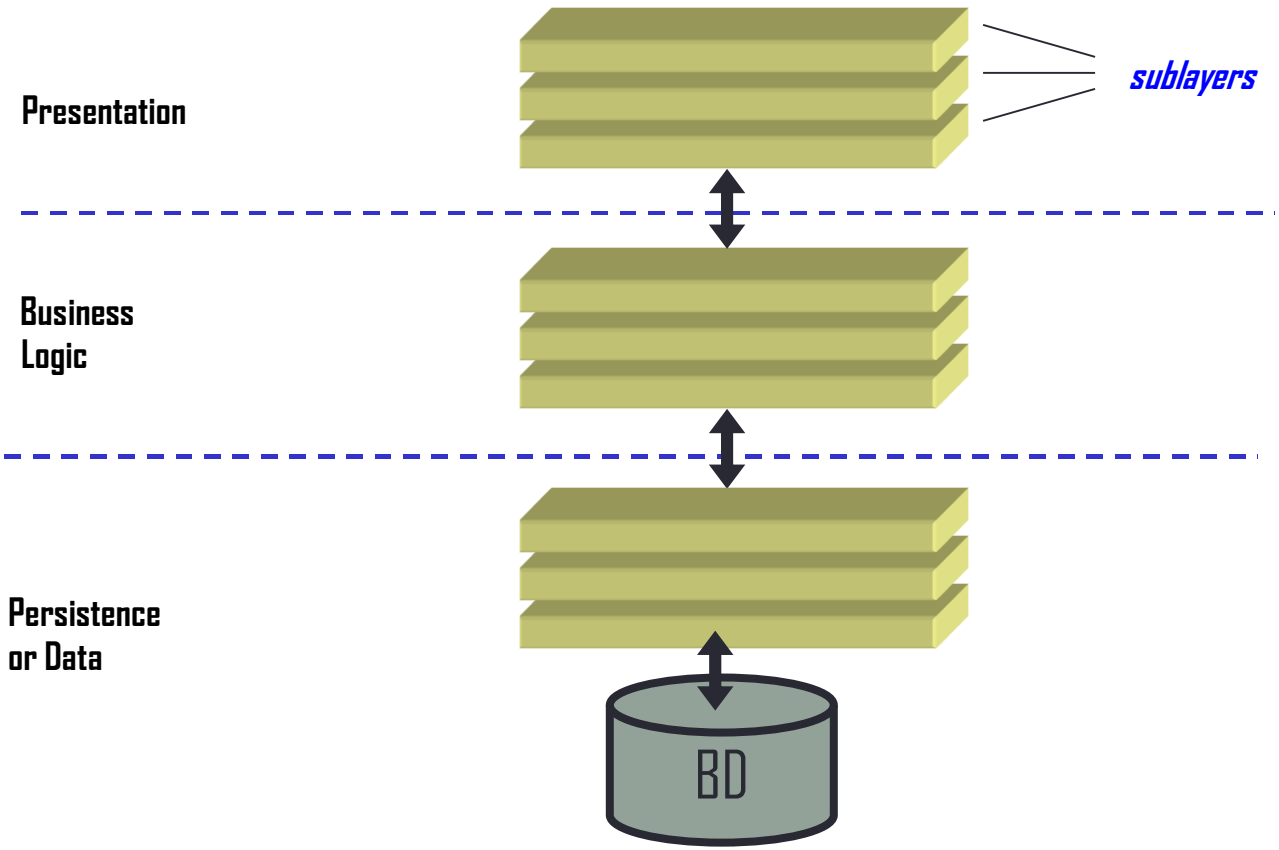
Advantages

- Isolate business logic in a separate component.
- Distribution of layers in different machines or processes.
- Possible parallel development.
- Assigning resources to each layer.
- SOFTWARE REUSE ...

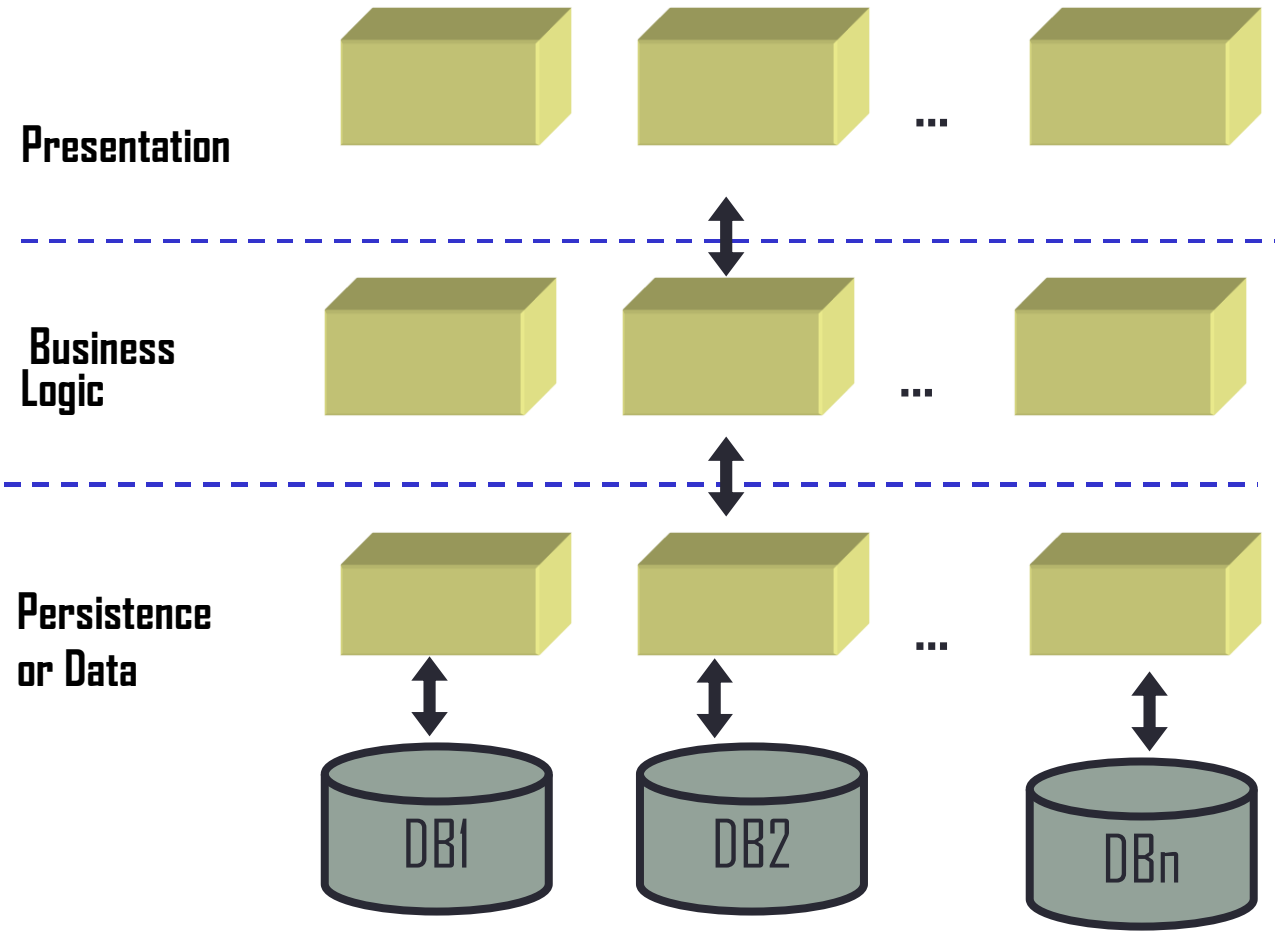
Advantages...



Three layered Architectures: variations



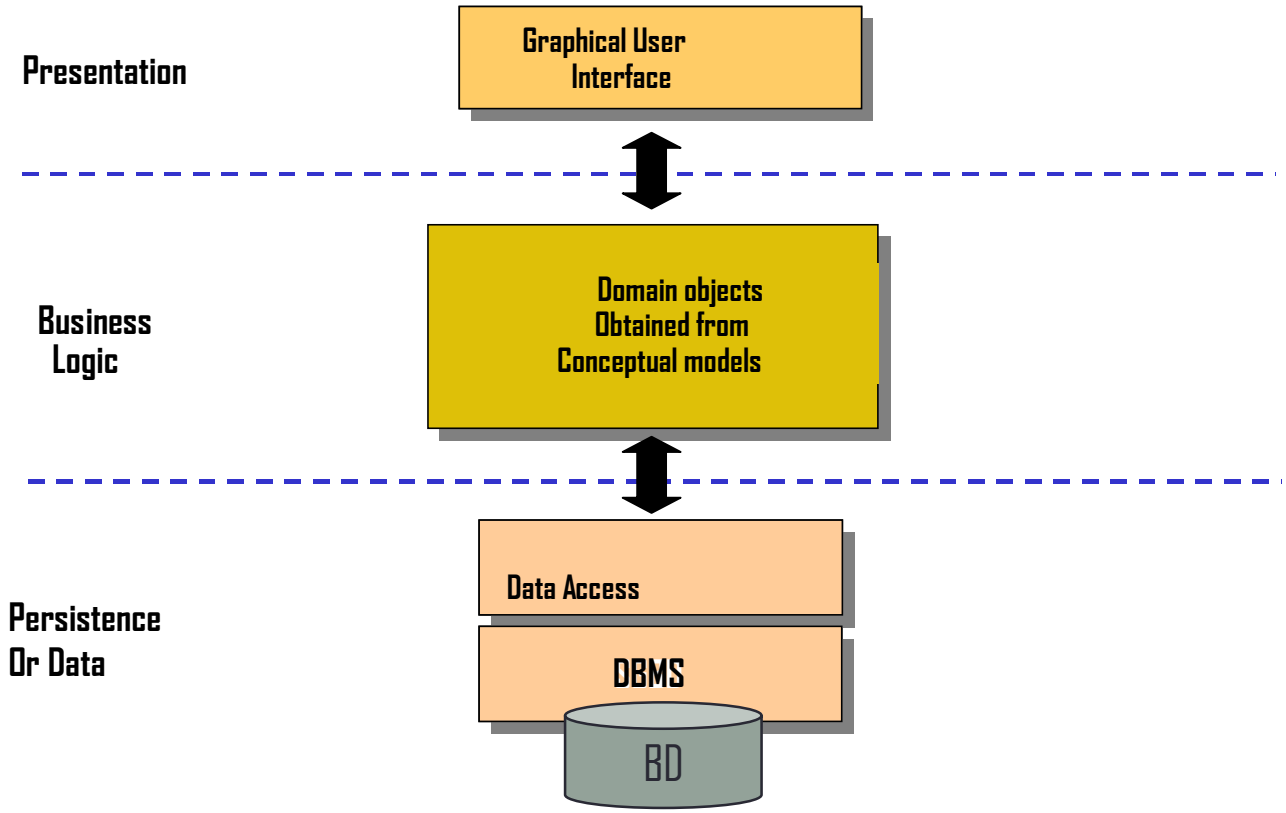
Three layered Architectures: variations



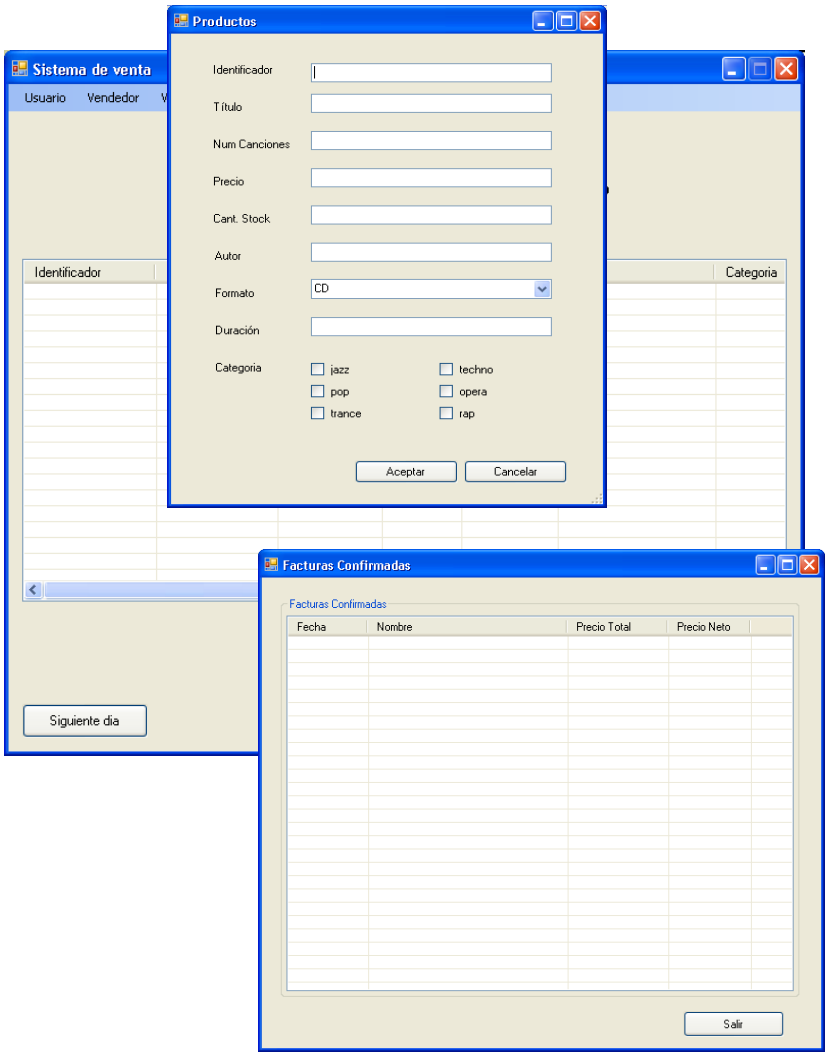
EXAMPLE

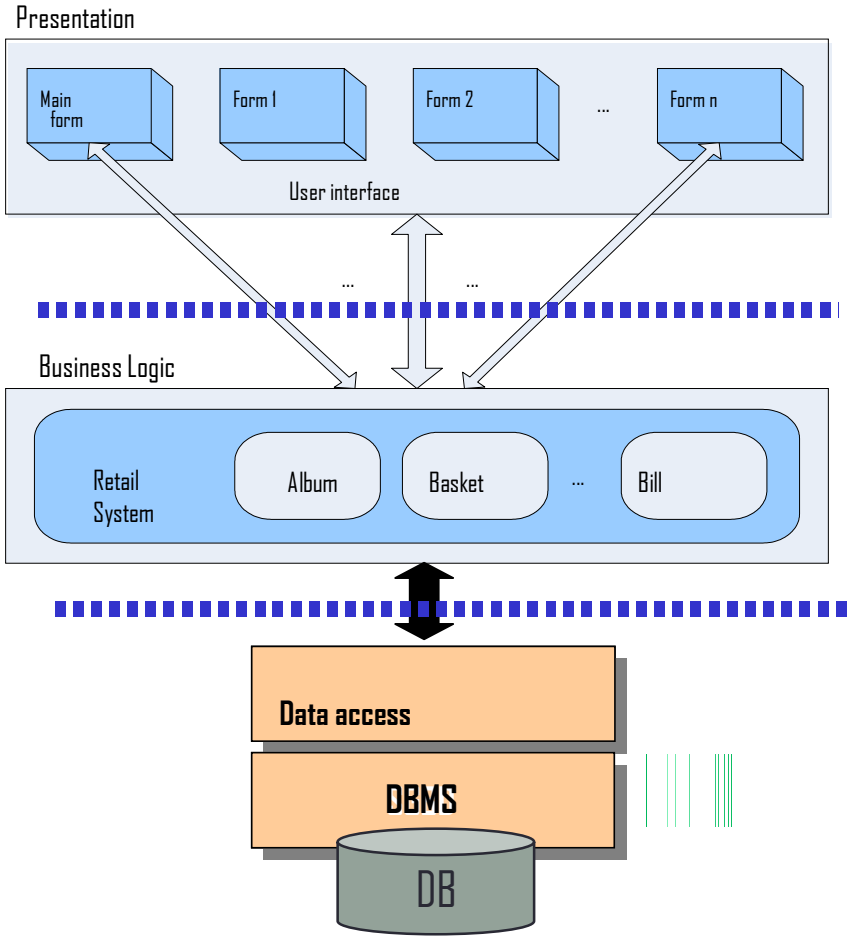
Three layered architecture: persistence

(Example)

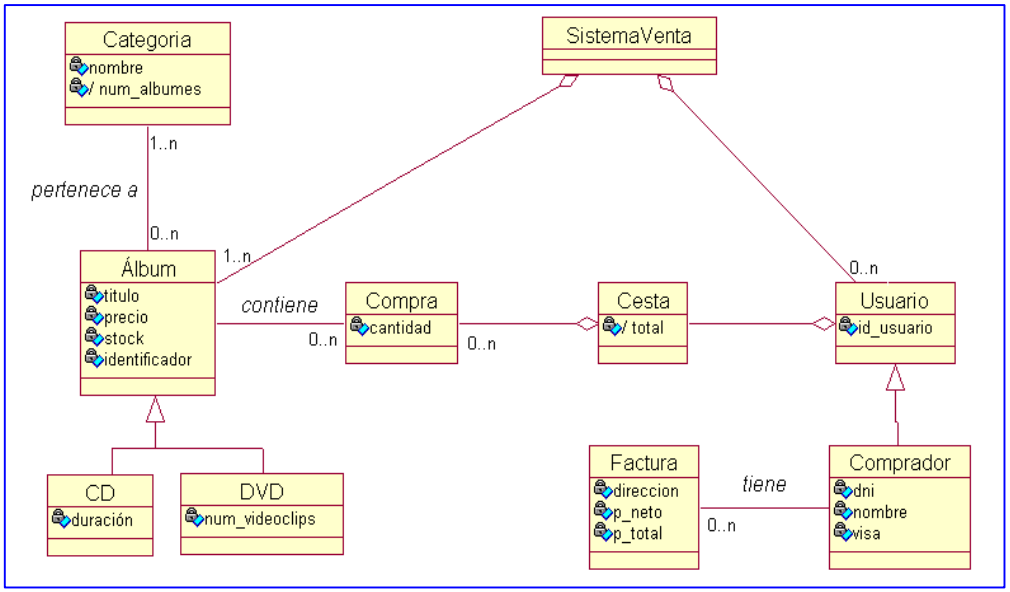


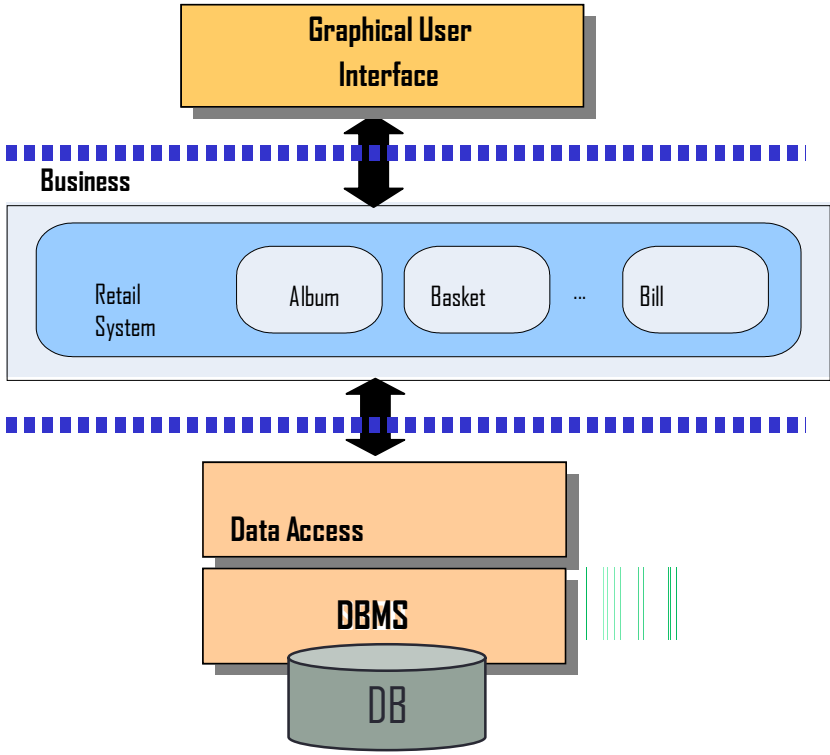
Three layered Architecture: Presentation





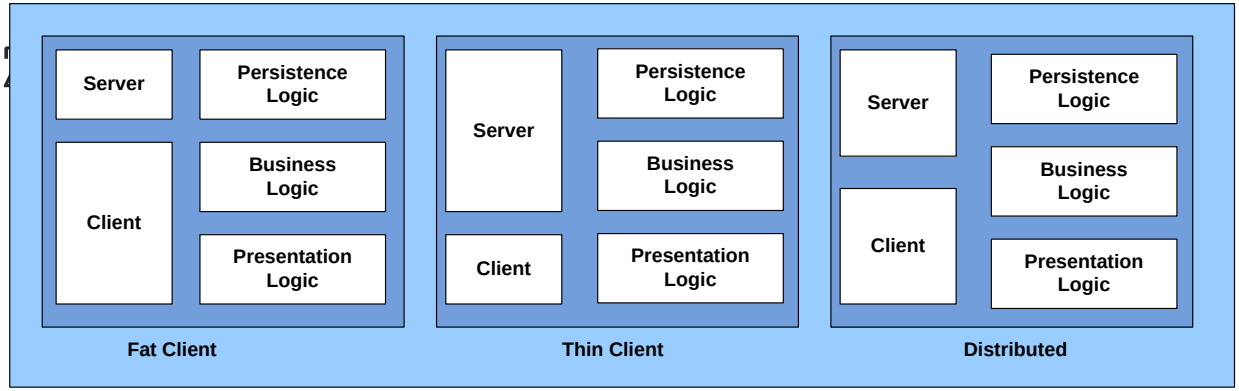
Three layered Architecture: Business Logic



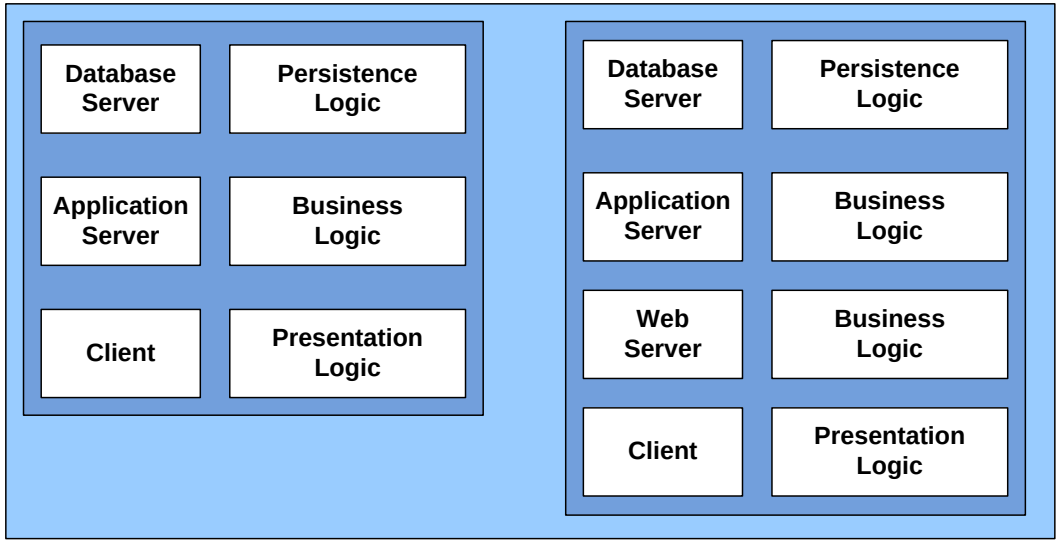


Summary: Business Logic distribution in architectures

-



- 3-n layers:



References

- Alonso et al., Web Services: Concepts, Architectures and Applications, Springer, 2004
  - Chapters 1 & 2
- Sommerville. “Software Engineering”. Chapter 12
- David Parnas, "On the Criteria to Be Used in Decomposing Systems Into Modules“. Communications of the ACM, December 1972.

CHAPTER 4: UML STRUCTURAL MODELS

Software Engineering

Computer Science School

Introduction

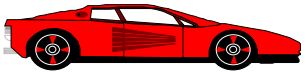
- The ***objects model*** or the ***classes diagrams*** reflect the **static structure** of the system.
- It is the **main tool** of most OO methods.
- An object model contains **interrelated classes** , by means of associations , organized as aggregation and generalization and specialization hierarchies.
- Sometimes it is useful to use diagrams that contain objects, ***instances diagrams***.

Introduction

- An objects is a concept, abstraction, or thing that makes sense within the context of an applications. It is an encapsulation of data and operations.

- Objects appear as names within the description of the problem or in discussions with users.

Introduction



Real World

House

Person

Car

Semantic Gap

(House)

Lives in

(Person)

drives

(Car)

(Equation)

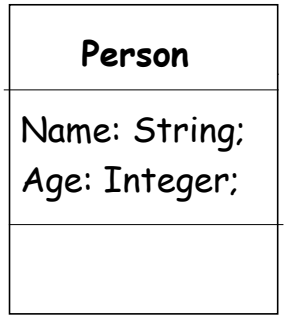
(Binary Tree)

Model

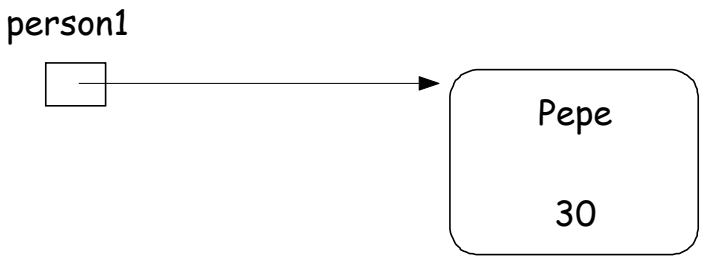
Introduction

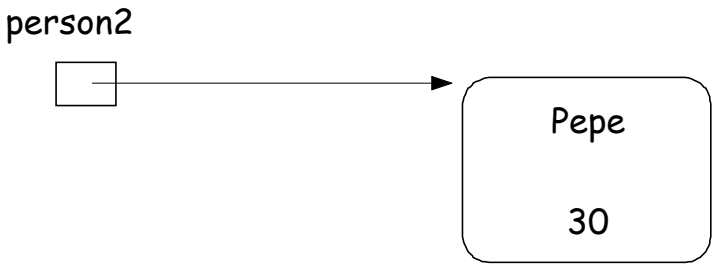
- The object **identifier** (oid) is a feature to differentiate two occurrences that have the same state (i.e: same values for attributes).

- During analysis it is assumed that objects have an identity. During implementation there must be an identification mechanism:
  - Memory addresses: programming languages.
  - Comination of attributes values: databases
  - Unique names (“surrogates”):some OO systems.



Programming Language

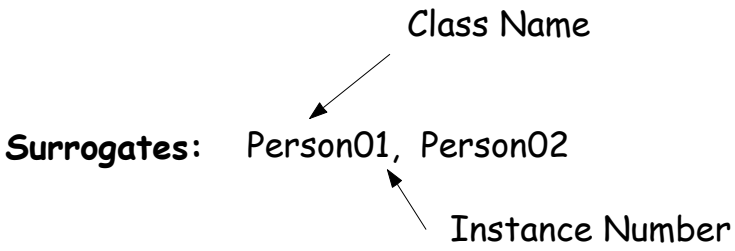




Relational Database

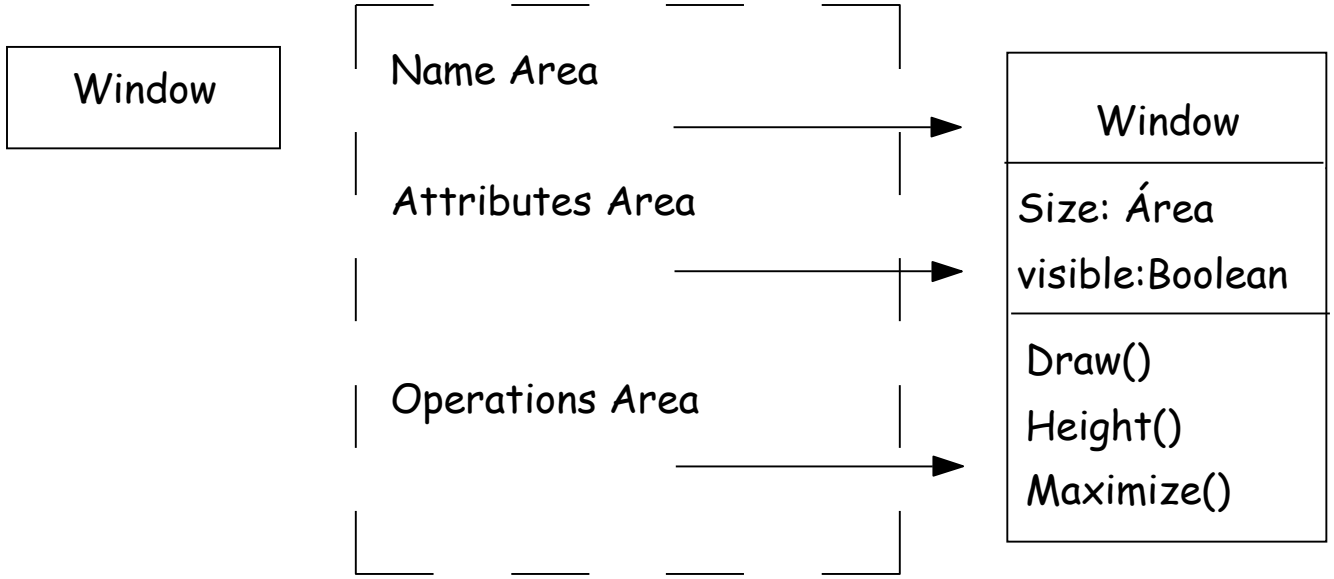
Table Person

ID Person	Name	Age
1	Pepe	30
2	Pepe	30



Classes

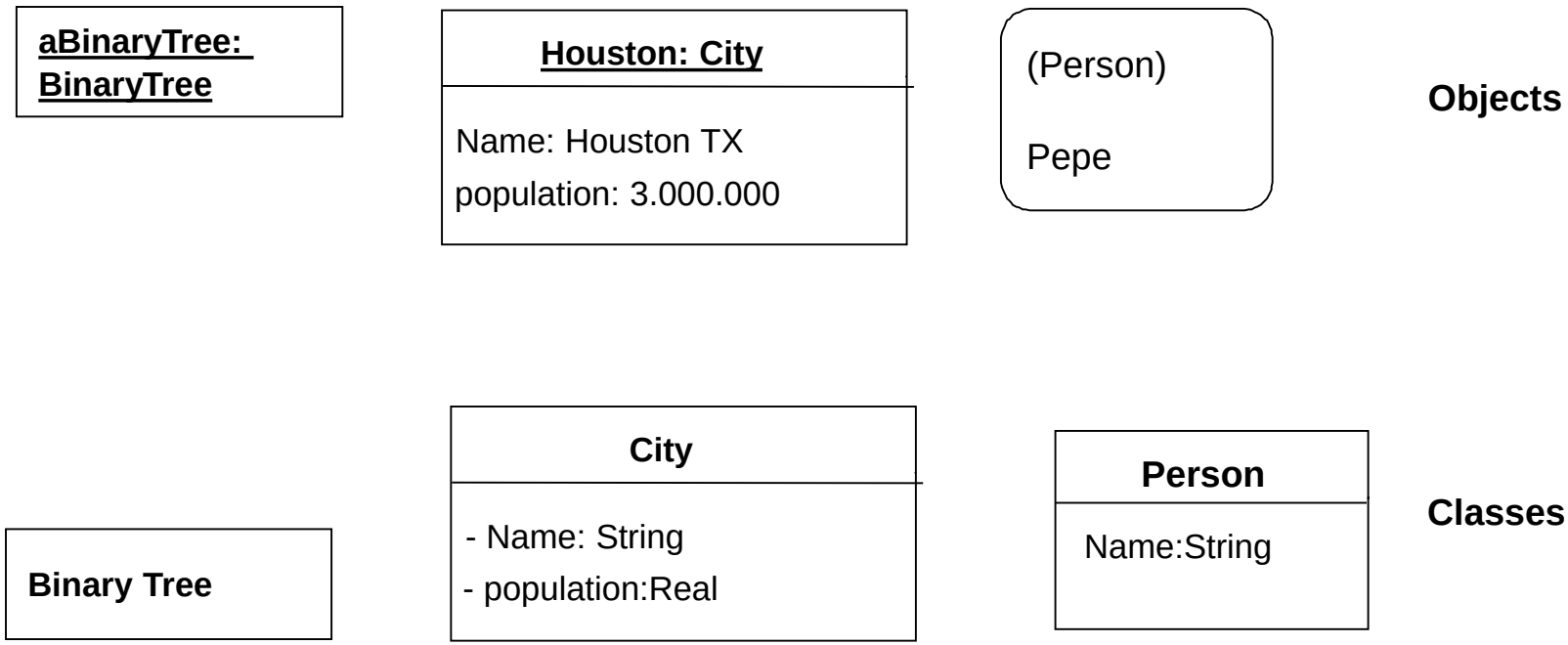
- A class is the description of a group of objects with similar structure, behavior and relationships.



- An additional area may be added to define responsibilities, description, etc.

Classes

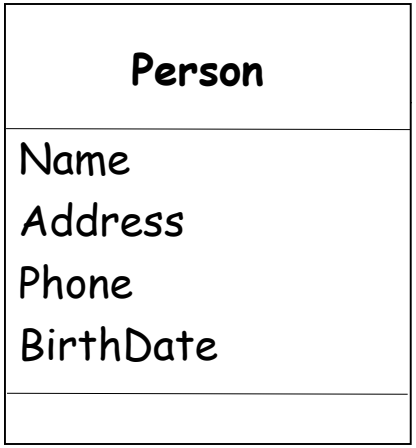
- A class is an abstraction; an object is a concrete realization of this abstraction.



Attributes

- An attribute is a property of a class identified with a name and describes a range of values.

- Attributes may be represented showing only their names.



- The general definition schema is:

  [visibility] Name [: Type] [=initial value]

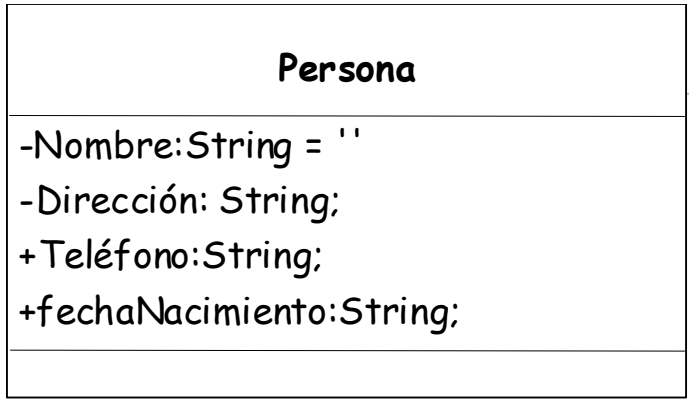
  where visibility may be:

  + = Public

  # = protected

  - = Private (default)

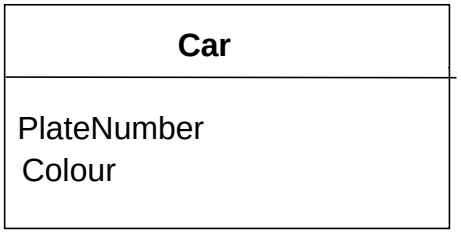
  = implementation or package



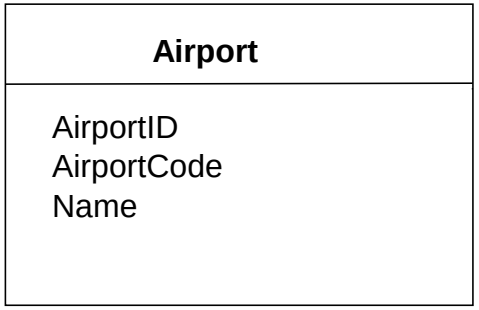
- Allowed types are (integer, real, char, string, etc.), no object types.

- Attributes do not include references to other objects, these references are represented as links.

- In the objects model attributes acting as objects identifiers should not be present



Good for analysis and design



Bad analysis, good for design

Operations

- An operation corresponds to a service that may be required to any object of the class.
- An operation is a function or transformation that may be applied to objects.
- A method is the implementation of an operation.

- Operations are defined as follows:

[visibility] Name([parameters]) [: return_type]

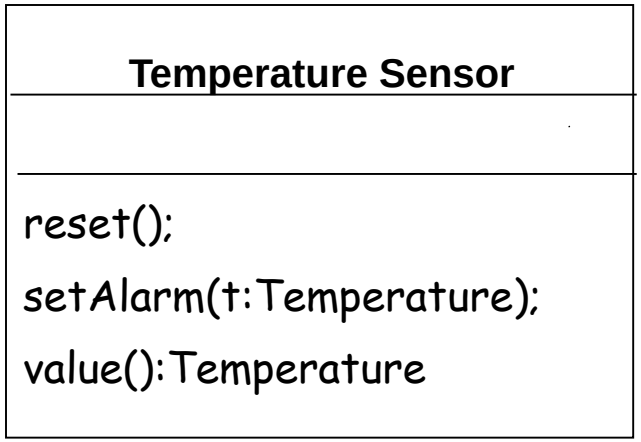
Where visibility is:

+ = public (default)

#= protected

- = private

= package

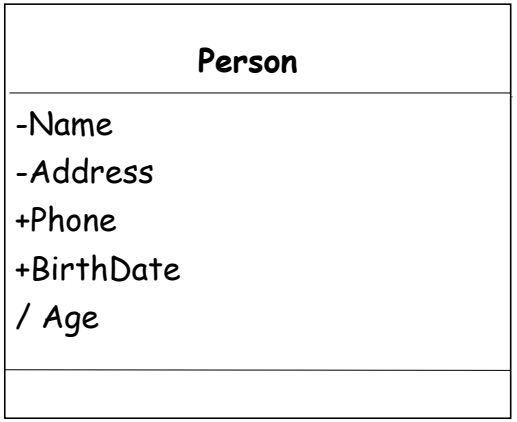


- Operations that change the state of objects are defined as having side effects.
- Operations that do not have side effects and just calculate a functional value are called queries.
- Queries return the values of attributes.

- An attribute is derived if its value is obtained in terms of the values of other attributes.

- The notation for derived attributes is:

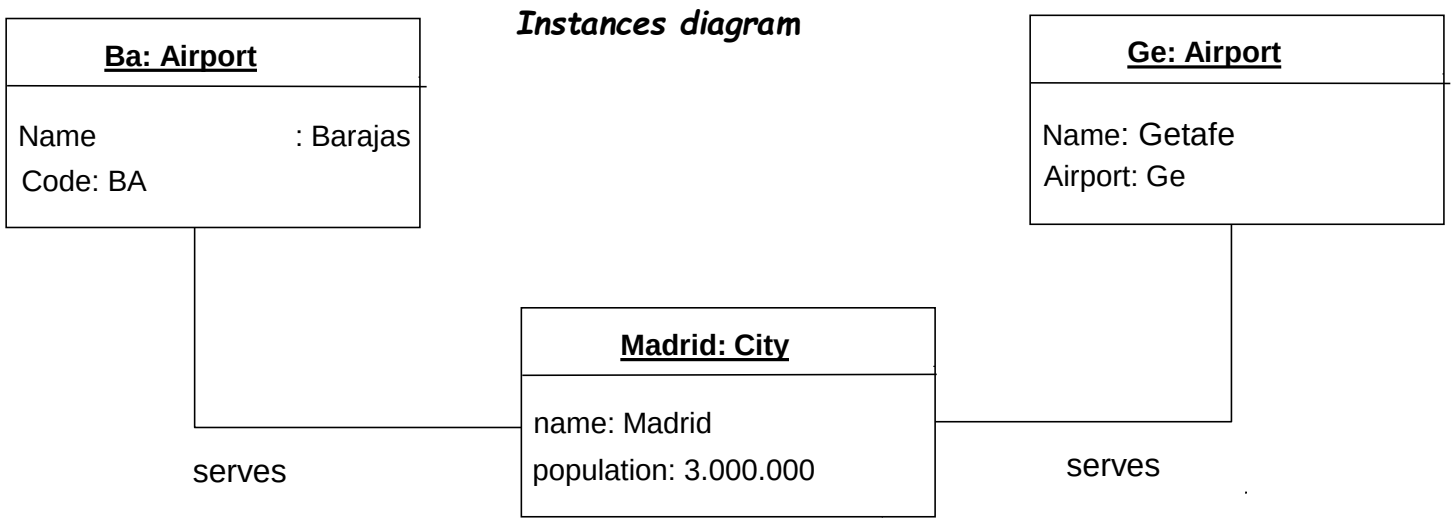
/Attribute_Name: Type

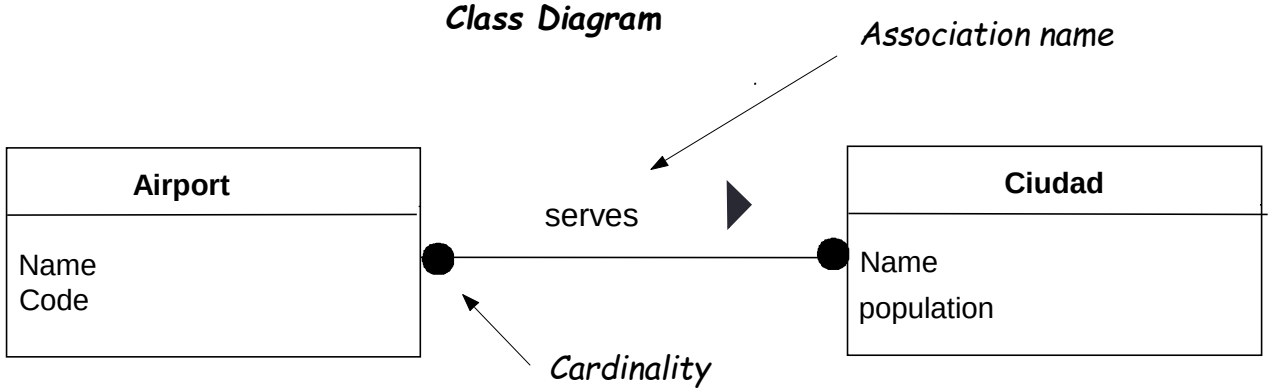


Associations & links

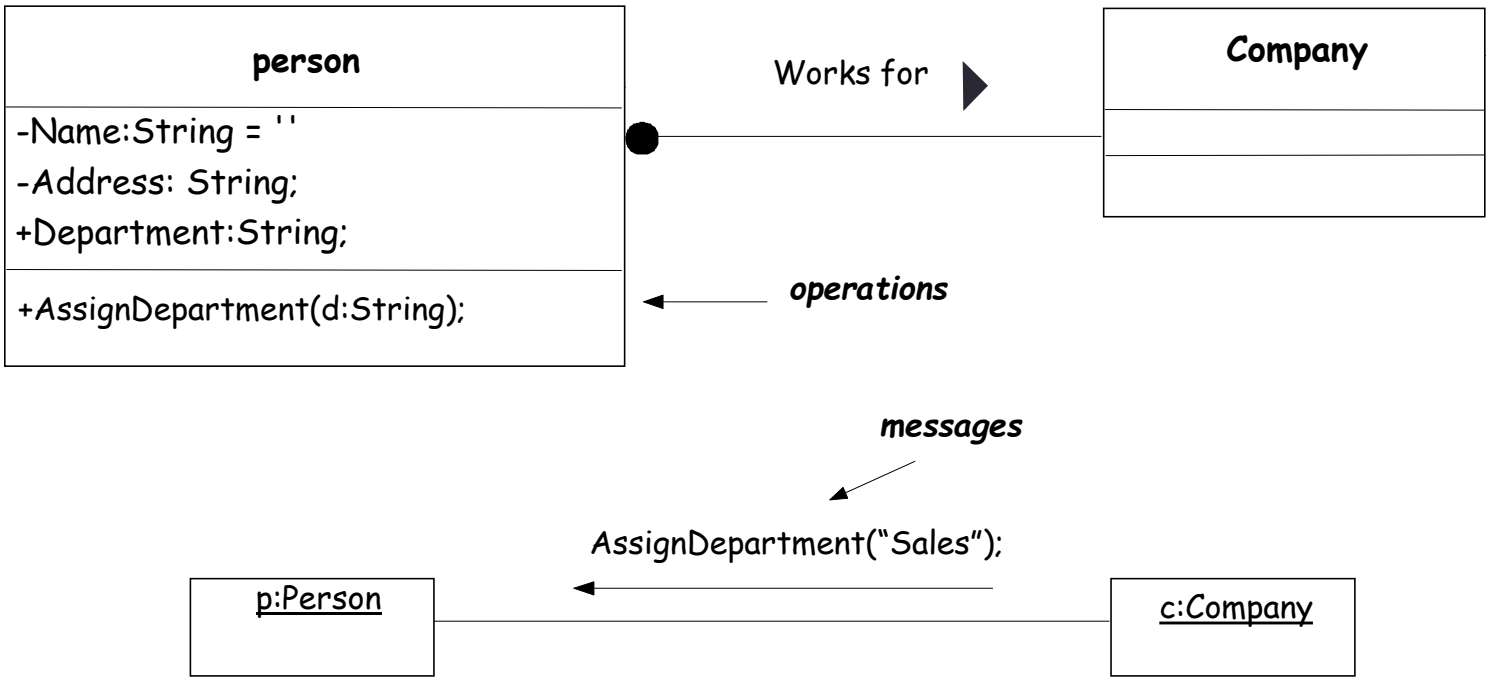
- A link is a physical or conceptual connection between objects.
- An association is a structural relationship to show that the objects of an element are linked to the objects of another element.
- Associations are represented in class diagrams whereas links appear in instances diagrams.

- Each association in a class diagram corresponds to a collection of links in the instances diagram.
- Given an association between two classes, it may be navigated in both directions.
- Binary associations are those connecting two classes.


Instances diagram
Ba: Airport
Name : Barajas
Code: BA
Ge: Airport
Name: Getafe
Airport: Ge
Madrid: City
name: Madrid
population: 3.000.000
serves
serves

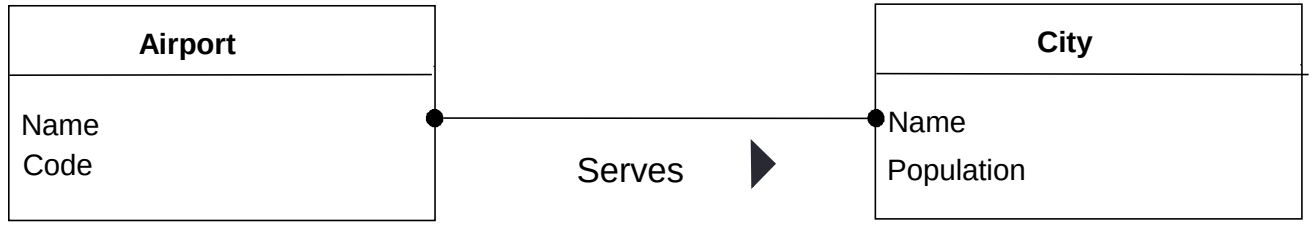

Class Diagram
Association name
Airport
Name
Code
serves
Ciudad
Name
population
Cardinality

- During analysis, references to objects are represented as links. Similar links are represented as associations.

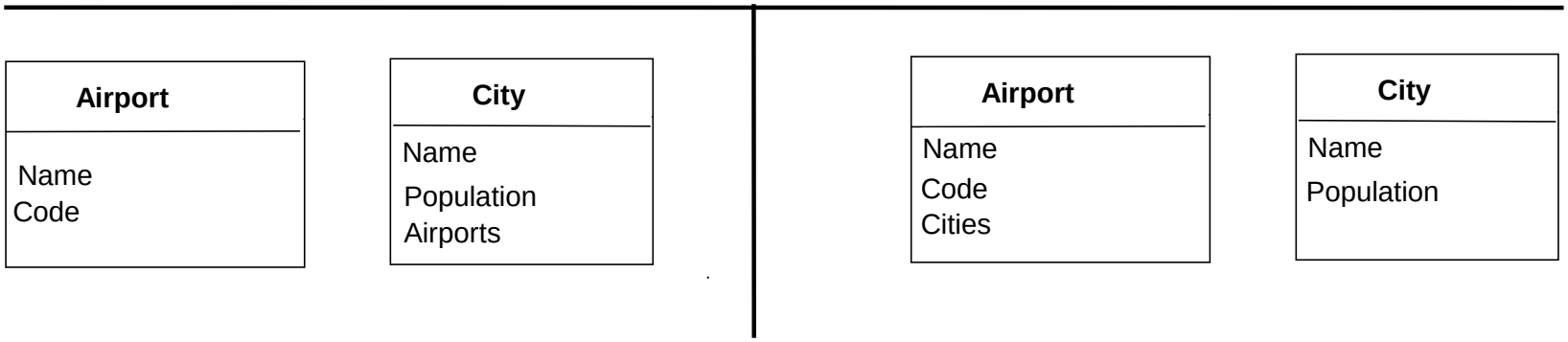


- During design associations are represented as pointers, foreign keys, etc.

- At the conceptual level there is no difference between an association and an attribute.
- At the implementation level the semantics is different:
  - Attributes:
    - Value semantics
    - Type: Date, Address, Integer, Real, String,..
  - Associations:
    - Referential semantics

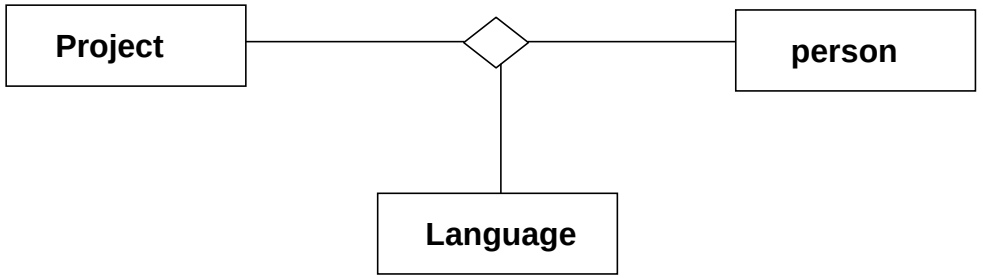


Analysis Model

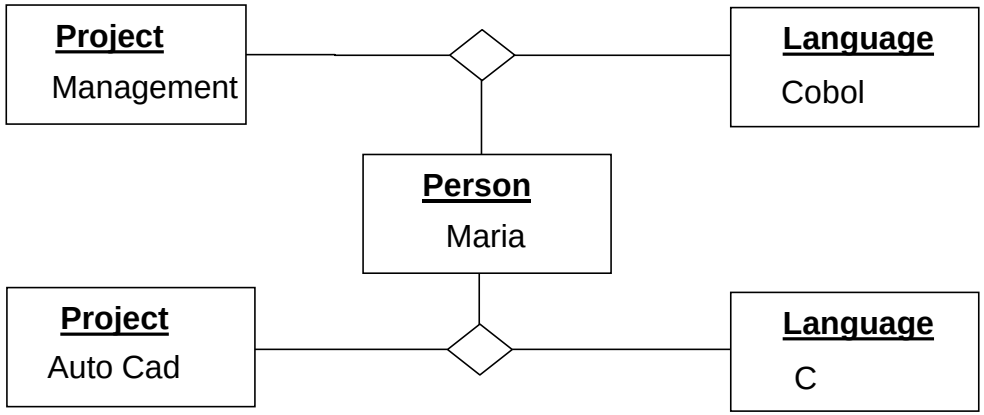


Design Models

- Associations may be of any order (2, 3, …).



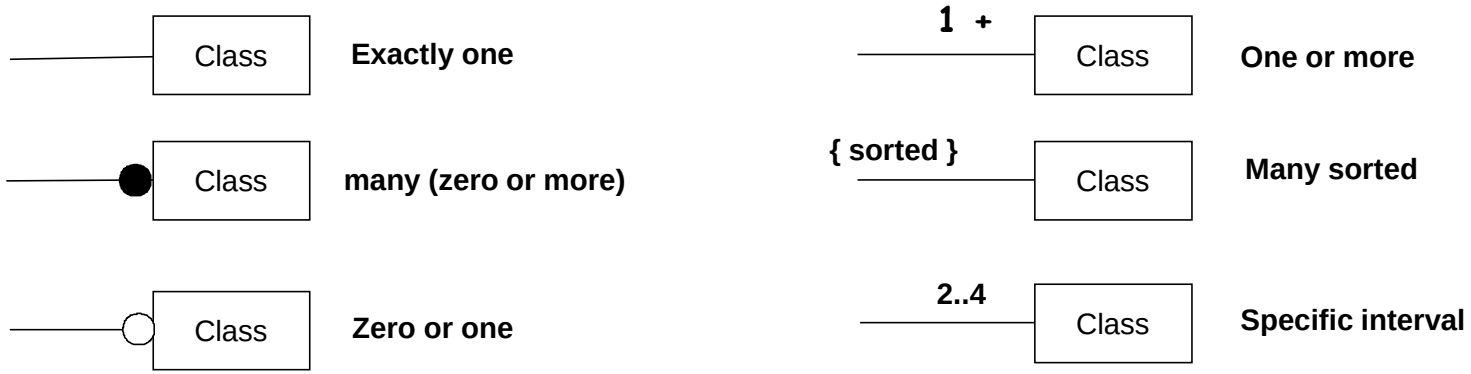
Associations



links

Cardinality

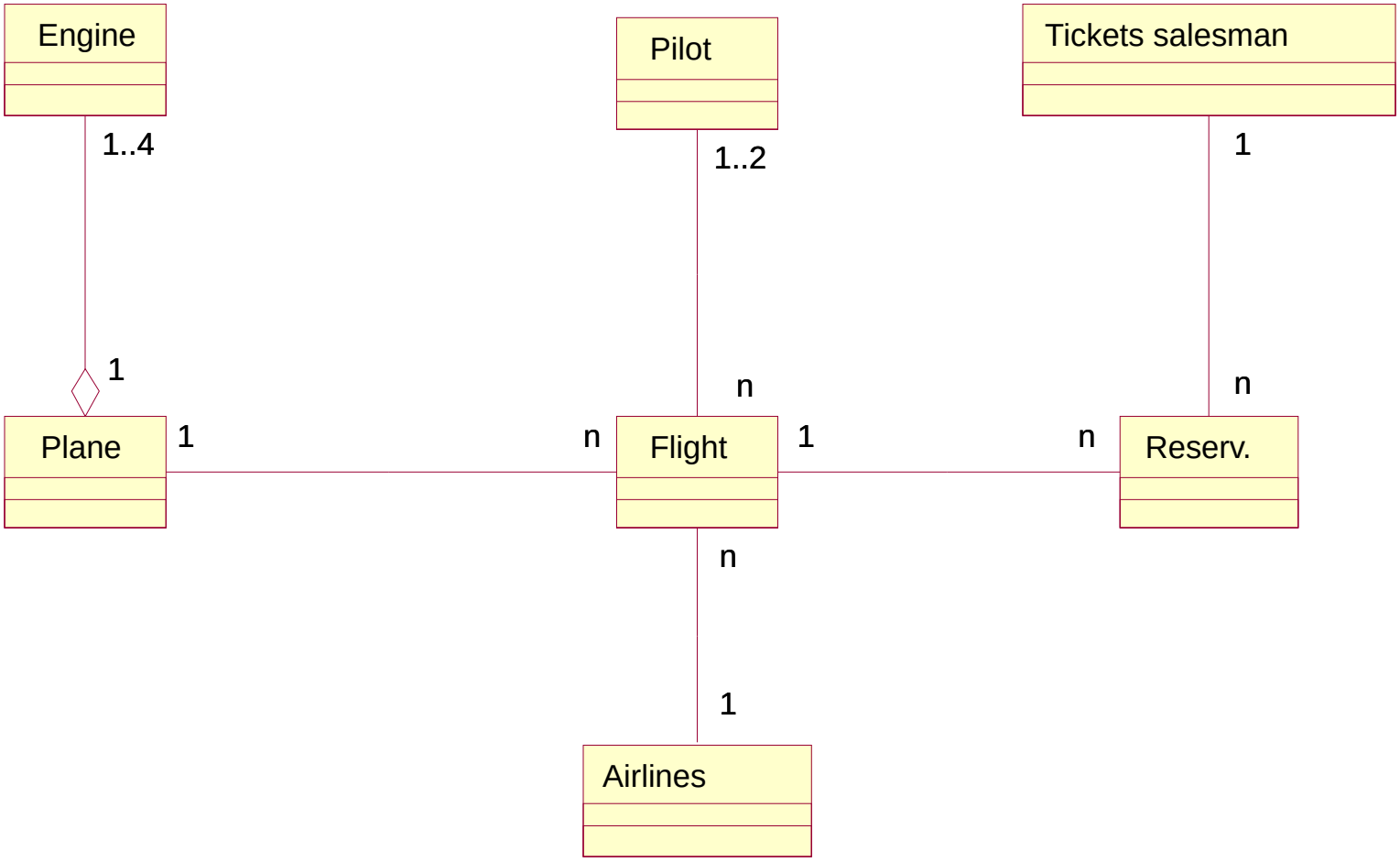
- The cardinality defines the number of instances of another class that may be related to an instance of a given class.



Cardinality in OMT

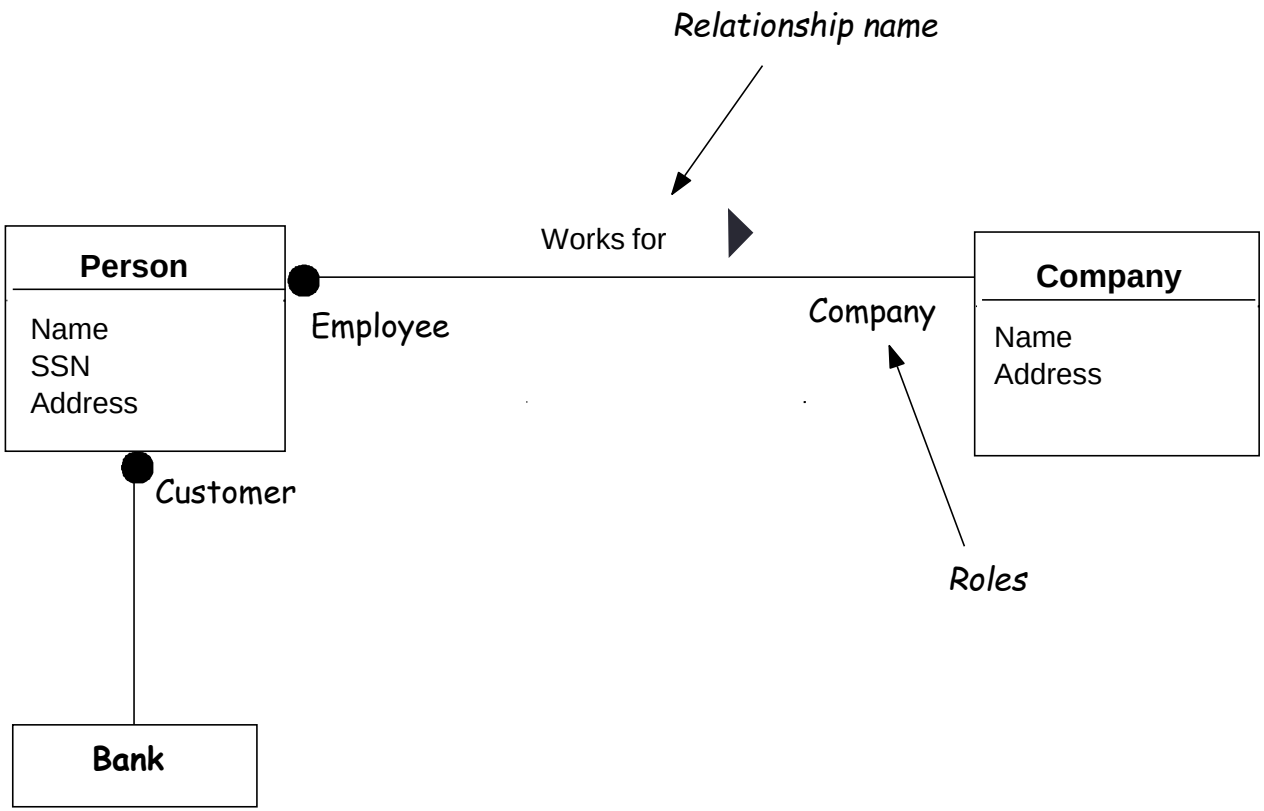
1	One and only one
5	Exactly five
0..1	zero or one
M..N	from M to N
*	from 0 to many
n	from 0 to many
0..*	from 0 to many
1..*	from 1 to many

Cardinality in UML


Engine
Pilot
Tickets salesman
1..4
1..2
1
1
n
n
Plane
1
n
Flight
1
n
Reserv.
n
1
Airlines

Roles

- Roles are names that define the role played by a class in a relationship.

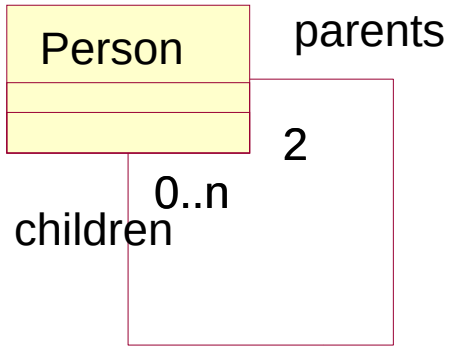


- Roles are used to navigate associations.

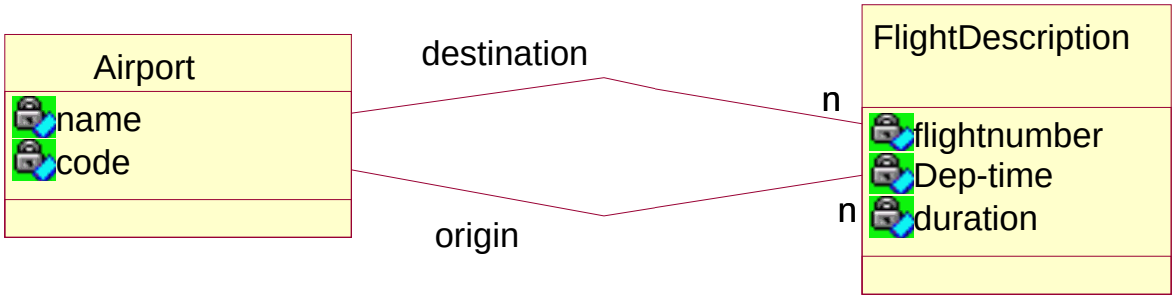
  `aCompany.employee`

  `aperson.company`

- They are mandatory to distinguish reflexive associations.

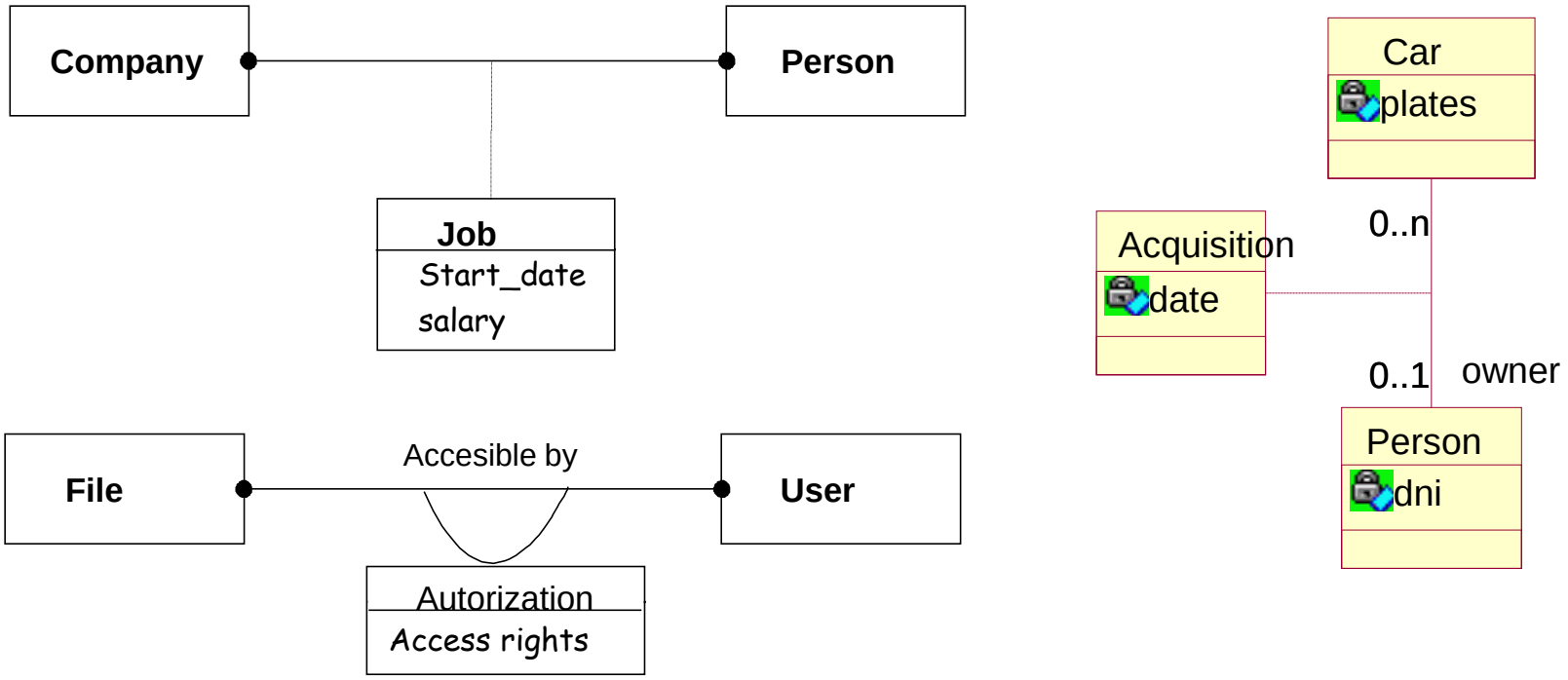


- They are mandatory to distinguish associations between same pair of classes.



Associations as classes

- In an association between two classes the relationship itself may have attributes. These are represented as association classes.



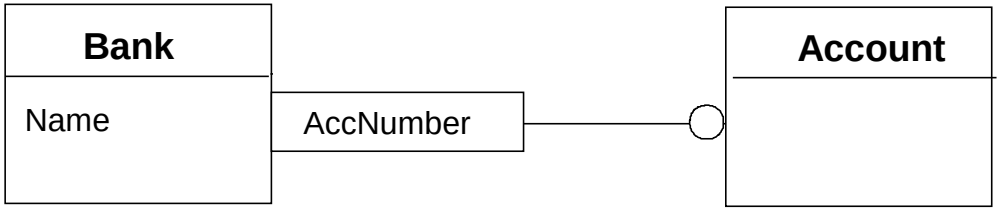
Qualified Associations

- Qualifiers in associations are used to resolve searches: given an object on one side of a relationship,how to identify an object or collection of objects on the other side?
- They act as indexes to navigate the relationship.

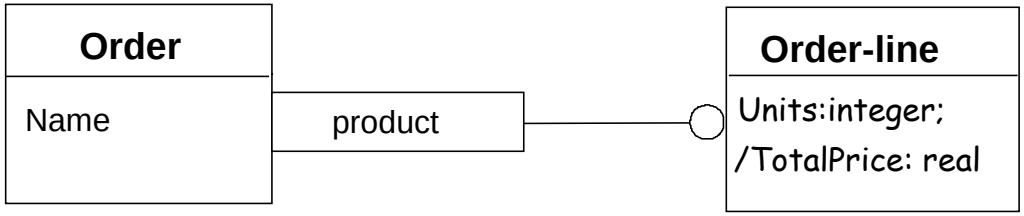
  `Bank + account number` → `one account`

- A qualifier may be used whenever that a data structure (hash table, binary trees) can be used to perform searches on one side of the association.

Not qualified

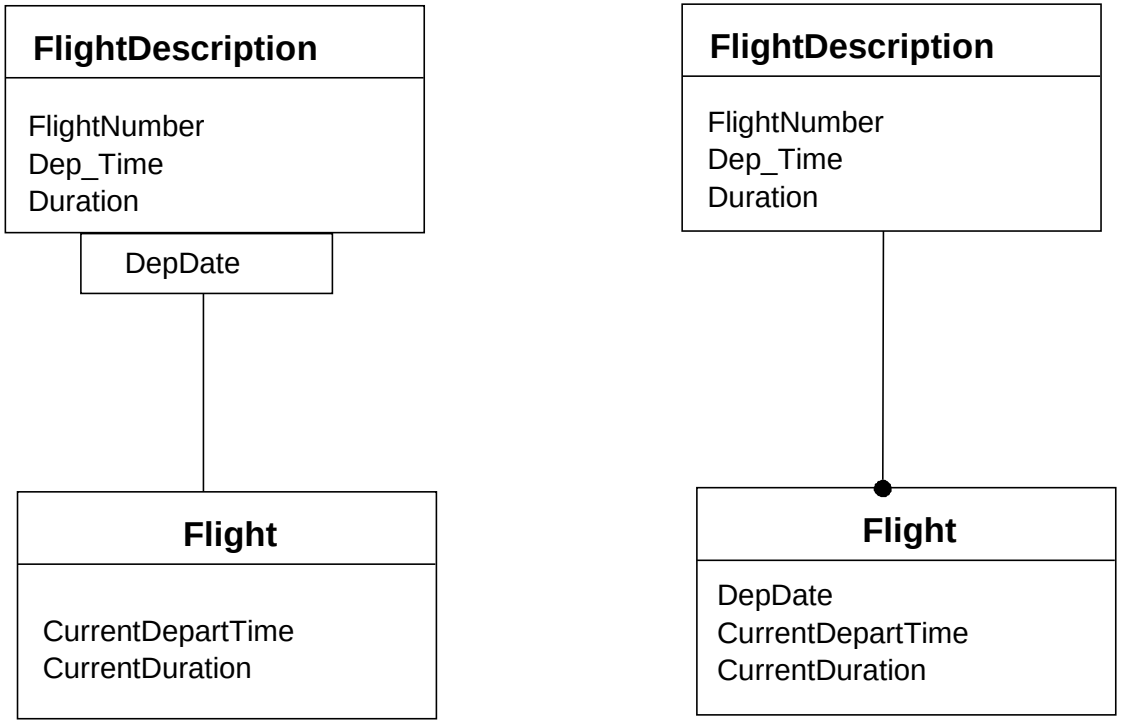


Qualified



Qualified

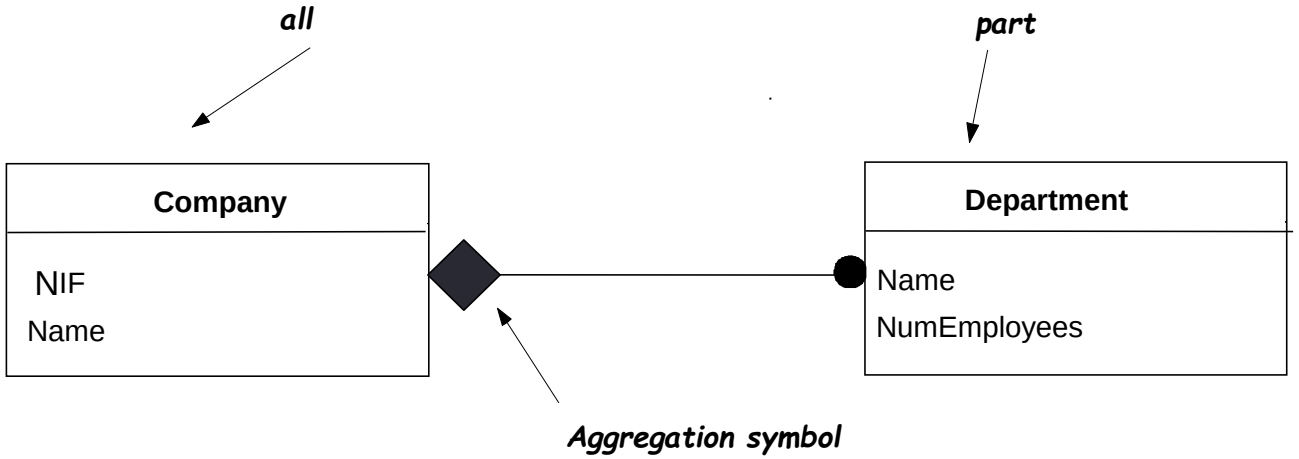
- Qualifiers increase the precision of models.



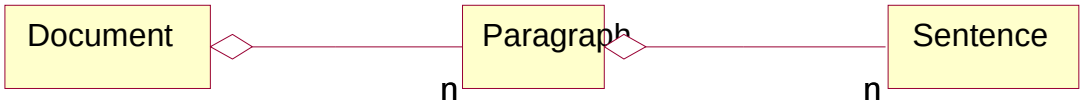
Aggregation

- An aggregation is a type of relationship with additional semantics.
- This relationship is used to model the semantincs “ **part_of consists_of** ”.

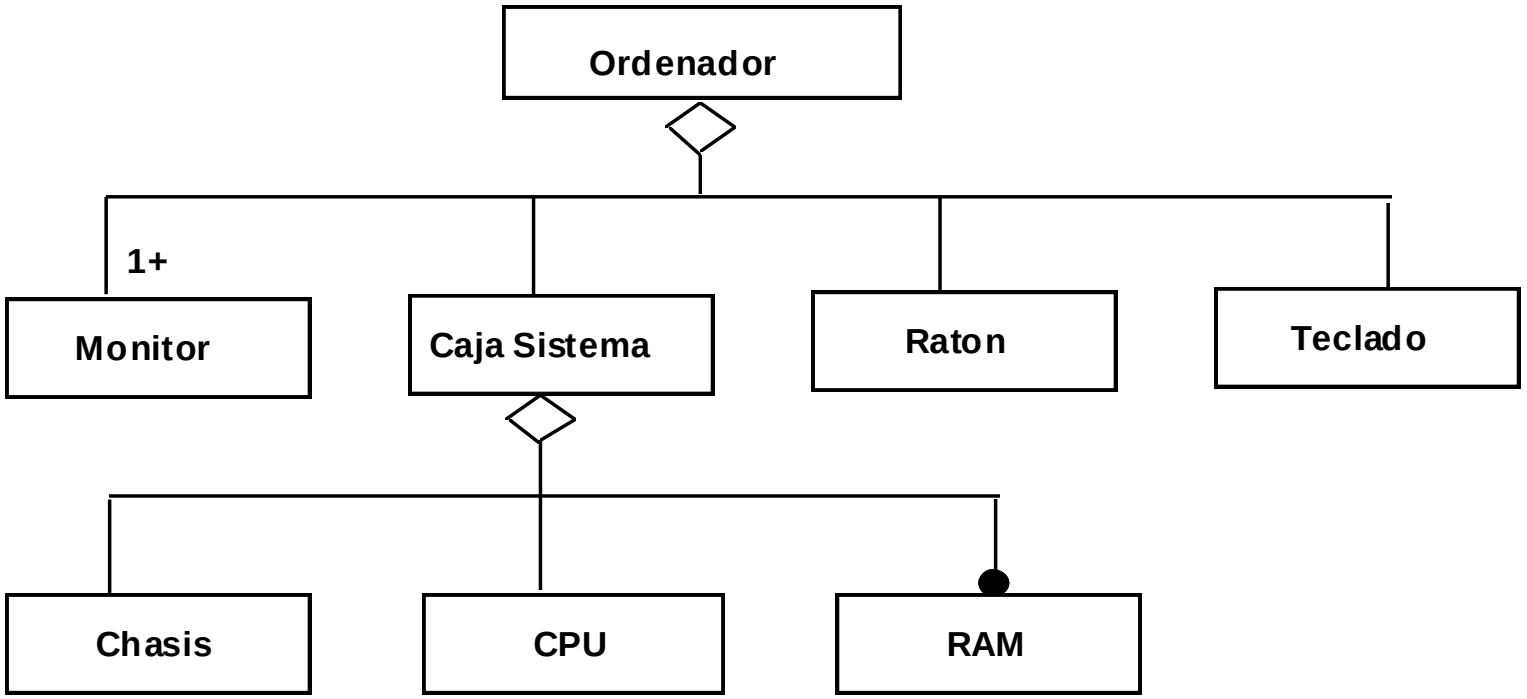
"A company consists of departments"



- The properties of the aggregation are:
  - **transitive** (if A is part of B and B is part of C then A is part of C)
  - **antisymmetric** (if A is part of B then B may not be part of A).

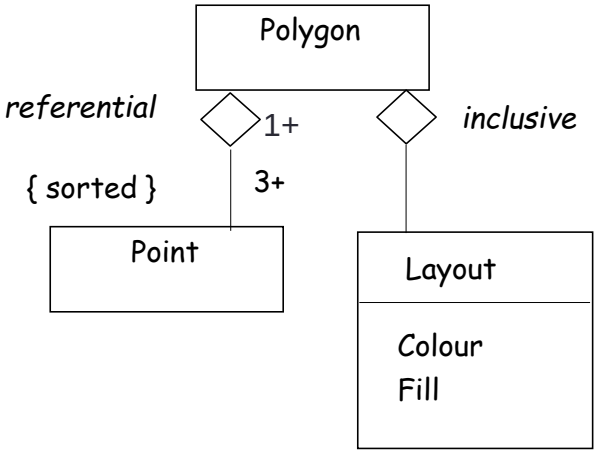


- There may be several nesting levels.

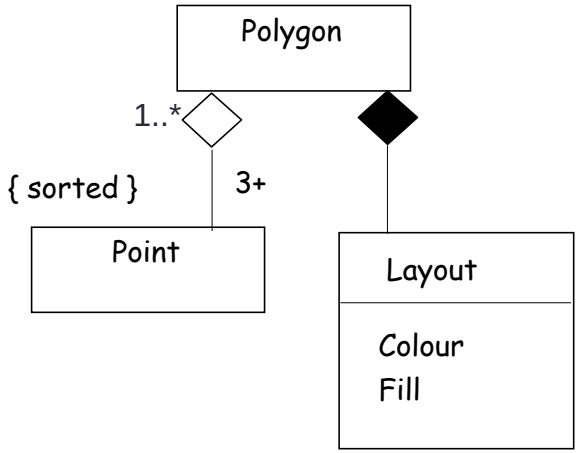


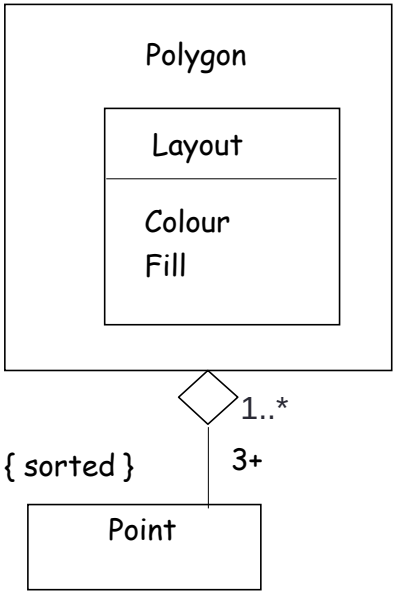
- There are two types of aggregations.
  - ***Inclusive or physical:*** each component may belong at most to one container. The destruction of the container implies the destruction of its parts.
  - ***Referential or catalogue:*** components may belong to several containers. Their lifetimes are not correlated.

OMT

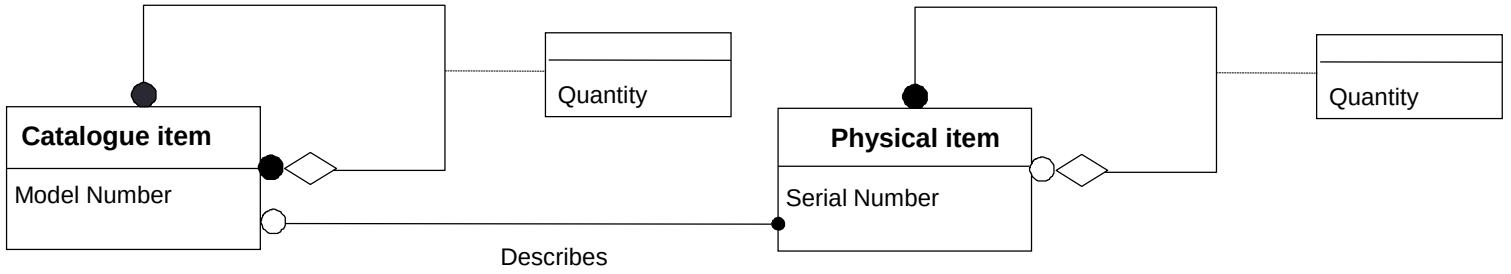

Polygon
referential
1+
inclusive
{ sorted }
3+
Point
Layout
Colour
Fill

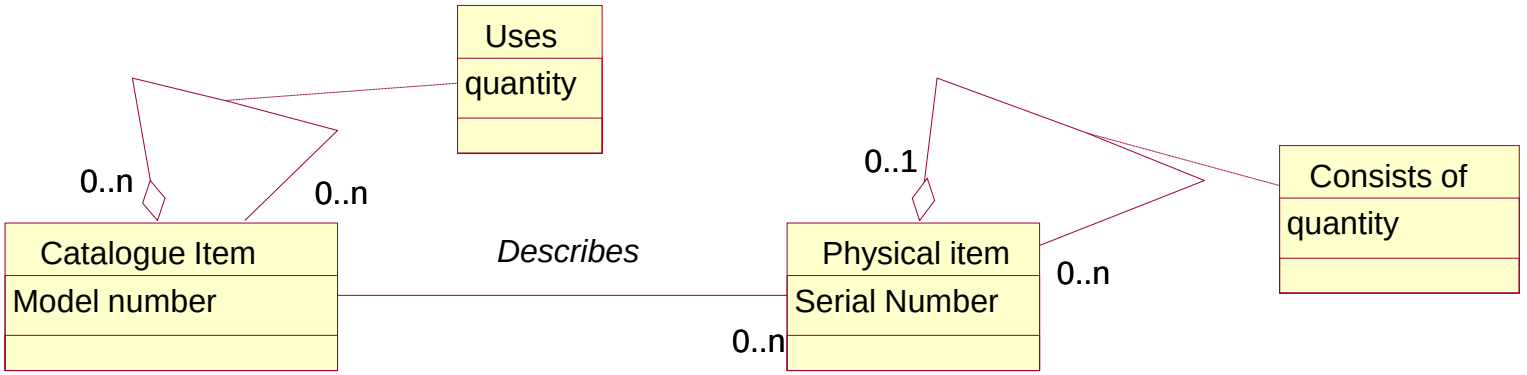
UML


Polygon
1..*
{ sorted }
3+
Point
Layout
Colour
Fill


Polygon
Layout
Colour
Fill
1..*
{ sorted }
3+
Point

referential or inclusive?



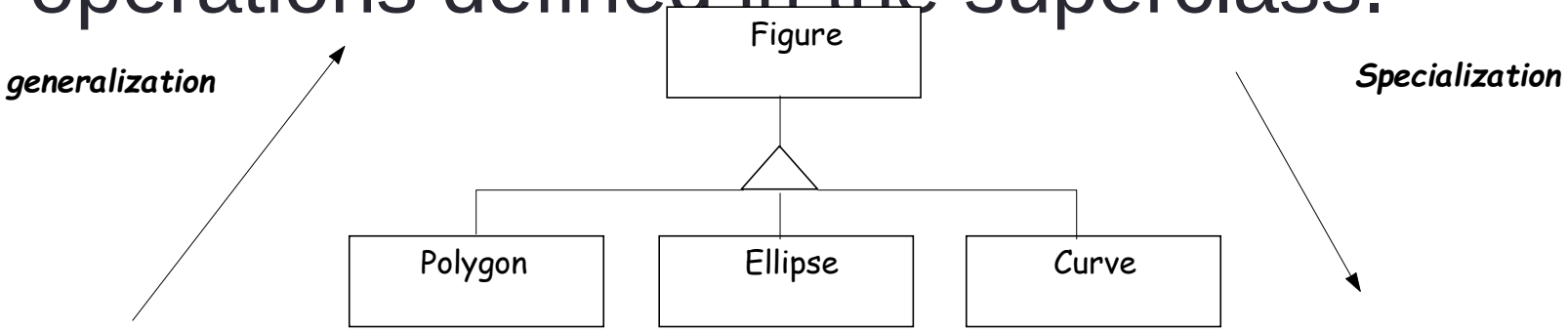


Generalization / Specialization

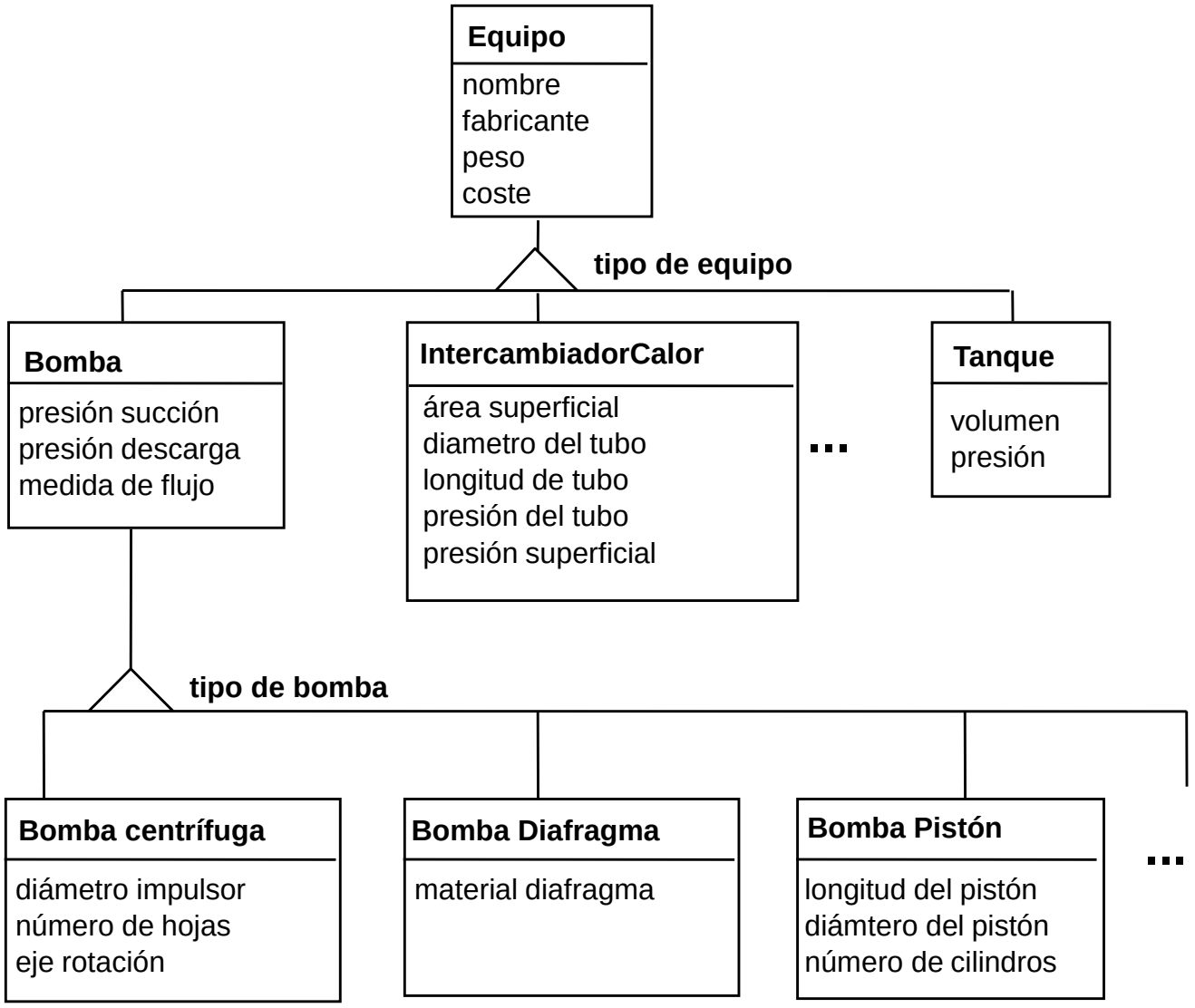
- Hierarchies of classes allow the management of the complexity in terms of taxonomic classifications.
- Starting from classes that have in common a number os attributes and operations, using generalization, a more generic class (super class) can be obtained from these initial classes (subclasses).
- Shared attributes and operations are placed in the superclass

- The specialization is the opposite relationship. Starting from a superclass the subclasses are obtained.

- Subclasses inherit attributes and operations defined in the superclass.



- Each instance in a subclass is also an instance of the parent class. (relationship **is_a**).



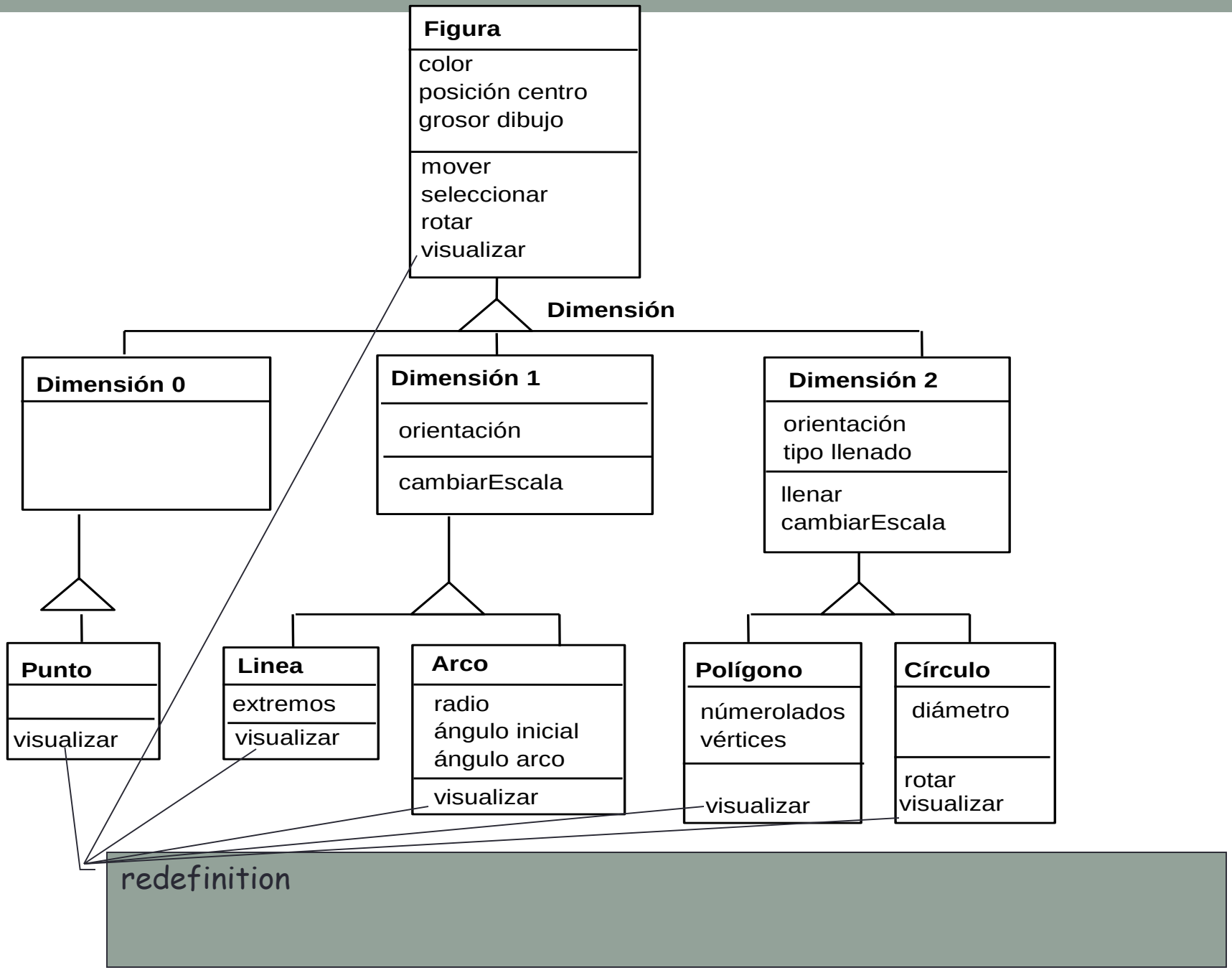
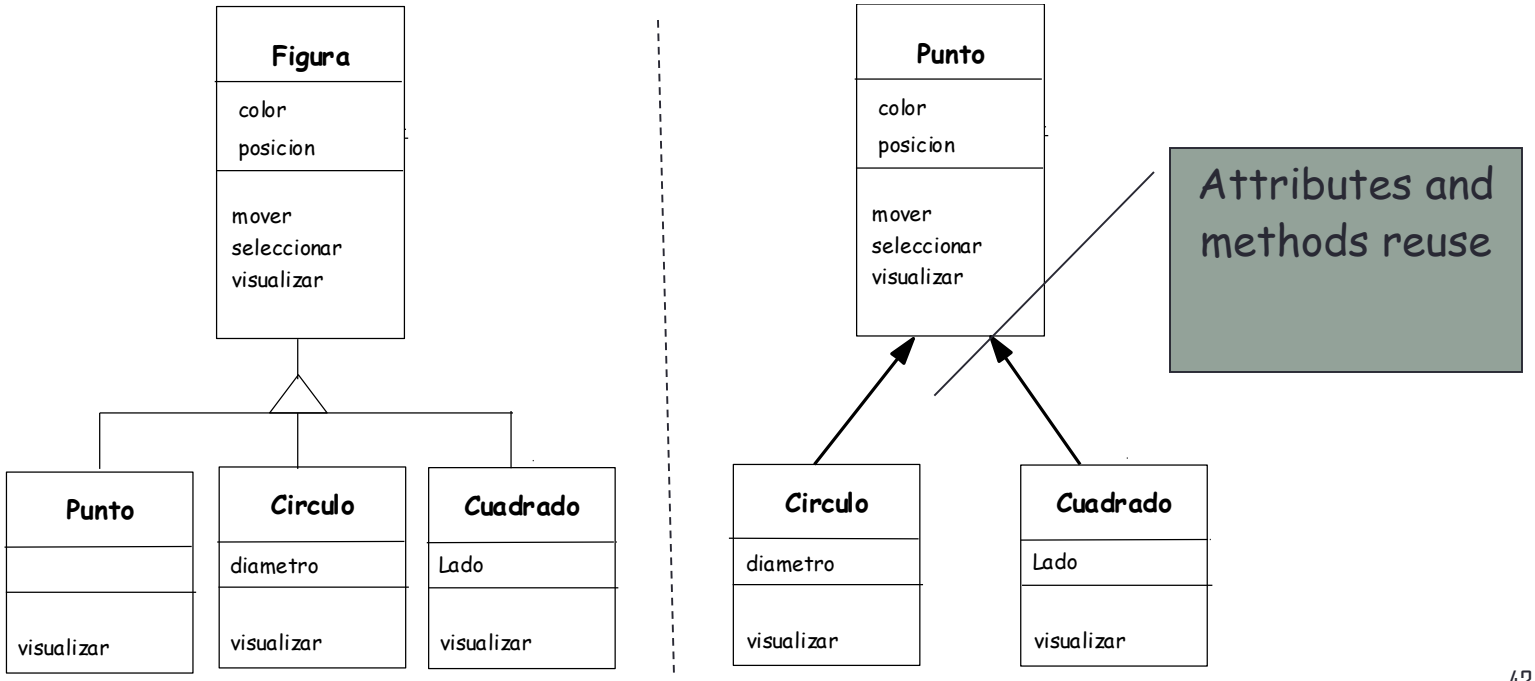
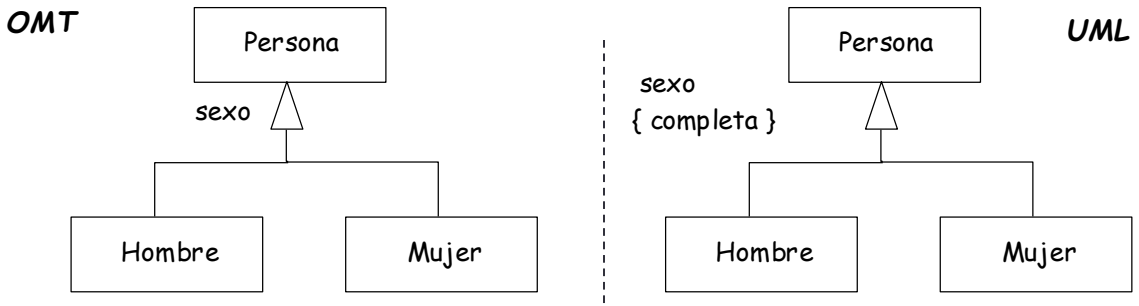


Figura
color
posición centro
grosor dibujo
mover
seleccionar
rotar
visualizar
Dimensión
Dimensión 0
Dimensión 1
orientación
cambiarEscala
Dimensión 2
orientación
tipo llenado
llenar
cambiarEscala
Punto
visualizar
Linea
extremos
visualizar
Arco
radio
ángulo inicial
ángulo arco
visualizar
Polígono
númerolados
vértices
visualizar
Círculo
diámetro
rotar
visualizar
redefinition

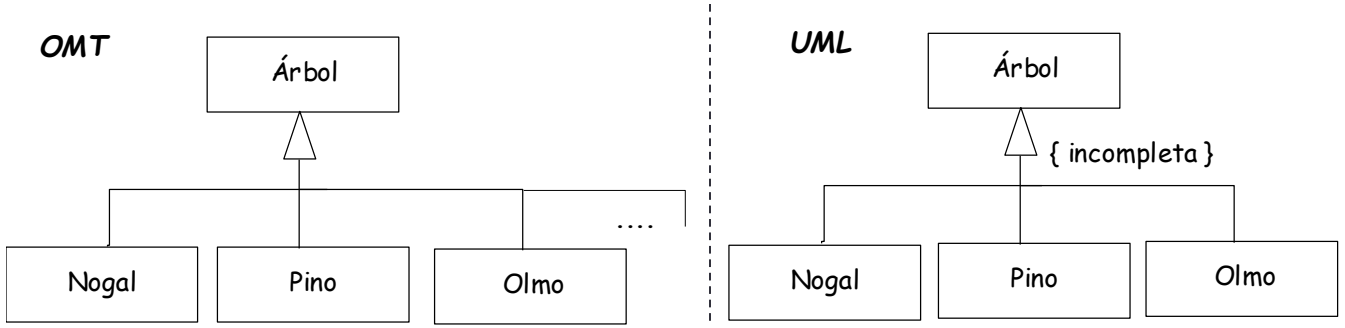
- The specialization relationship is used for conceptual modelling whereas inheritance is a code reusing mechanism during the implementation phase.



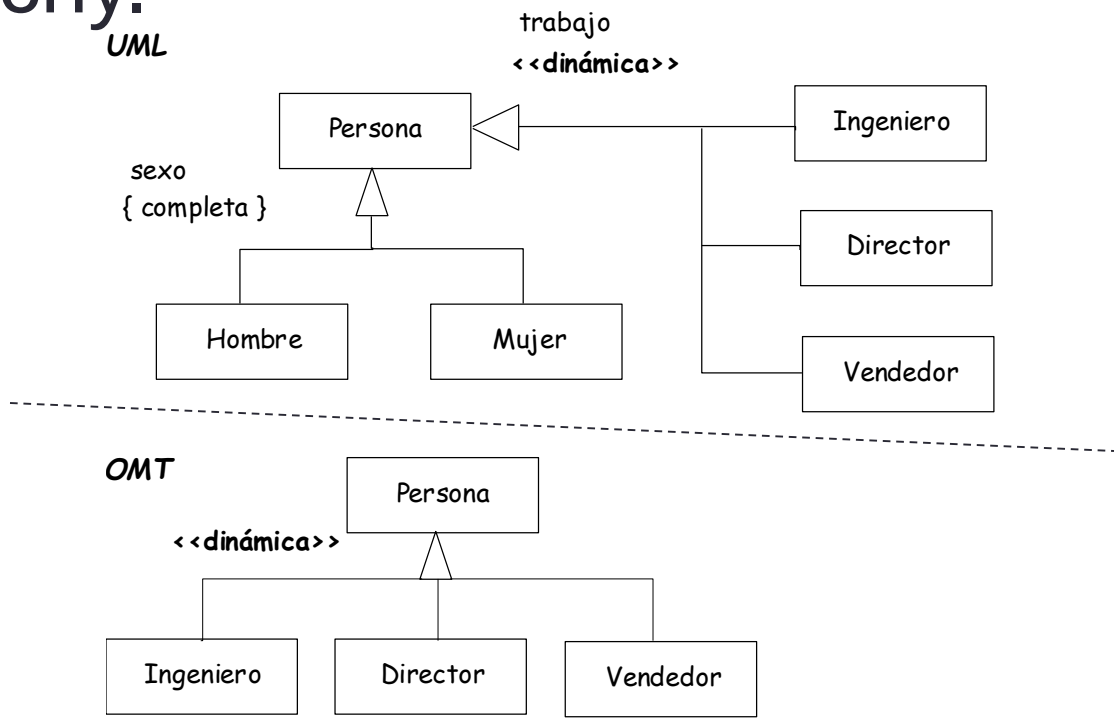
- Two types of **restrictions**:
  - **Complete**: All children classes are specified in the model



  - **Incomplete**: Not all children specified



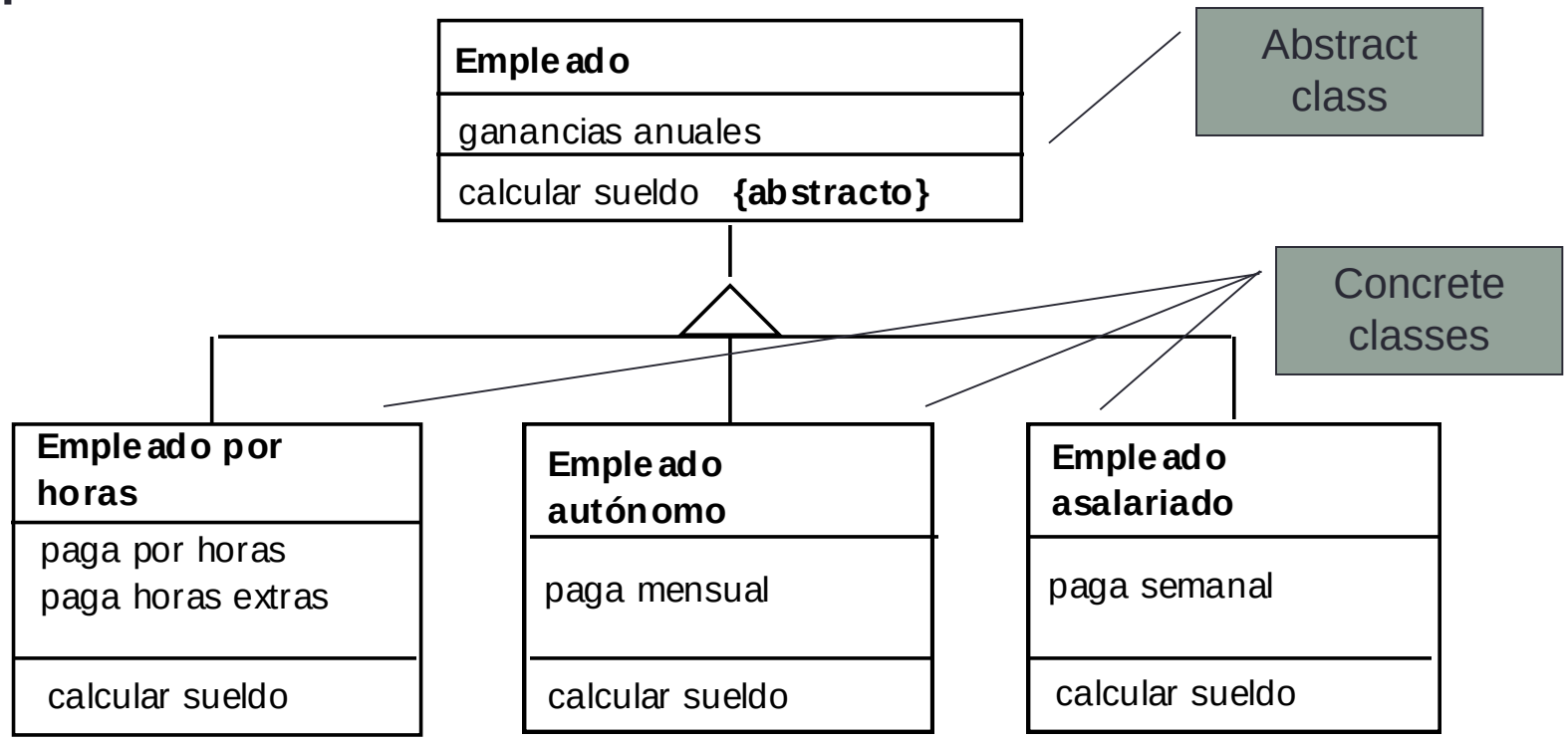
- A specialization is dynamic if an object may change to a different class within the hierarchy.



Abstract Classes

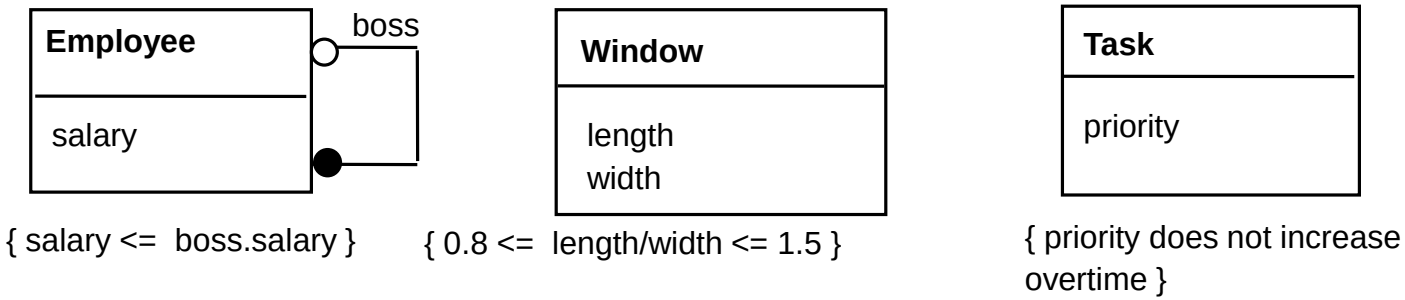
- An abstract class has no instances. Its descendant classes have them.
- An abstract class has at least one method without code (undefined methods)
- Abstract classes are used to defined the operations that are inherited by the descendant classes. They provide the protocol (interface) but without giving a concrete implementation.

- All concrete subclasses must provide an implementation for an abstract method.

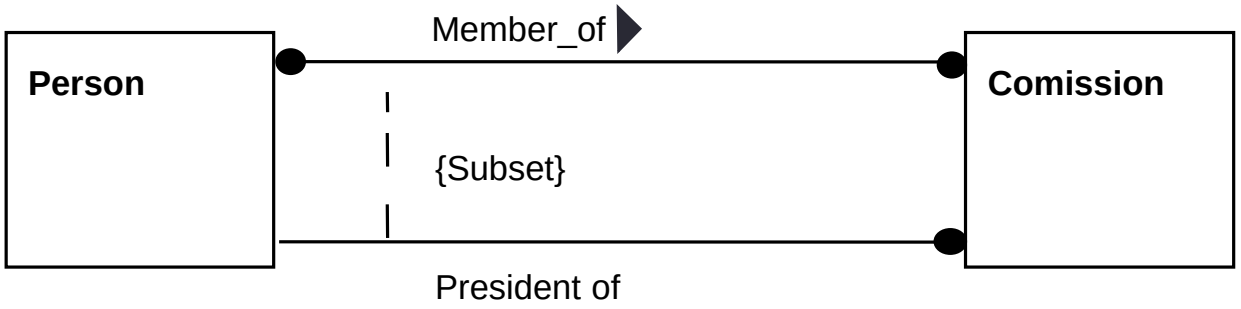


Restrictions

- Restrictions are functional relationships between entities in the model.
- Usually expressed in a declarative form but also natural language can be used
- These restrictions may refer to values of the attributes of an object.



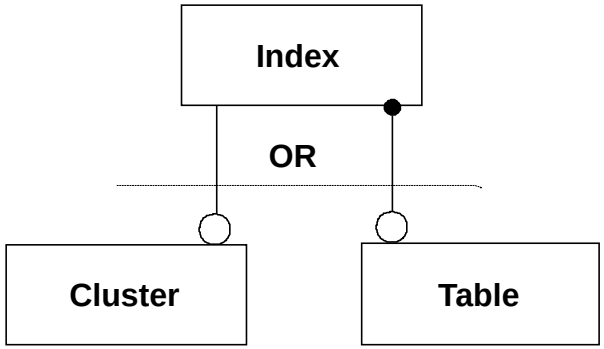
- They are also used between association relationships.

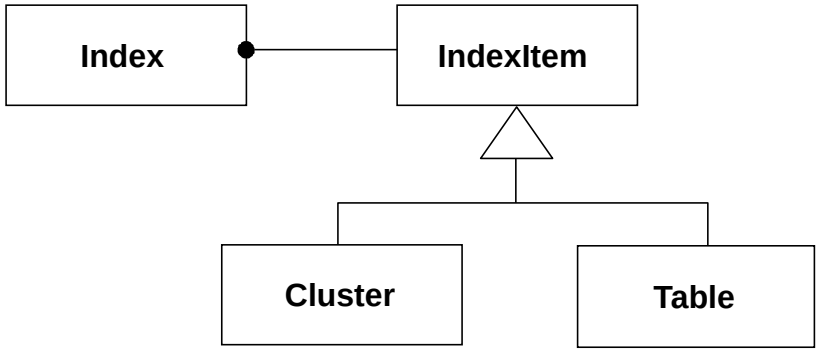


Exclusive associations

- An axclusive association (or-association) consists of set of associations that relate an initial class (source) with several destination classes.

- Taking an object of the source class, it is at most related with one object of a destination class.





CHAPTER 4: OO MODELING WITH UML

Software Engineering

Contents

Motivation and Origins

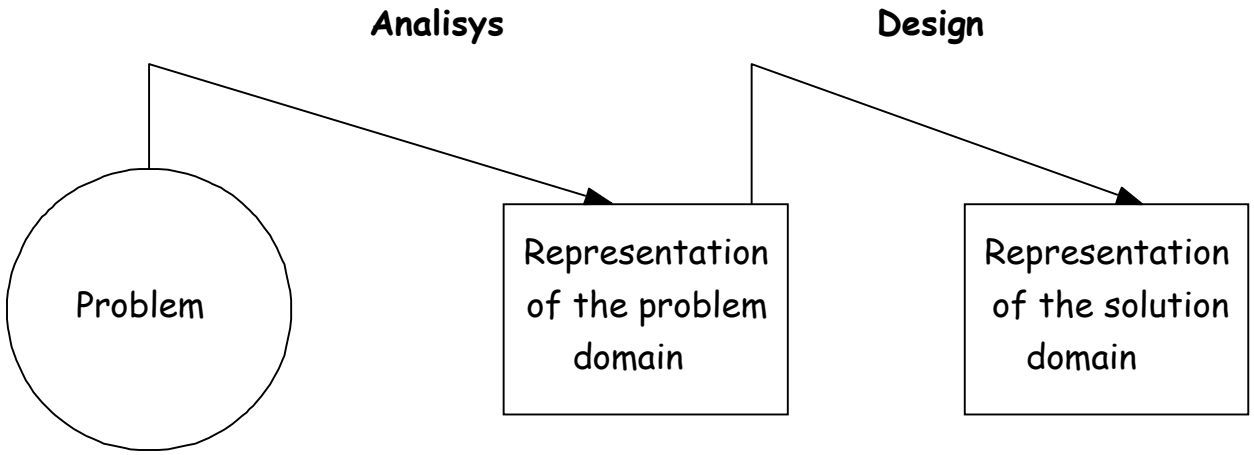
- OO Programming languages appear.
- The use of these languages requires a new viewpoint with respect to analysis and design.
- First OO analysis and design methods appear.

Motivation

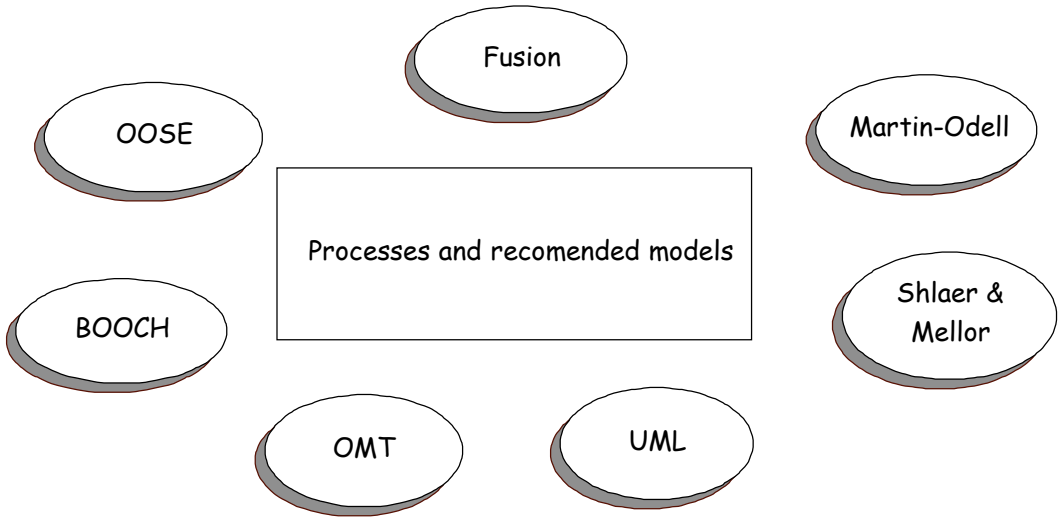
- OO methods represent requirements in terms of objects and the services they offer.
- OO methods are more “natural” than traditional ones:
  - Functions/processes vs objects.

Motivation

- In OO methods the decomposition of the system is based on objects or classes that are discovered in the problem domain

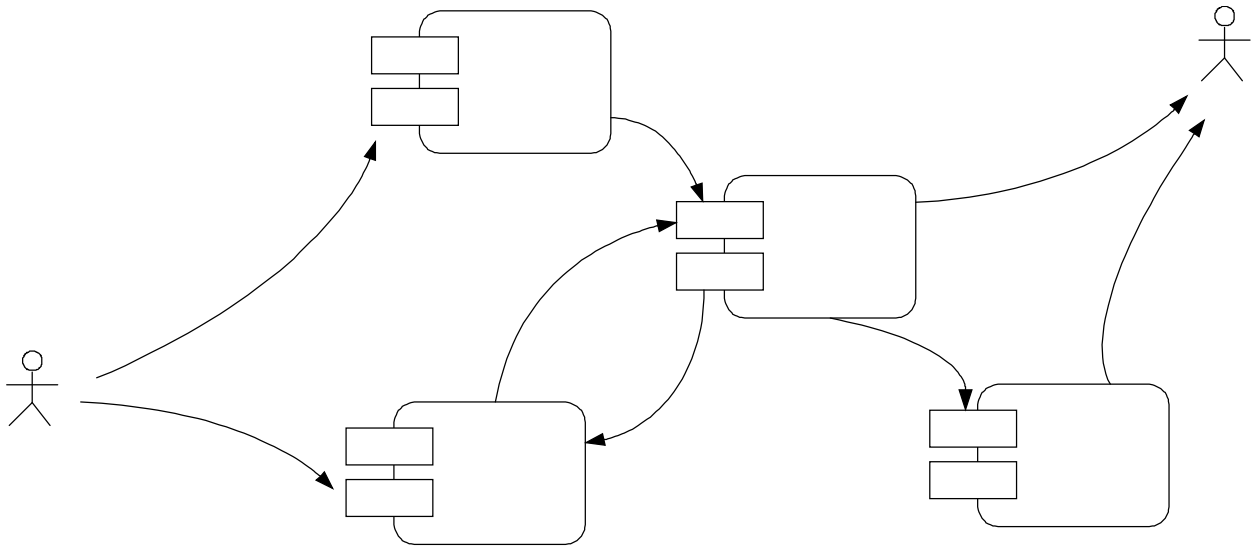


Origins



- Rumbaugh, Blaha, (OMT)- 1991
- Coad, Yourdon – 1991
- Shlaer, Mellor- 1992
- Booch- 1992
- Odell, Martin –1992
- Jacobson (OOSE) –1992
- Fusion – 1994
- Booch, Rumbaugh, Jacobson (UML) -1997

2 View of a Software System

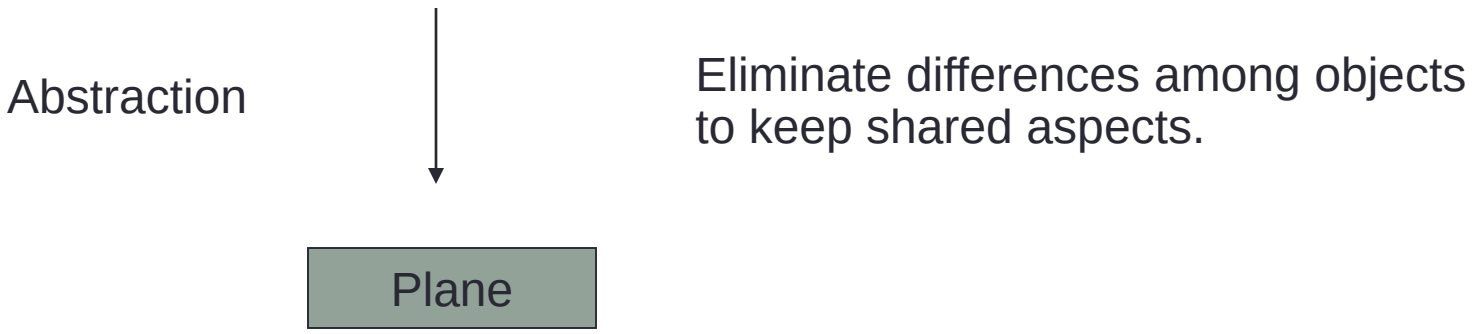


Static View

- Object:
  - Entity that exists in the real world.
  - Have identity and are differentiated.
  - Examples:
    - The bill 2003/0010
    - The plane with plate number 123
    - A customer
    - The plane with plate number 345

Static view

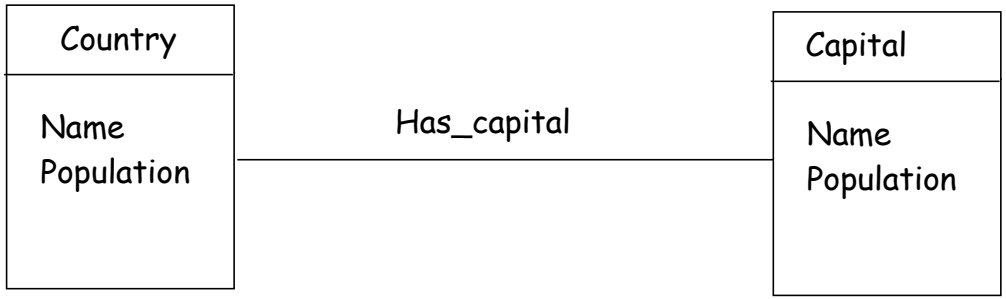
- Object Classes: Describe a collection of objects with:
  - Same properties.
  - Shared Behavior.
    - The plane with plate number 123
    - The plane with plate number 345



Static View: Associations

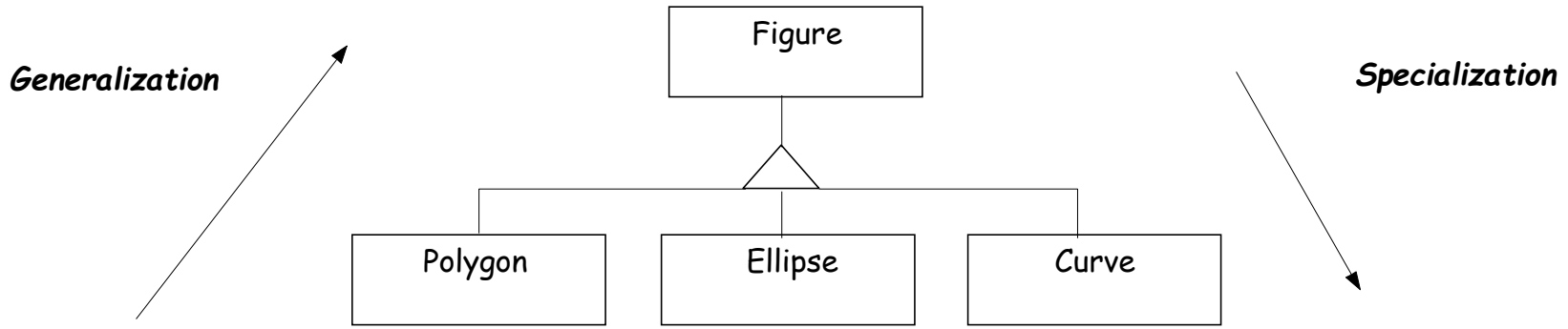
- Association: Allows linking or connecting objects of different classes.

- Example: A country has only one capital.



Static View: Generalization/ Specialization

- **Generalization:** Act or result obtained after distinguishing a concept that is more general than another.



- Inheritance: Allows properties and operations of a class to be accessible by a subclass.

Static View

- Static Aspect: Describes the static structure of a system and its interrelationships.

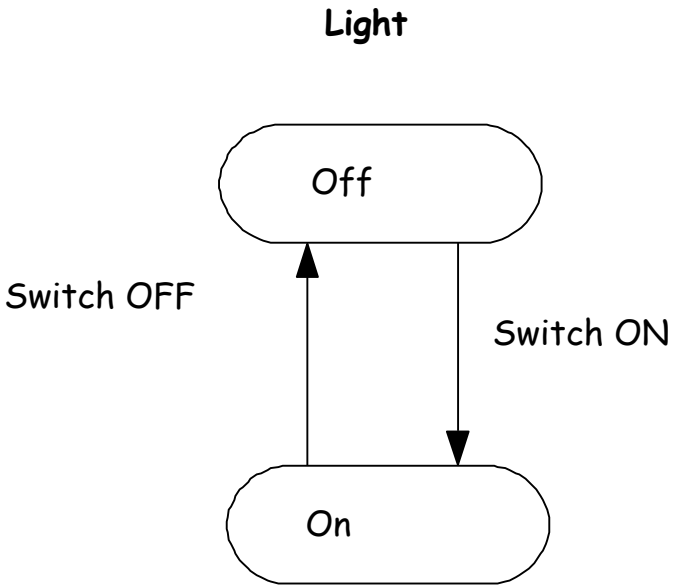
	Intra-objects	Inter-objects
Static Aspect	Object classes. Attributes Operations	Association Generalization ....

Dynamic View

- Objects communicate by means of invocation of operations on other objects.
- The dynamic view describes the aspects of a system that change over time:
  - Interactions between objects.
  - Possible states of an object.
  - Transitions between states.
  - What events are produced.
  - What operations are executed.

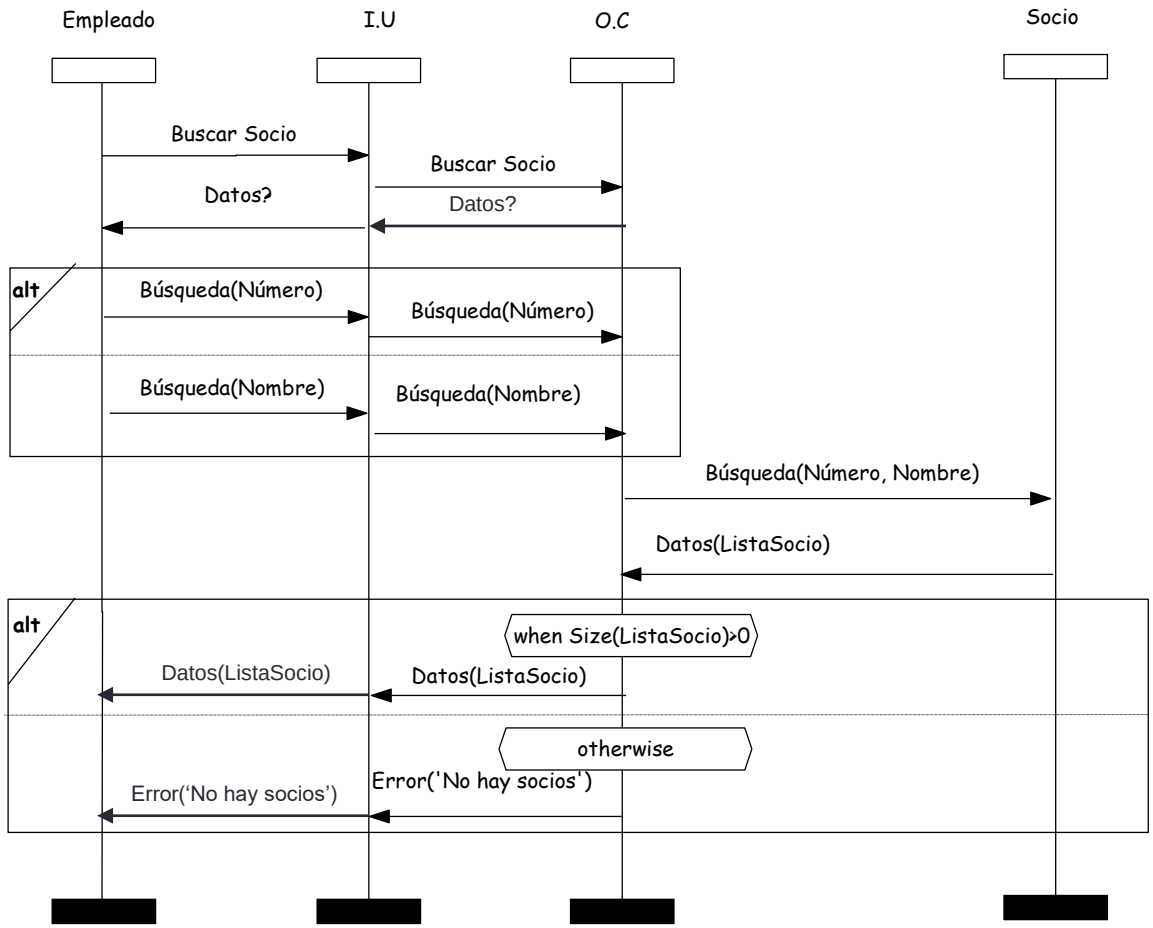
Dynamic View

- State transition diagram.



Dynamic View

- MSCs: describe interactions between different objects.



Static/Dynamic Views

- Static View: Structure and interrelationships.
- Dynamic View: Aspects that change overtime.

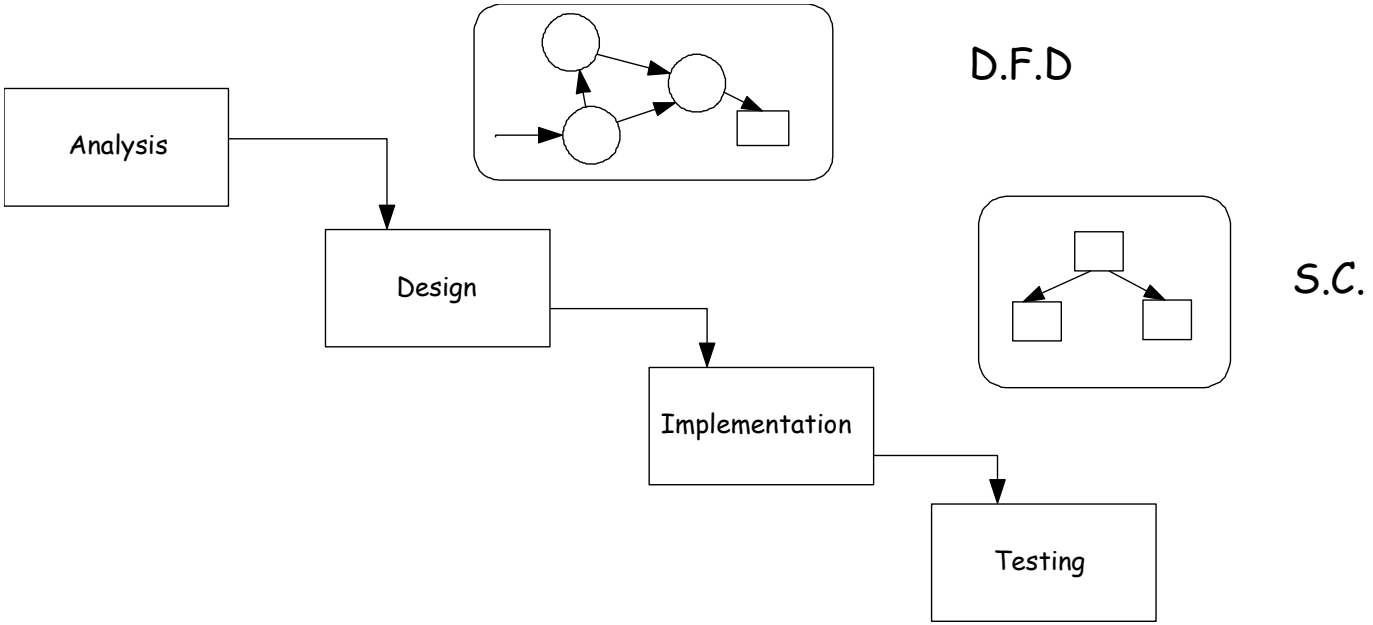
	Intra-object	Inter-objects
Static Aspect	Object classes. Attributes Operations	Association Generalization ....
Dynamic Aspect	State Transition Diagrams	MSCs ....

3 OO Methods

- OO Analysis:
  - A **specification** of the problem is created.
  - Describes **what** to do with the system.
- OO Design:
  - Definition of a software **solution** to satisfy the requirements.
  - Describes **how** the system will work

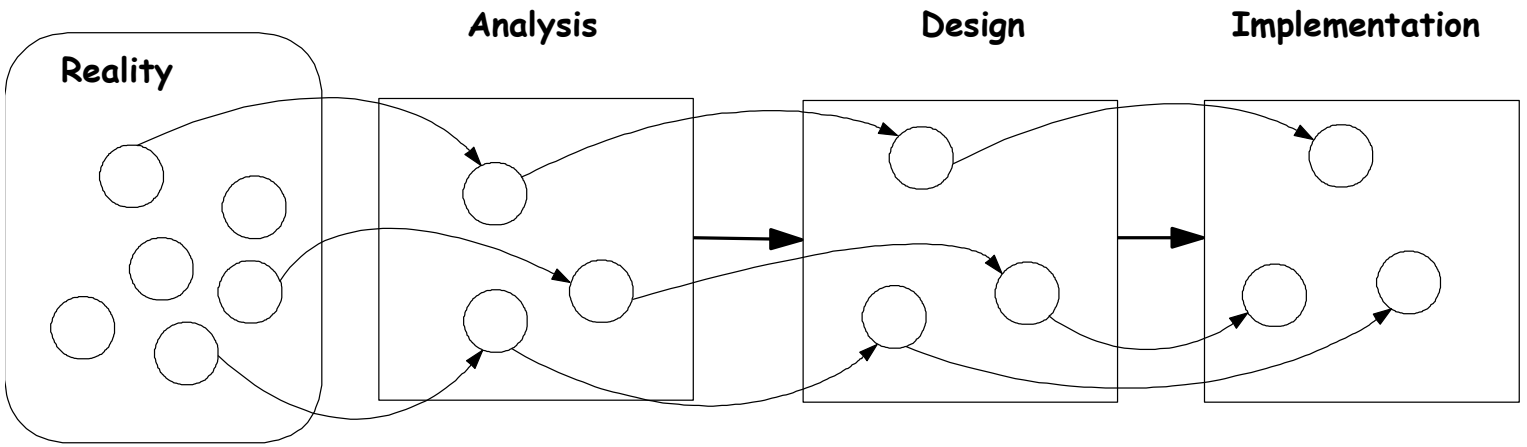
OO Methods: Continuity between models

- Structured techniques:



OO Methods: Continuity between models

- In OO:



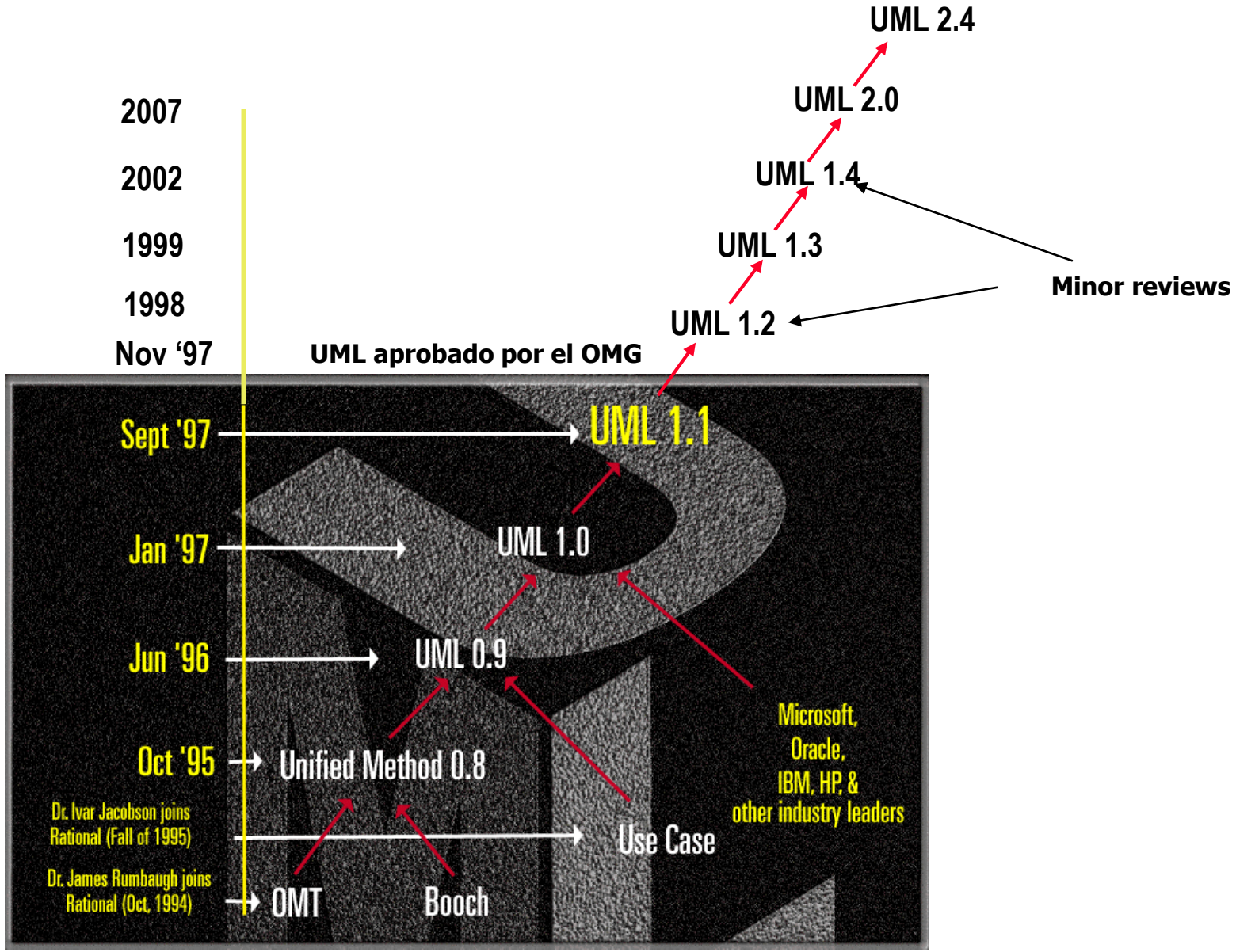
4 The UML Language

- UML = Unified Modeling Language
- UML: A general purpose language for OO modelling
- Starting Point:
  - Many OO methods with different notations.
  - Learning and tool construction inconvenients.
  - A Uniform notation needed.

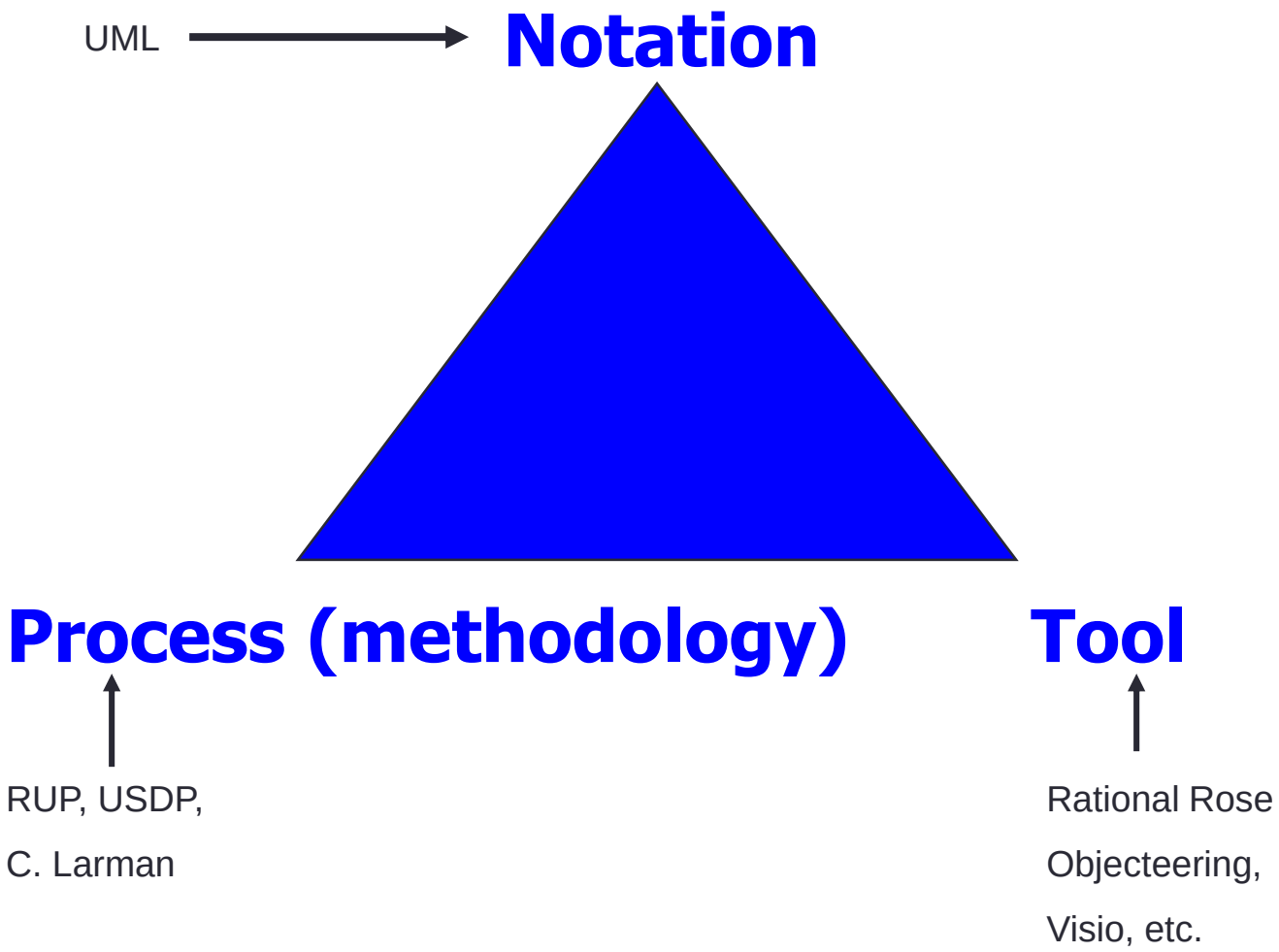
UML History

- Started as the “unified method”with the participation of J. Rumbaugh and G. Booch in 1995.The same year I. Jacobson is incorporated.
- Partners in Rational Software, CASE tool Rational Rose.

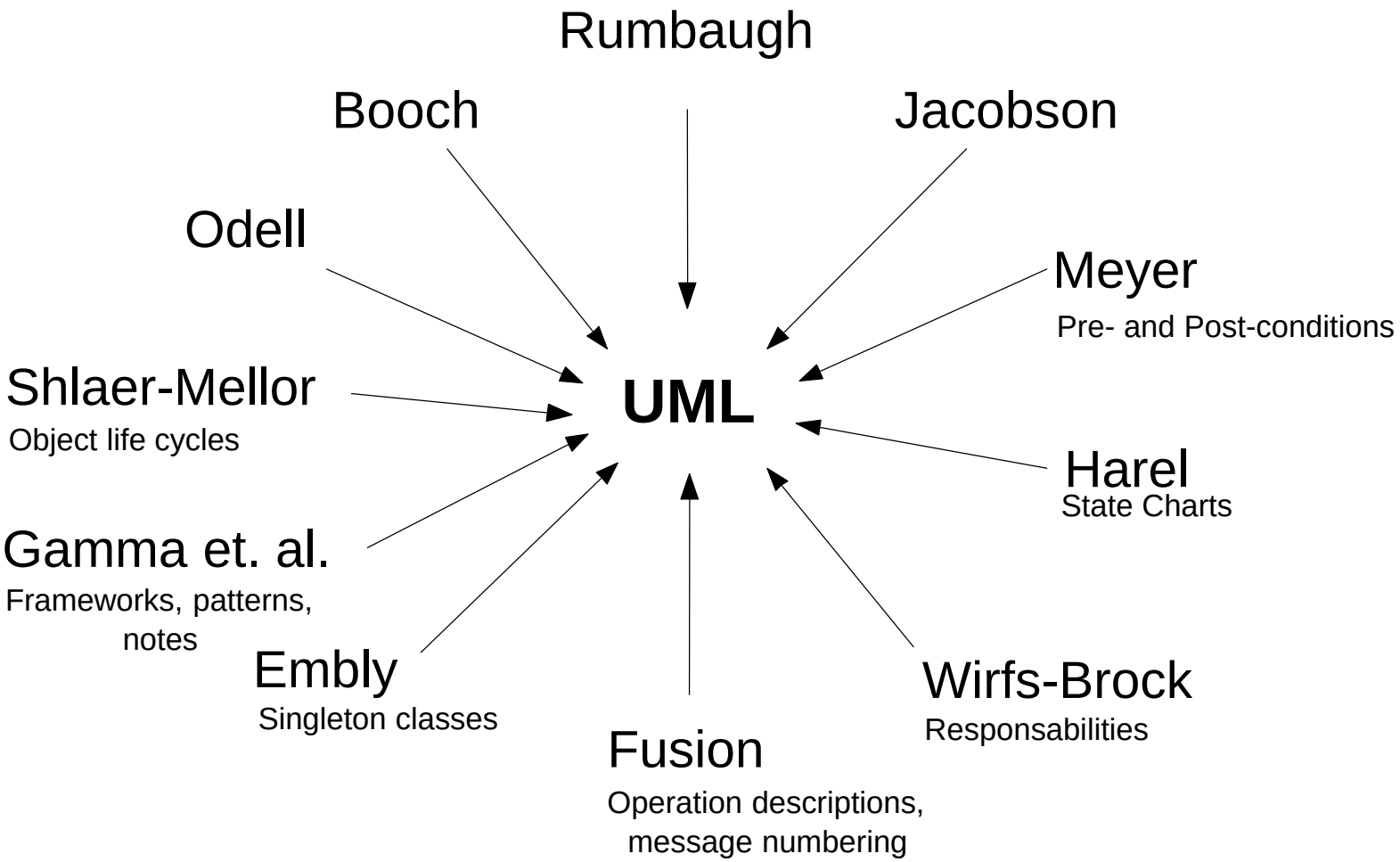
UML evolution



UML: the success triangle



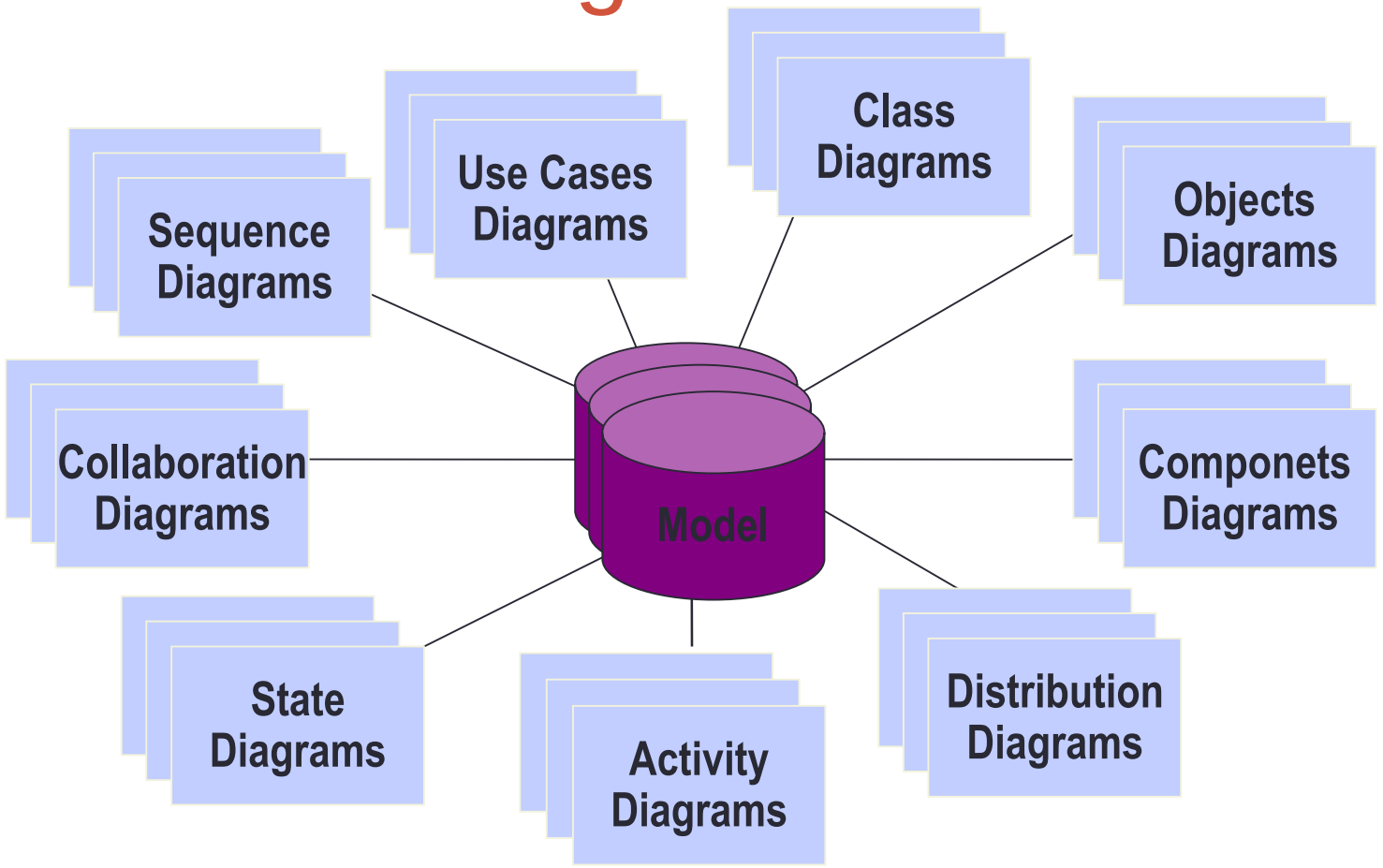
UML merges OO approaches



UML

- UML is not a method, it is a notation to describe systems.
  - Processes based on UML:
    - USD “Unified Software Development Process” by I. Jacobson.
    - RUP “Rational Unified Process” by Rational Software.
    - C. Larman “UML and patterns”.

UML Modelling



“A model is a complete description of a system from a concrete viewpoint”

UML Charts

- Use cases Chart
- Class Chart (including instances chart)
- Behavior Charts
  - States Chart
  - Activity Chart
  - Interaction Diagrams
    - Sequence Diagram
    - Collaboration Diagram
- Implementation Diagrams
  - Components Diagram
  - Deployment Diagram

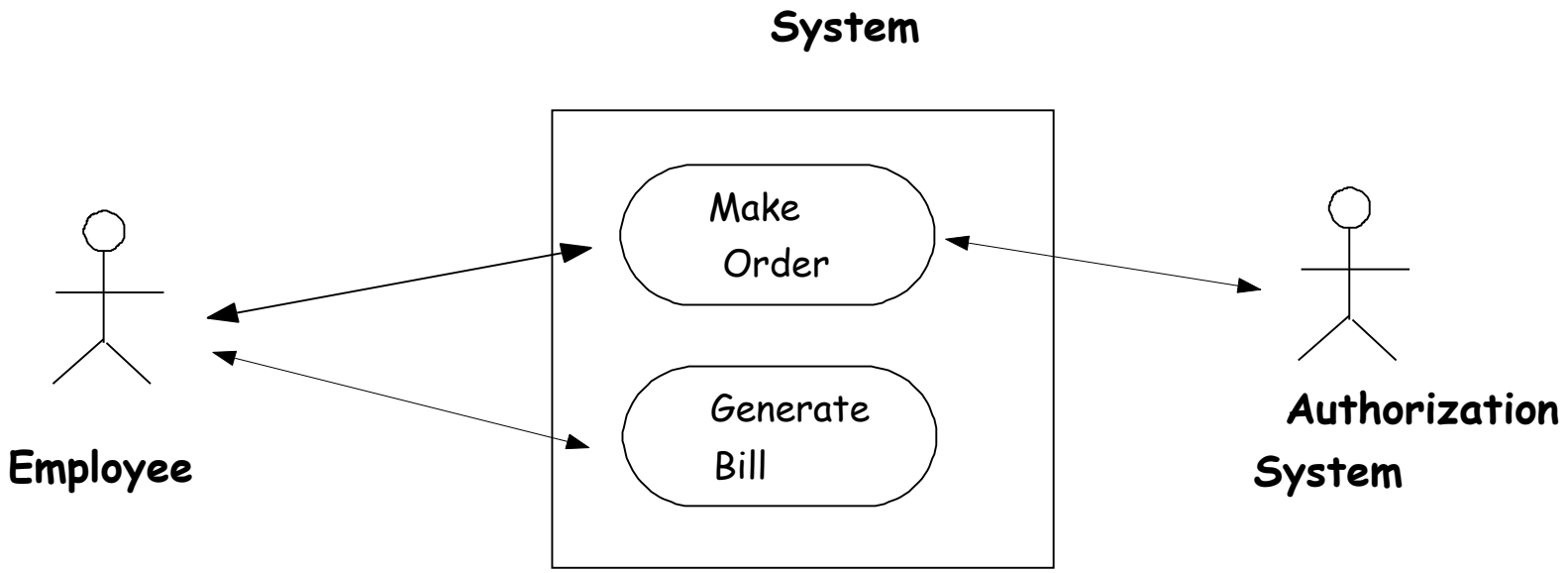
Use Cases model

- Use cases is a technique to capture information about how a system or business presently works and how it is required to work in the future.
- They are used during requirements gathering to capture functional requirements of a system to be developed.

Use Cases

- Actors: Entities that exchange information with the system.
- Types of Actors:
  - Humans.
  - Devices.
  - Other software systems.
- A use case contains a sequence of transactions that exchange the actors and the system whenever a given functionality must be executed.

Use Cases: notation



Make Order Use Case Description

User Intentions	**System Obligations**
1. Employee selects New Order	
	2. System requests customer code.
3.Empoloyee inserts customer code	
	4. System checks it exists
5. ***While not end lines selected***	
6. Employee inserts item code and quantity	
	7. System checks code exists
	8. System calculates total of line and echoes description and total of line
End While	
	9. System calculates Orde r total and echoes total.
	10. System requests customer payment card number
11. Employee inserts card number	
	12. The system verifies validity against authorization system
13. Employee selects process order	
	14. System generates order number, echoes it and stores all the information.

Synchronous extensions

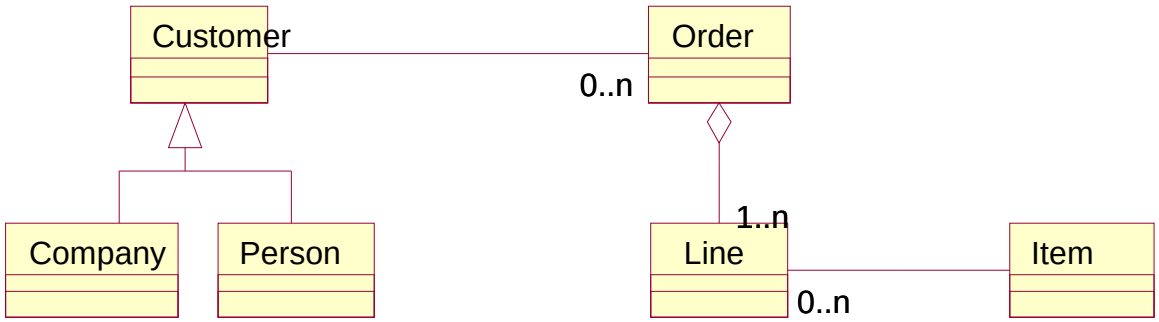
#1	15. At 4 the customer does not exist, the system reports error and go to setp 2.
#2	16. At 7, the item does not exist, the sytem reports error and go to step 5.
#3	17. At 12 The card number is not valid. Go to step 11.

Asynchronous Extensions

#4	
18. In every step the employee may select Abort	
	19. The use case ends without any information storage.

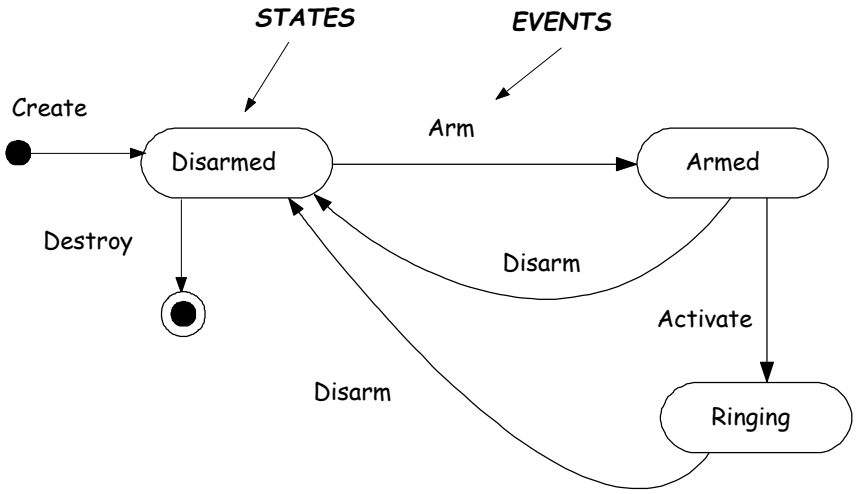
Static Models

- Show the classes of a system and the relationships between them.



Dynamic Models

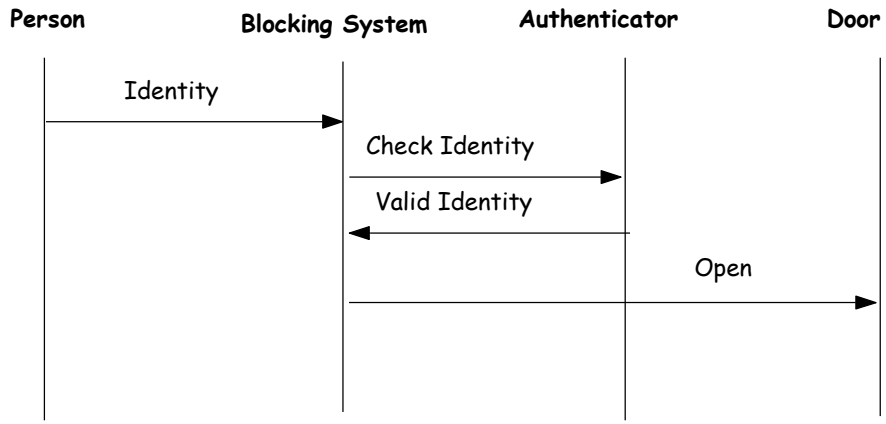
- State Transition Chart: It shows the lifecycles of the objects in the system.

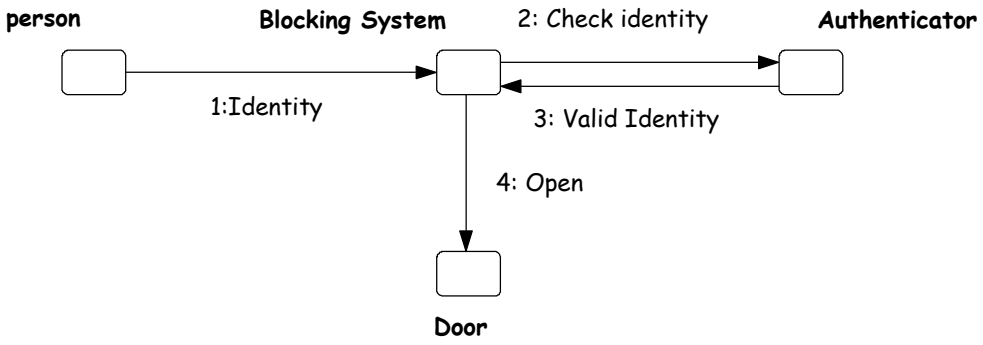


Dynamic Models

- Sequence and Collaboration Diagrams: Show messages that are exchanged by objects that participate in a scenario or use case.

Sequence & Collaboration





Activity Diagrams

- Special case of a States diagram where:

  All (or most) states are action-ones

  All (most) transitions are triggered by the finalization of actions.

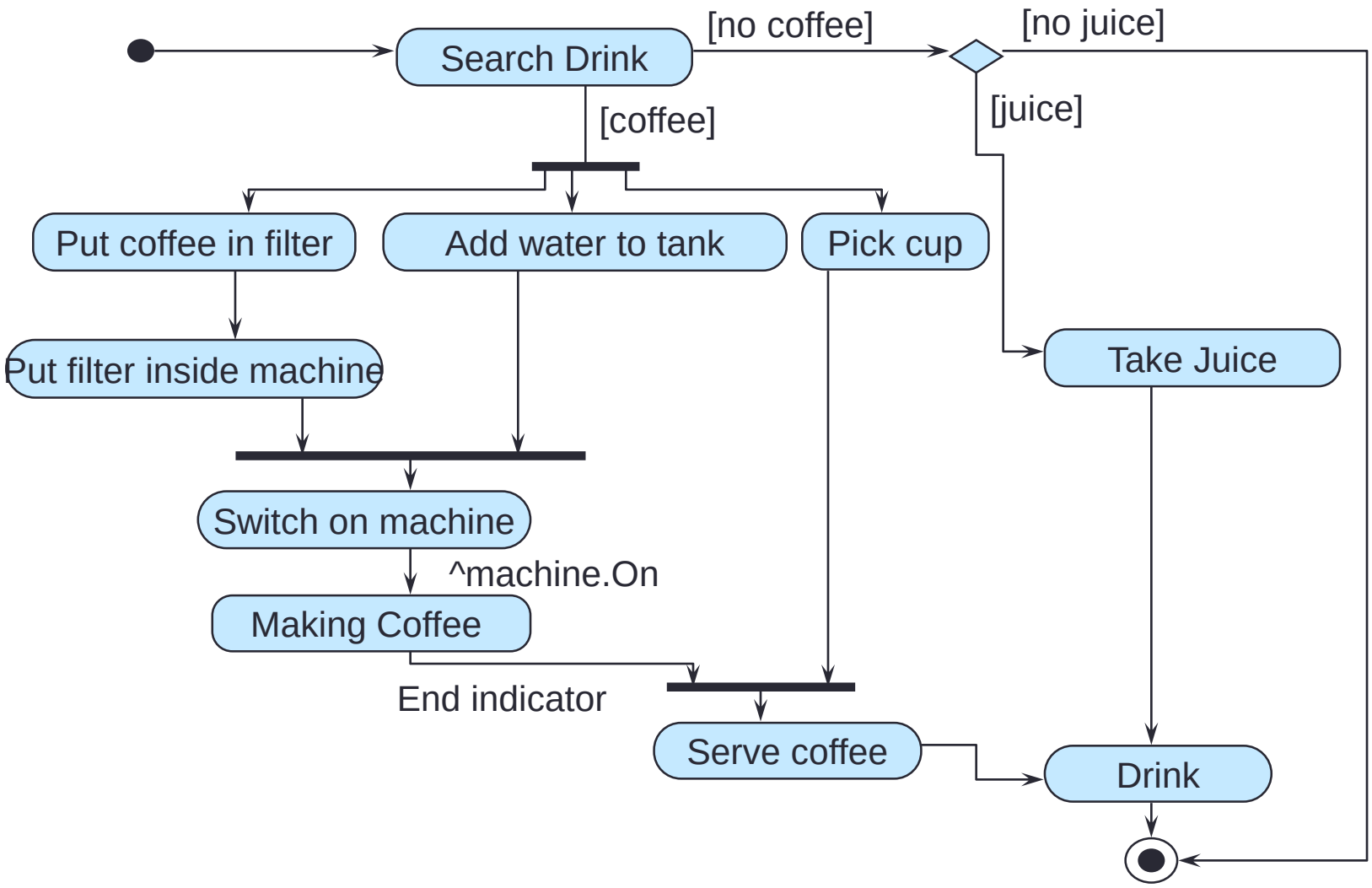
- The diagram may be associated to a :

  Class

  Implementation of an operation

  A use case

Example


Search Drink
[no coffee]
[no juice]
[coffee]
[juice]
Put coffee in filter
Add water to tank
Pick cup
Put filter inside machine
Take Juice
Switch on machine
^machine.On
Making Coffee
End indicator
Serve coffee
Drink

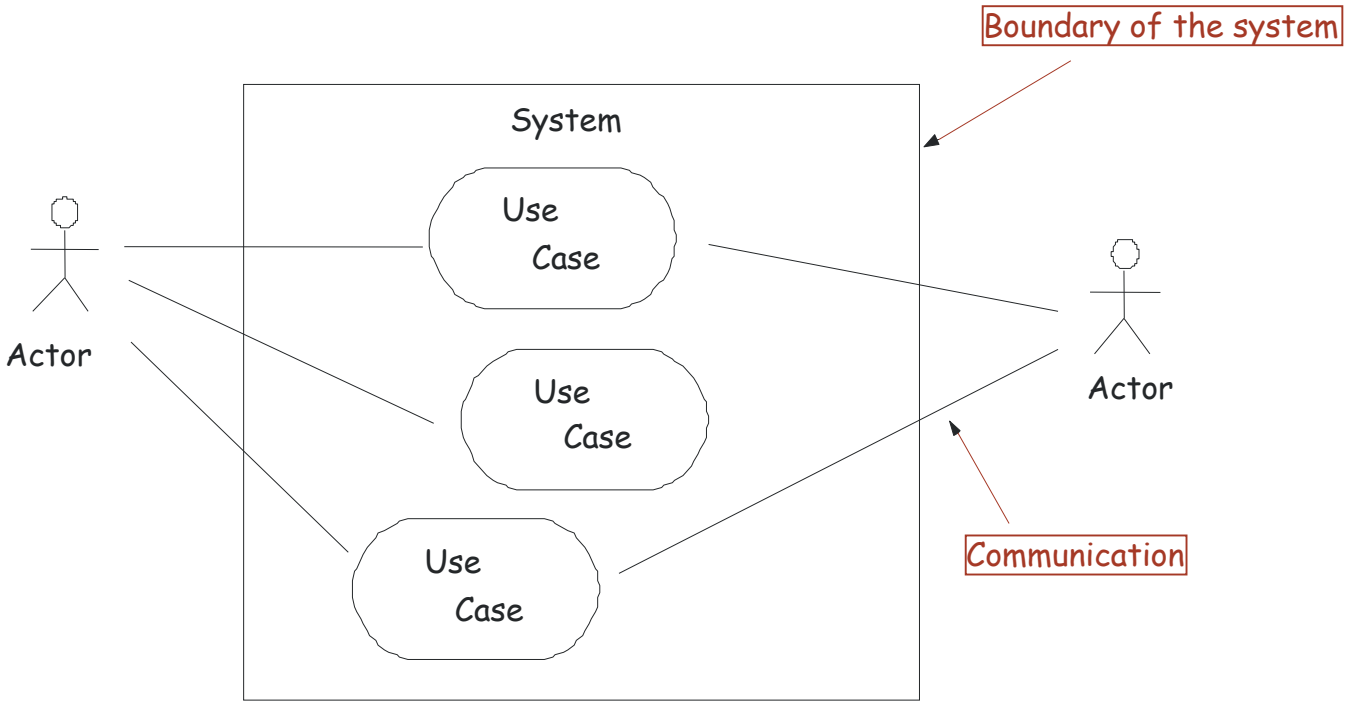
Chapter: Use cases Models

- Actors & Use Cases
- Relationships
- Use cases Diagrams
  - Context Diagram &
  - Initial Diagram
  - Specification Templates
  - Construction Process

Use Cases

- A technique to capture how an existing system works or how a future system should work
- Use to capture functional requirements.

Notation

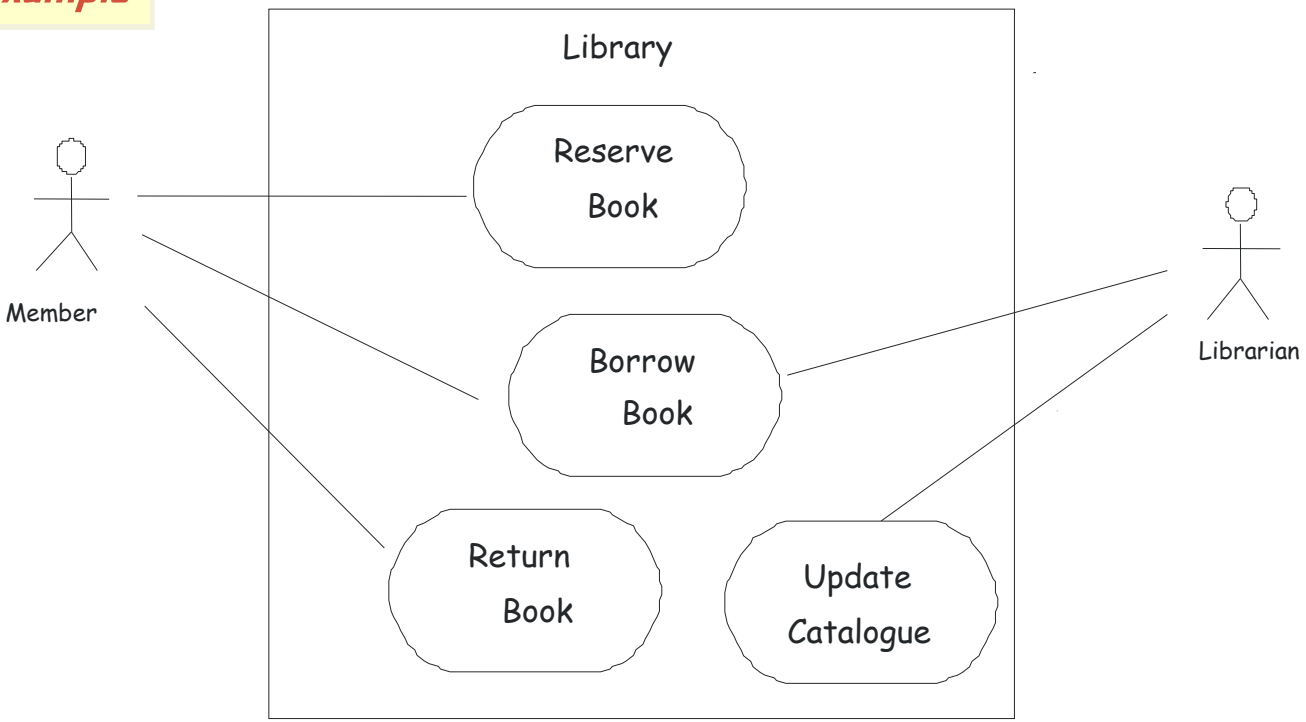


Actors and Use cases

Actor: Entity (Human, Device or another software system) exchanging information with the system

Use case: Consists of the sequence of transactions/messages that actors and the system exchange when a given functionality of the system is executed

Example



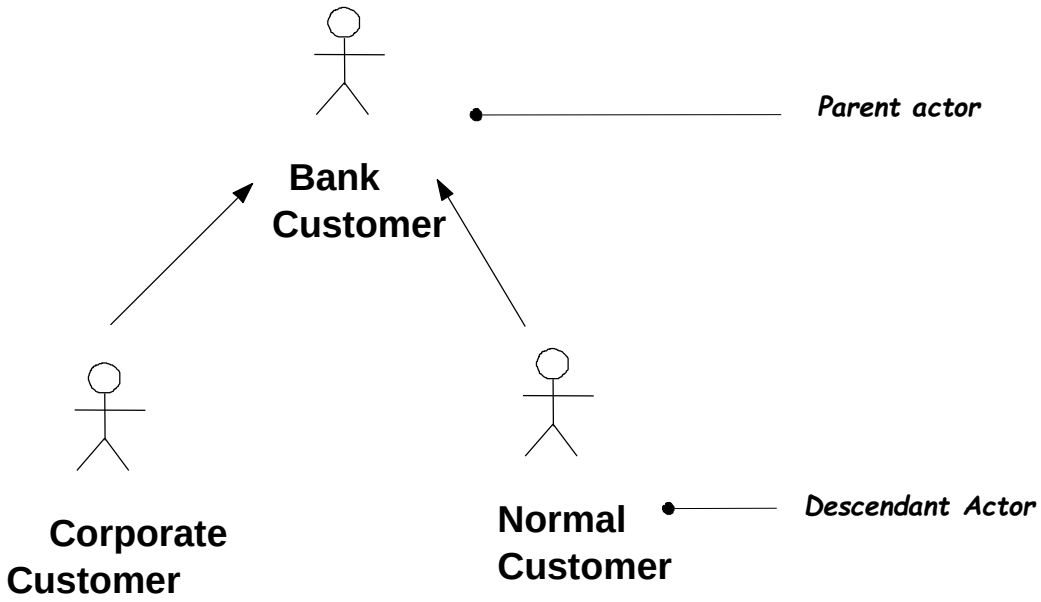
Description Templates

- Use cases are described using templates and natural language

Use Case	
Actors	
Summary	
Preconditions	
Postconditions	
Includes	
Extends	
Inherits from	
Flow of events	
Actor	*System*

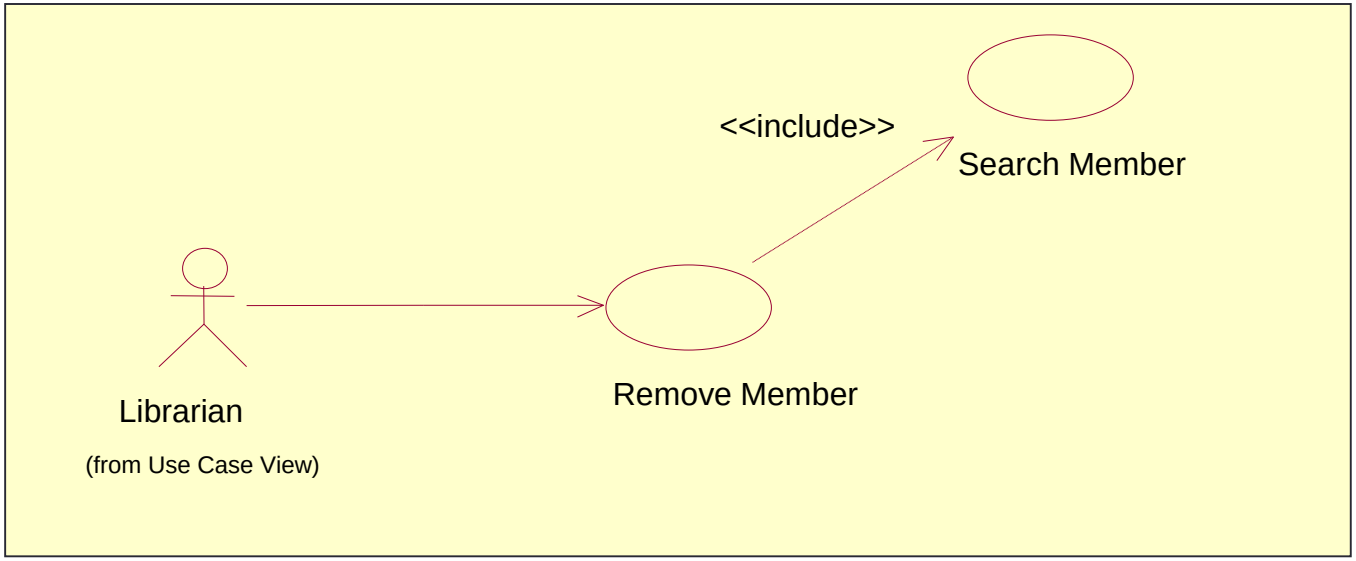
Relationships between actors- Inheritance

- The inheritance relationship indicates that the descendant actor may play all the roles of its predecessor actor.



Relationships between use cases- Inclusion

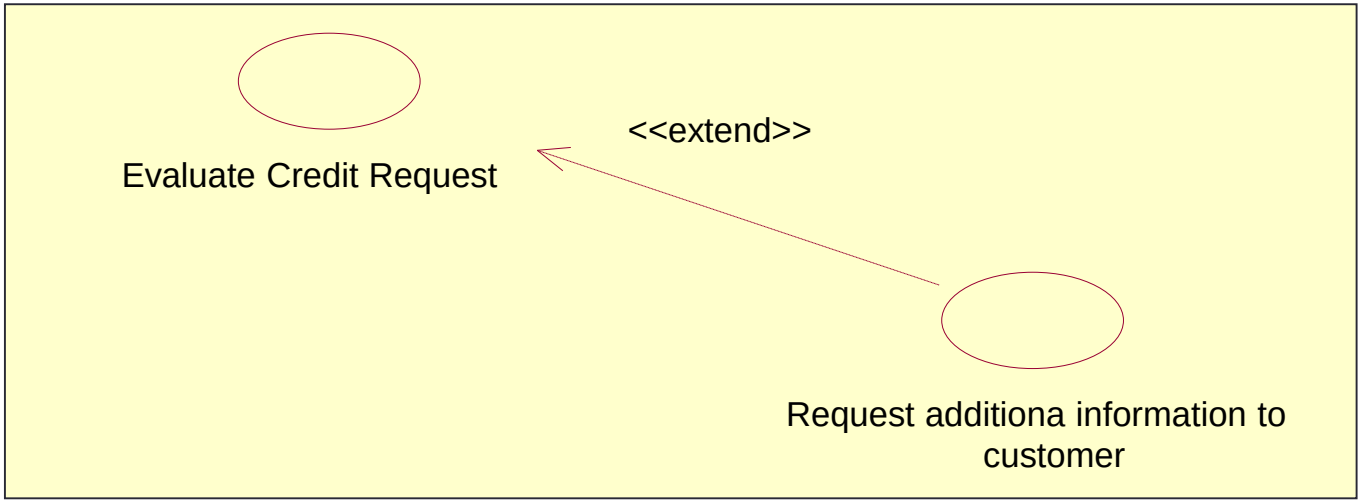
- A use case A includes a use case B, if an instance of A executes all the events that are described in B.



The instantiation of Remove Member always uses the flow of events defined in Search Member

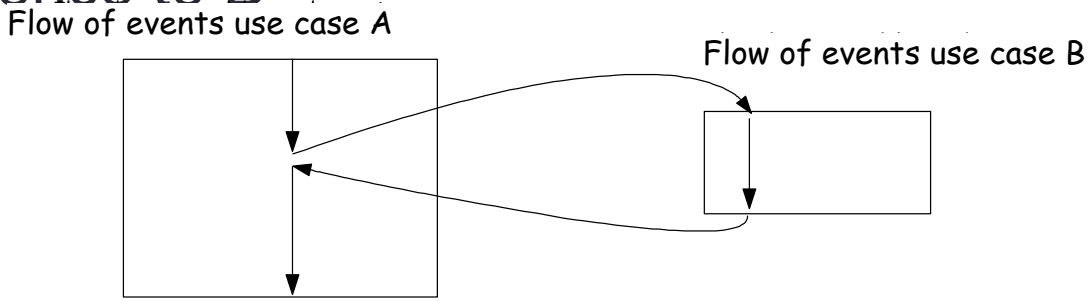
Relationships between use cases- Extension

- A use case B extends a use case A, if there is a condition in the description of A that if evaluated to true all the events described in B are executed.

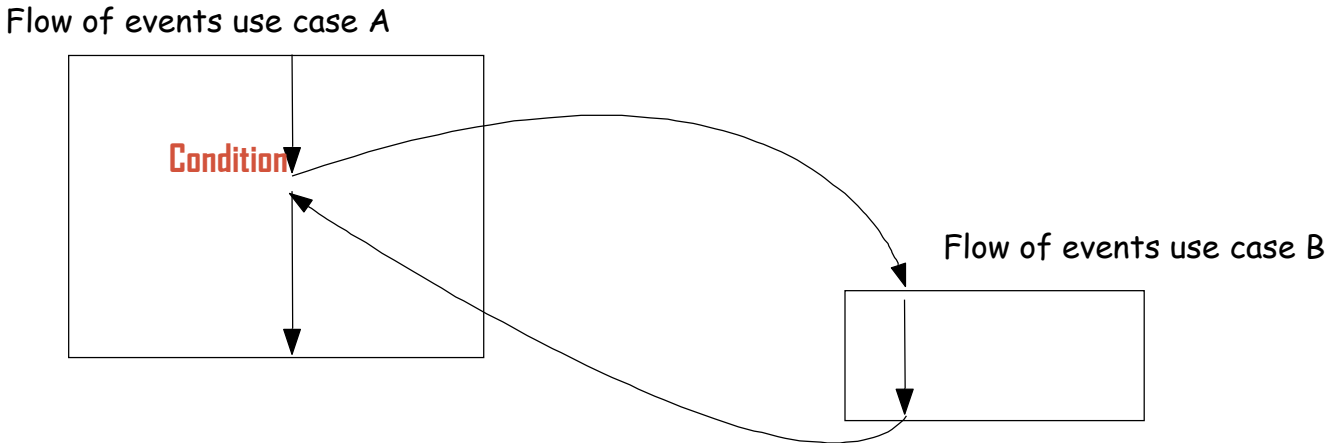


Relationships between use cases

- Inclusion: The description of use case A includes a reference to B

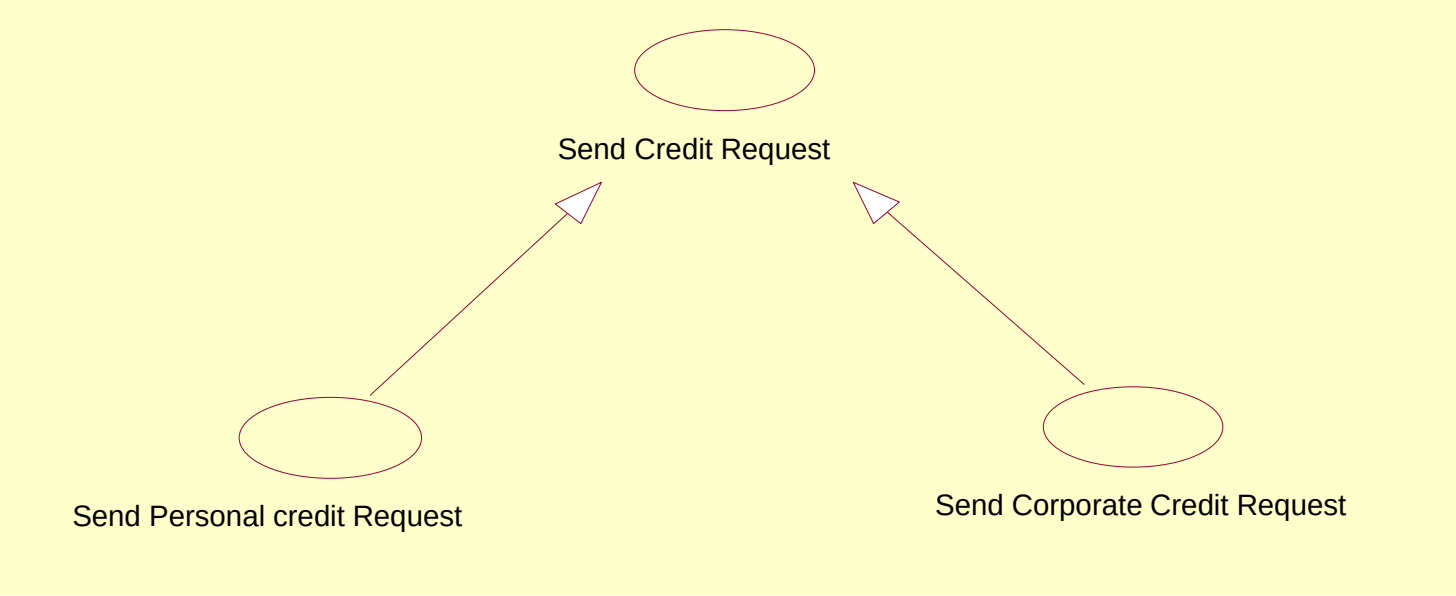


- Extension: Equivalent to an inclusion + a condition



Relationships between use cases- Inheritance

- A use case B is an specialization of another use case A, if the flow of events in B is a refinement of the flow of events in A
  - Similar to inheritance in OO (it allows the separation between a generic interaction pattern (parent use case) and a more specific case (descendant case).

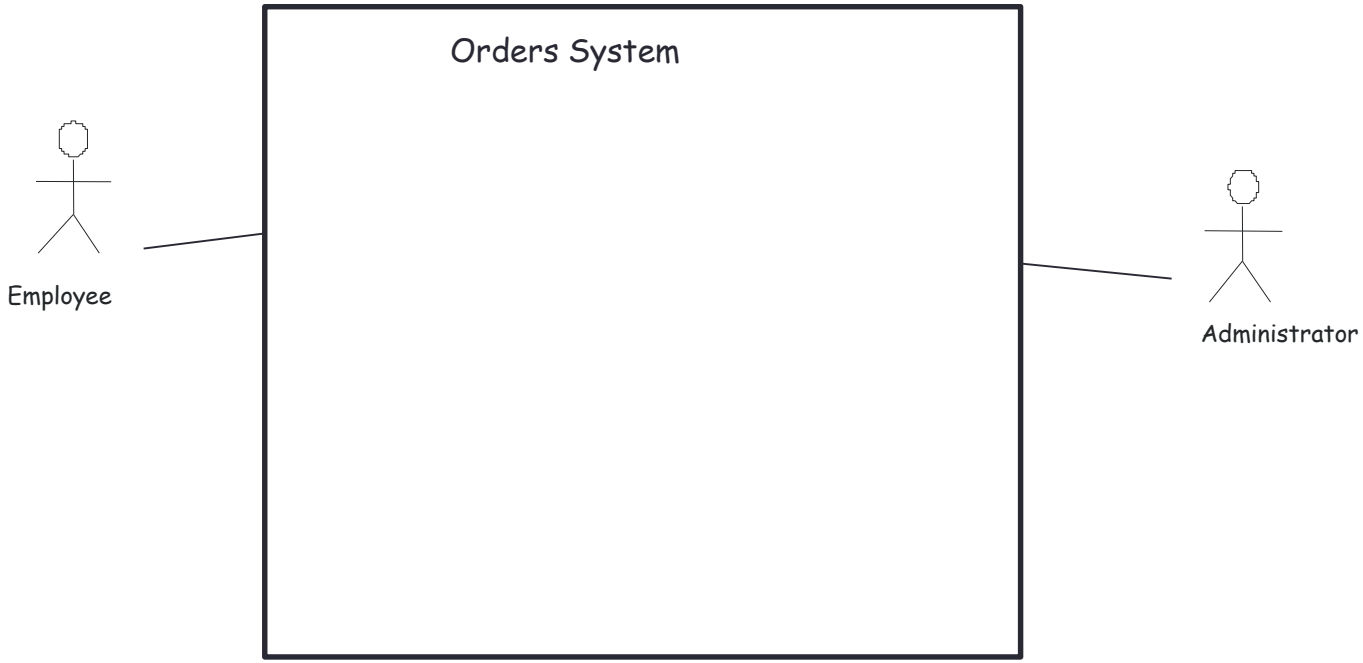


Use cases Diagrams

- Structured in three layers
  - Context diagram and Initial context
  - Description templates
  - Structured Diagram

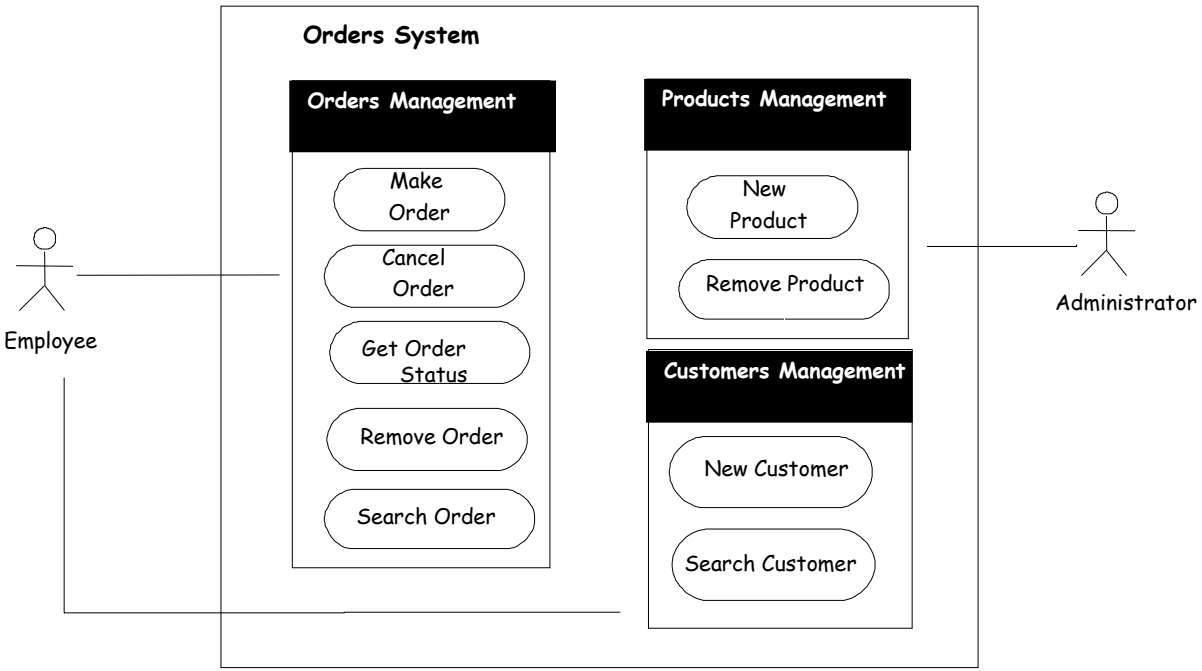
Context Diagram

- It shows the boundaries of the system and the interacting actors

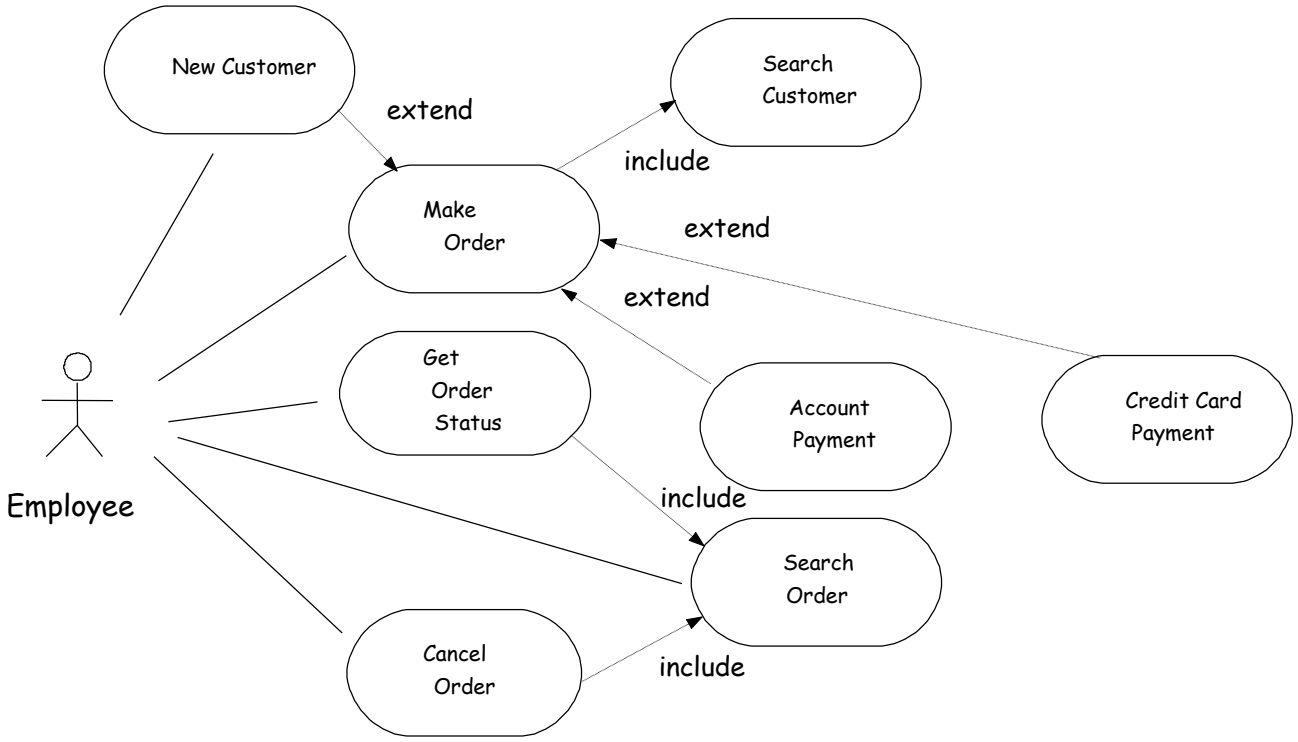


Initial Diagram

- It contains a grouping of the main use cases



Structured Diagram

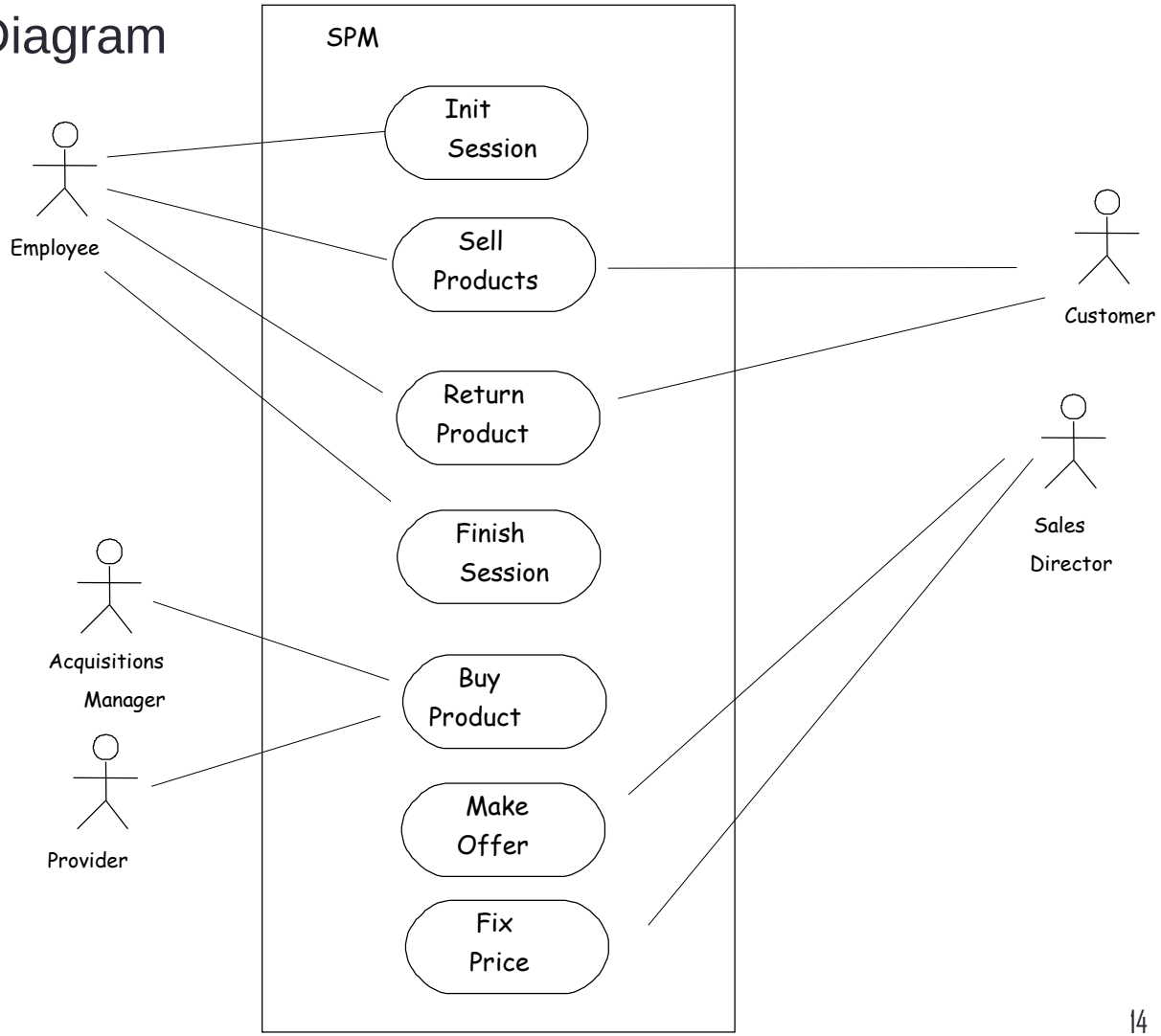


(... the model is incomplete)

Example SPM

Sales Point Machine

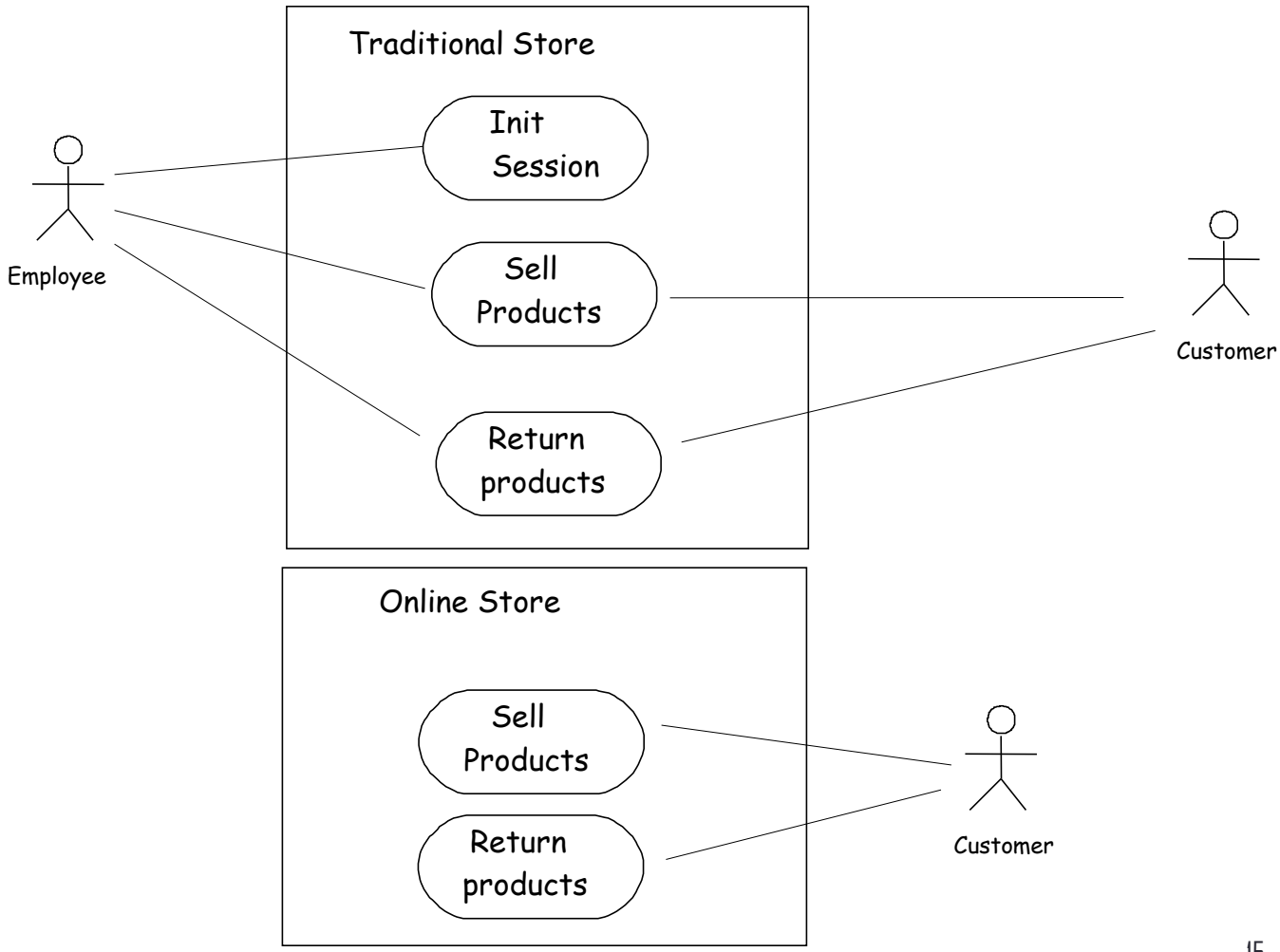
Context and Initial Diagram



. . . Example SPM

Sales Point Machine

Variations

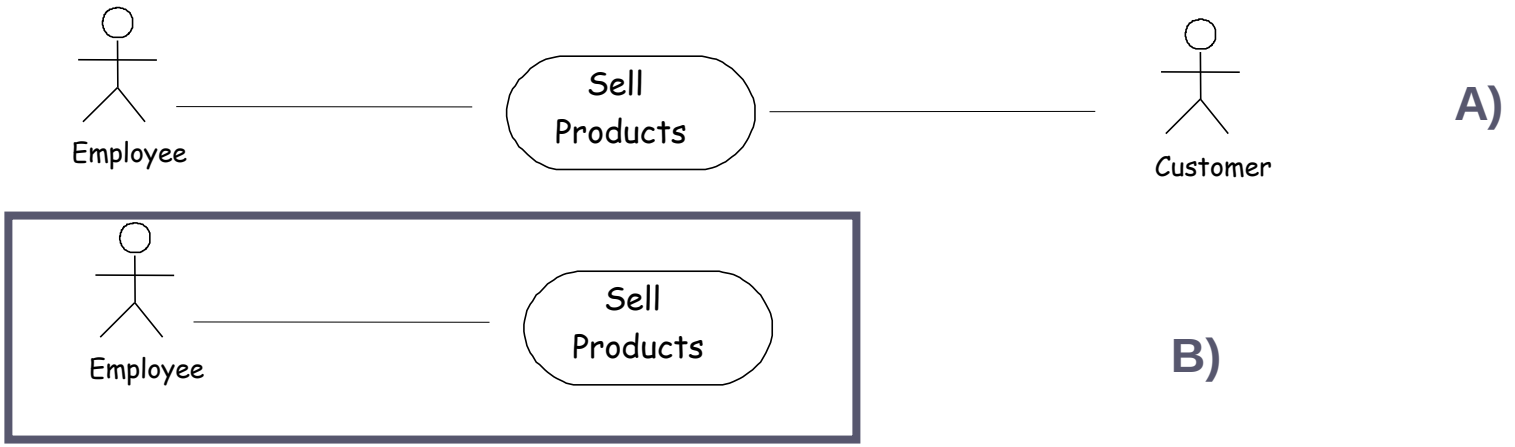


. . . Example SPM

Sales Point Machine

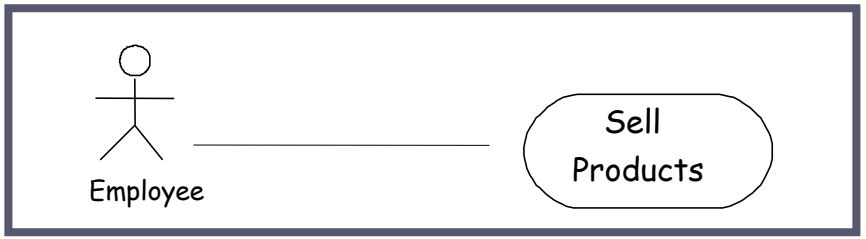
Variations on the example

If only actors interacting with the computer-based system are shown.



. . . Example SPM

Sales Point Machine



B)

Description template

Use Case	Sell Products
Actors	Employee (initiator)
Goal	Capture a sale and its cash payment
Summary	A customer arrives to the sales point with products. The employee registers products and manages payments in cash. The customer leaves with the products.
Preconditions	The employee is logged in the system.
Postconditions	The sale is stored in the system.
Includes	-
Extends	-
Inherits from	-

. . . Example SPM

Sales Point Machine

. . . Description template

	User Intentions	**System Obligations**
	1. The employee indicates a new sale starts.	2. The system records the start of a new sale
	3. The employee inserts the code of the product and the quantity	4. The system calculates the cost of the product and adds the information to the bill.
	5. The employee indicates the end of the sale	6. The system calculates and shows the total cost.
	7. The employee indicates the money received	8. The system calculates and shows the change. It prints a receipt and records the sale.
	Synchronous Extensions	
	#1. If at step 3 a code of a non-existent product is inserted the system generates an error message.	
	#2. At 7 the employee may cancel the sale.	
	Asynchronous Extensions	
	None	

Description Styles

Get Cash,
(Concrete Use case)

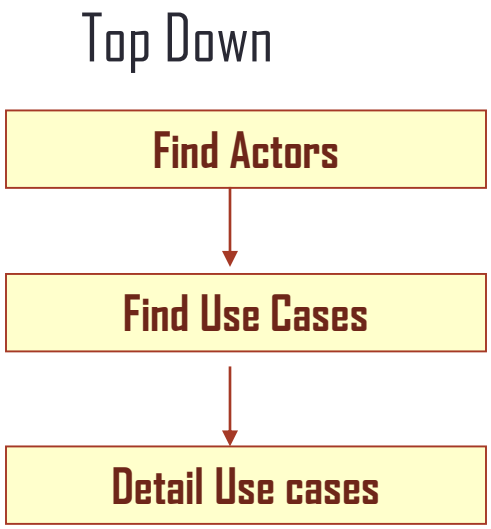
User Action	System answer
Insert card	
	Read magnetic band
	Request PIN
InsertPIN	
	Verify PIN
	Show Main Menu
Press Key	
	Show Accounts Menu
Press Key	
	Request Ammount
Insert Ammount	
	Show Ammount
Press Key	
	Return card
Take Card	
	Give Money
Take Money	

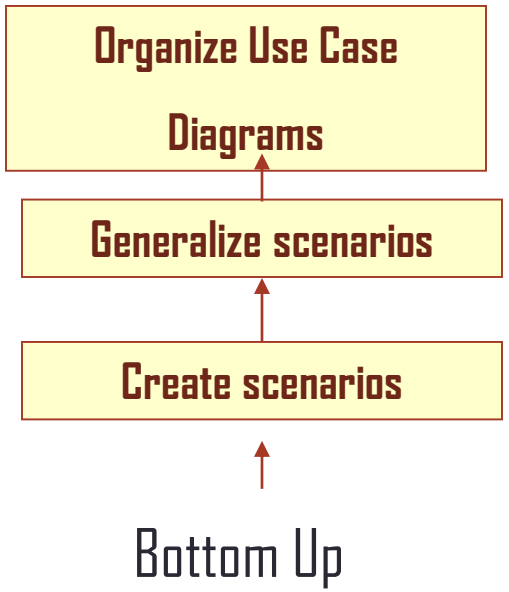
Get Cash,
(Essential use case)

User Intention	System Responsability
Identify	
	Verify identity
	Show Operations
Selection	
	Give Money
Take Money	

Building the diagram

- Top-down technique
- Bottom-up technique





Building the diagram

Rules to find Actors

- Users play roles when interacting with the system
- A user may correspond to many roles

- Any group or individual in one of the following categories:
  - Who will use the system?
  - Who will install the system?
  - Who will maitain the system?
  - Who will switch off the system?
  - What other systems will communicate with this one?
  - Who gets information?
  - Who provides information?

Building the diagram.

Rules to identify Use Cases

- Paying attention to actors
  - What are the tasks required by actors from the system?.
  - Will an actor be able to create, store, change or remove information from the system?.
  - Will an actor inform to the system about changes occuring outside?.
  - Will any actor be informed about state changes of the system?.

The answers to these questions represent flows of events that are associated to potential candidate use cases.

BUSINESS LOGIC DESIGN

Chapter 5

Software Engineering
Computer Science School
DSIC – UPV

Goals

- Understand the software design as a set of objects that interact with each other and manage their own state and operations.
- Learn how to derive a design model from a class diagram.
- Learn how to derive methods from sequence diagrams.

References

- Weitzenfeld, A., Ingeniería del Software OO con UML. Java e Internet. Thomson, 2005
- Stevens, P., Pooley, R. Utilización de UML en Ingeniería del Software con Objetos y Componentes. Addison-Wesley Iberoamericana 2002.
- . . .

Contents

Introduction

Conceptual Modeling *(Analysis)*

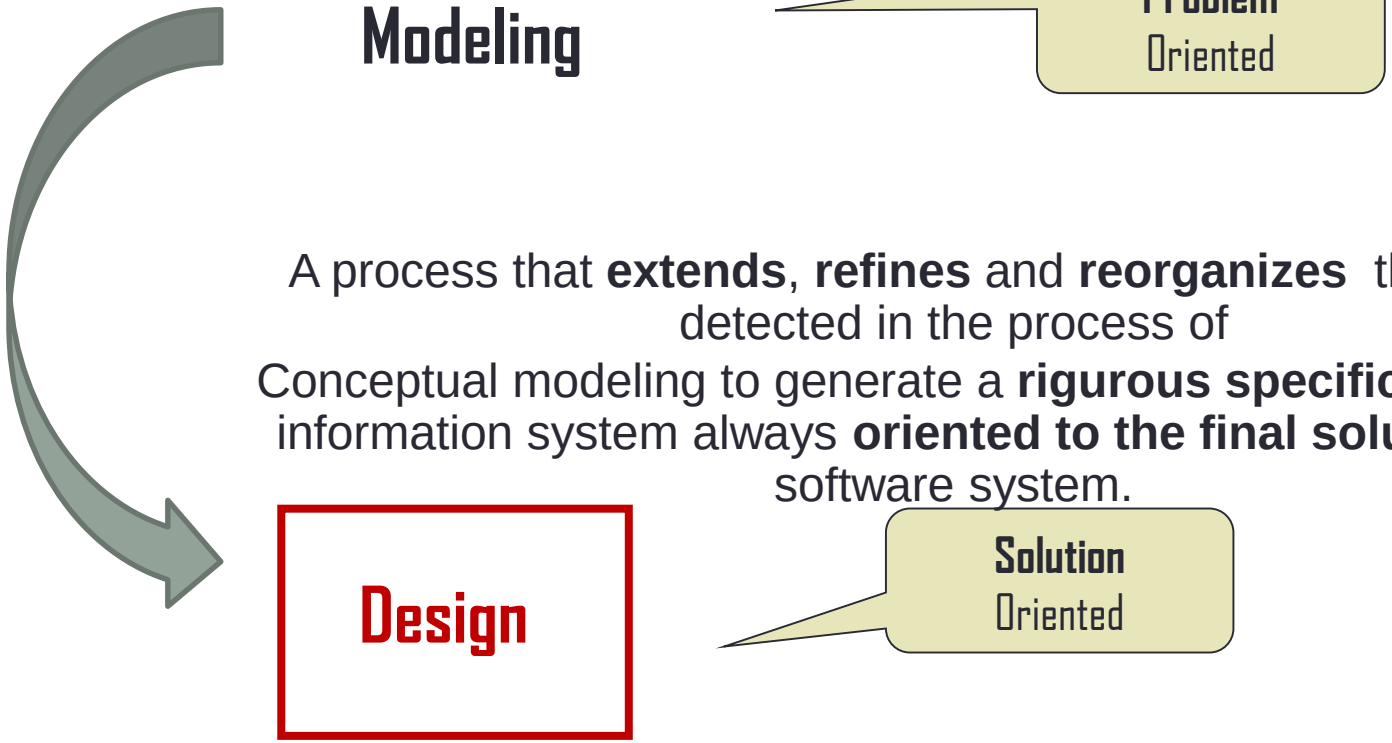
It is the process of constructing a **model** / of a detailed specification of
A problem of the real world we are confronted with.
It does not **contain** ***design and implementation*** elements

Modeling = Design?

NO

Introduction

Modeling *vs.* Design



The design adds the development environment and the implementation language as elements to consider.

OBJECTS DESIGN

Objects Design

- Input: Conceptual Modeling – **Class diagram**

** Refine Analysis class diagram

- Output: Design – **C# Design**

Design of Classes

Design of Associations

Design of Aggregations

Design of Specializations

Objects Design

** Refine analysis class diagram

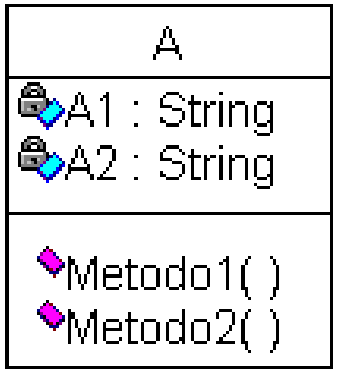
Design decisions

- ✓ Create new classes
- ✓ Remove/Join classes
- ✓ Create new relationships
- ✓ Modify existing relationships
  - ✓ Restrict navegability
- ✓ . . .

Design Class Diagram

Design Patterns. Classes (1/2)

Conceptual Modeling



Design

```
class A
{
 private String A1;
 private String A2;

 public int Metodo1(){…}
 public String Metodo2() {…}

 public void setA1(string a) {…}
 public void setA2(string a) {…}
 public String getA1() {…}
 public String geA2() {…}

}
```

Classes (2/2)

design

```
public int Metodo1()
{
   ...
   return ...
}

public String Metodo2()
{
   ...
   return ...
}

 public void setA1(string a){ A1=a;}

 public void setA2(string a){ A2=a;}


 public String getA1(){return A1;}

 public String getA2(){return A2;}
```

Design Patterns. Associations (1/10)

1-to-1 Relationship

Conceptual Modeling

design

```
class A
{
   private B Rb;
   public void setRb(B vB){…}
   public B    getRb(){…}
}
```

```
class B
{
   private A Ra;
   public void setRa(A vA){…}
   public A    getRa(){…}
}
```

Associations (2/10)

1-to-1 Relationship

Design

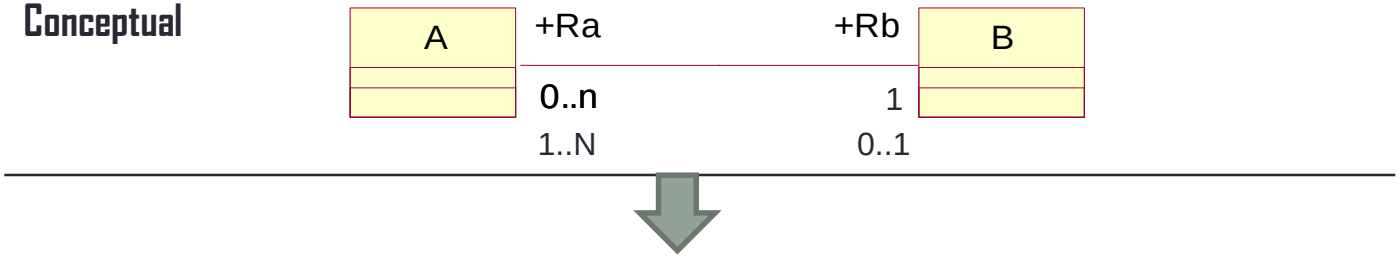
```
public void setRb(B vB)
{
   Rb=vB;
}

public void setRa(A vA)
{
   Ra=vA;
}
```

```
public B getRb()
{
   return Rb;
}

public A getRa()
{
   return Ra;
}
```

Associations (3/10)

1-to-Many Relationship

Modelado Conceptual



Design

```
class A
{
...   // same pattern as before with cardinality 1
}

class B
{
 private ICollection<A> collectionRa; // (Interface Collections)
 public A getA(string identifier_A){…}
 public void add_A(A objetoA){…}
 public void remove_A(A objetoA){…}
 // or remove_A(string identifier_A){…}
}
```

```
// Methods manipulating
// the A Collection
```

Associations (4/10)

```
class B
{
 private ICollection<A> collectionRa; // if it is List: List,
LinkedList

public void add_Ra(A objectA)
{
    collectionRa.Add(objectA)
}

public void remove_Ra(A objectA)
{
    collectionRa.Remove(ObjectA)
}
 . . .

]
```

SortedList, Stack, Queue, Dictionary, SortedDictionary, HashSet…

Associations (5/10)

The design of the **Associations**
Will be as defined depending on the max cardinality being 1 or N.

Associations (6/10)

Many-to-Many Relationship

Conceptual Modeling

Design

```
class A
{
 private ICollection<B> ColectionOfB //Set, Queue, Map, …  A []
 public B get_B(string identifier_B){…}
 public void add_B(B objectB){…}
 public void remove_B(C objectB){…}
}

class B
{
 private ICollection<A> CollectionOfA //Set, Queue, Map, …  A []
 public A get_A(string identificador_A){…}
 public void add_A(A objectA){…}
 public void remove_A(A objectA){…}

}
```

Associations (7/10)

1-1 Association (Association Class)

Conceptual Modeling

Design

```
class A
{
   private C The_C;
   public void setC(C vC){…}
   public C getC(){…}
}
```

```
class B
{
 private C The_C;
   public void setC(C vC){…}
   public C getC(){…}
}
```

Associations (8/10)

1-1 Association (Association Class)

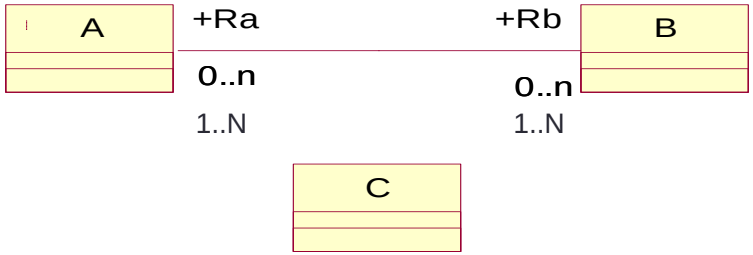
Design

```
class C
{
   private A Ref_A;
   private B Ref_B;

   public void setA(A vA){…}
   public A getA(){…}

   public void setB(B vB){…}
   public B getB(){…}
}
```

Associations (9/10)

Many-to-Many Association (Association Class)

Modelado Conceptual



Design

```
class A
{
 private ICollection<C> CollectionOfC; //Set, Queue, Map, …  A []
 public C get_C(string identifier_C){…}
 public void add_C(C objectC){…}
 public void remove_C(C objectC){…}
}

class B
{
 private ICollection<C>  CollectionOfC; //Set, Queue, Map, …  A []
 public C get_C(string identifier_C){…}
 public void add_C(C objectC){…}
 public void remove_C(C objectC){…}
}
```

Associations (10/10)

Many-to-Many Association (Association Class)

Design

```
class C
{
   private A Ref_A;
   private B Ref_B;

   public void setA(A vA){…}
   public A getA(){…}

   public void setB(B vB){…}
   public B getB(){…}
}
```

Design Patterns. Aggregation/Composition (1/2)

1-1 Aggregation

Conceptual Modeling

Design

```
class A
{
        private   B Rb;
        public void setRb(B vB){…}
        public B getRb(){…}
}

class B
{
        private   A Ra;
        public void setRa(A vA){…}
        public A getRa(){…}
}
```

Aggregation/Composition (2/2)

Aggregation 1-Many

Conceotual Modeling

Aggregation Many-Many

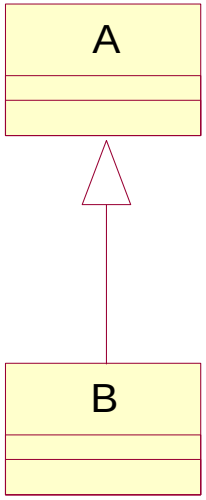
Conceptual Modeling

```
....  // same patterns already discussed
```

Specialization/Generalization

Simple Specialization

Conceptual Modeling



Design

```
class A
{
 ...
}

class B : A
{
 ...
}
```

DESIGN OF MESSAGES

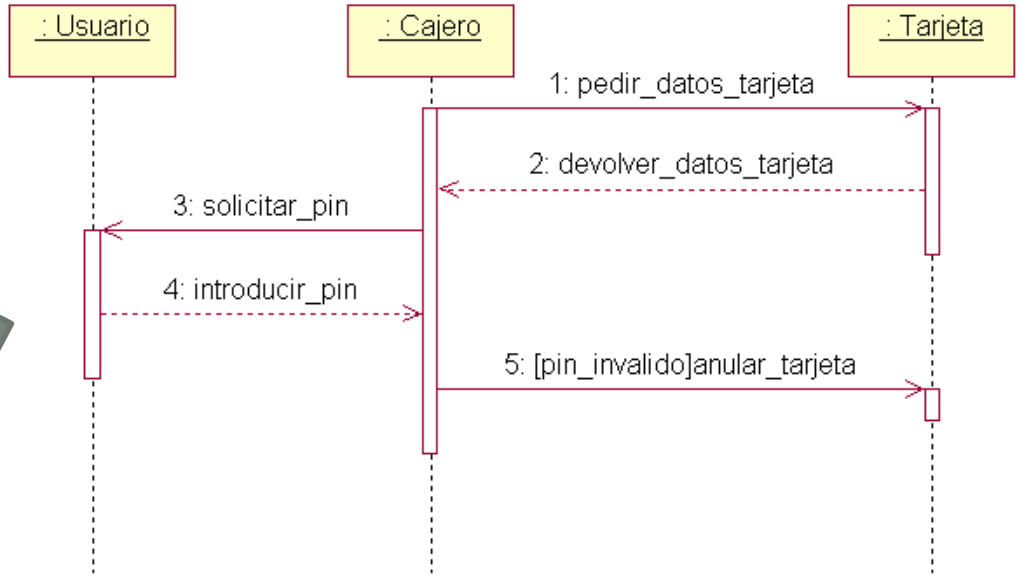
Design of Messages

- *Input:* **Sequence Diagram**
  - Identify communication within the system
- *Output:* Obtain operations (**methods**) of classes

- If an object receives a message → its class will offer a method to serve the message
- If an object sends a message → the invocation of the method of the destination object will be in the body of a method of the sender object.

Design of Messages

Scenario: Pin Update



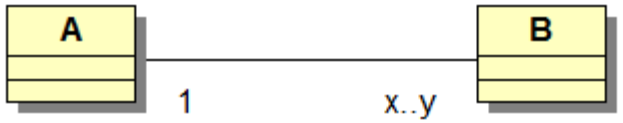

```
class ATM
{
 private . . .
. . .
 public void change_pin()
  { . . .
    ObjTarjeta.get_card_data(. .  .);
    . . .
    if (pin_invalid){
    ObjTarjeta.cancel_card (. . .);}

  }}
```

```
class Card
{
   private . . .;

   . . .
   public void get_card_data(. . .){…}
   public void cancel_card(. . .){…}

}
```

Constructors Implementation

Considerations about constructors (1/2)

- Initializing an object results in giving values not only to attributes but also to links with objects of other classes.
- The minimum cardinality of associations/aggregations determines how the initialization is done.



x	y	Declaration	Constructor	Inicialization
0	1	B the_B	A(...)	null
1	1		A(..., B the_B, ...)	this.the_B=the_B
0	*	ArrayList<B>the_Bs	A(...)	the_Bs=new ArrayList<B>
1	*		A(..., B the_B, ...)	the_Bs=new ArrayList<B>; the_Bs.add(the_B)