Lesson 1: The Software Life-Cycle

Software Analysis and Design

2nd Year, Computer Science

Universidad Autónoma de Madrid

Index

- Introduction
- Phases and the software life-cycle
- Life-cycle models
- Methodologies

What is Software?

- Software =

  programs + data + documentation

- All of which needs to be developed, deployed and **maintained**

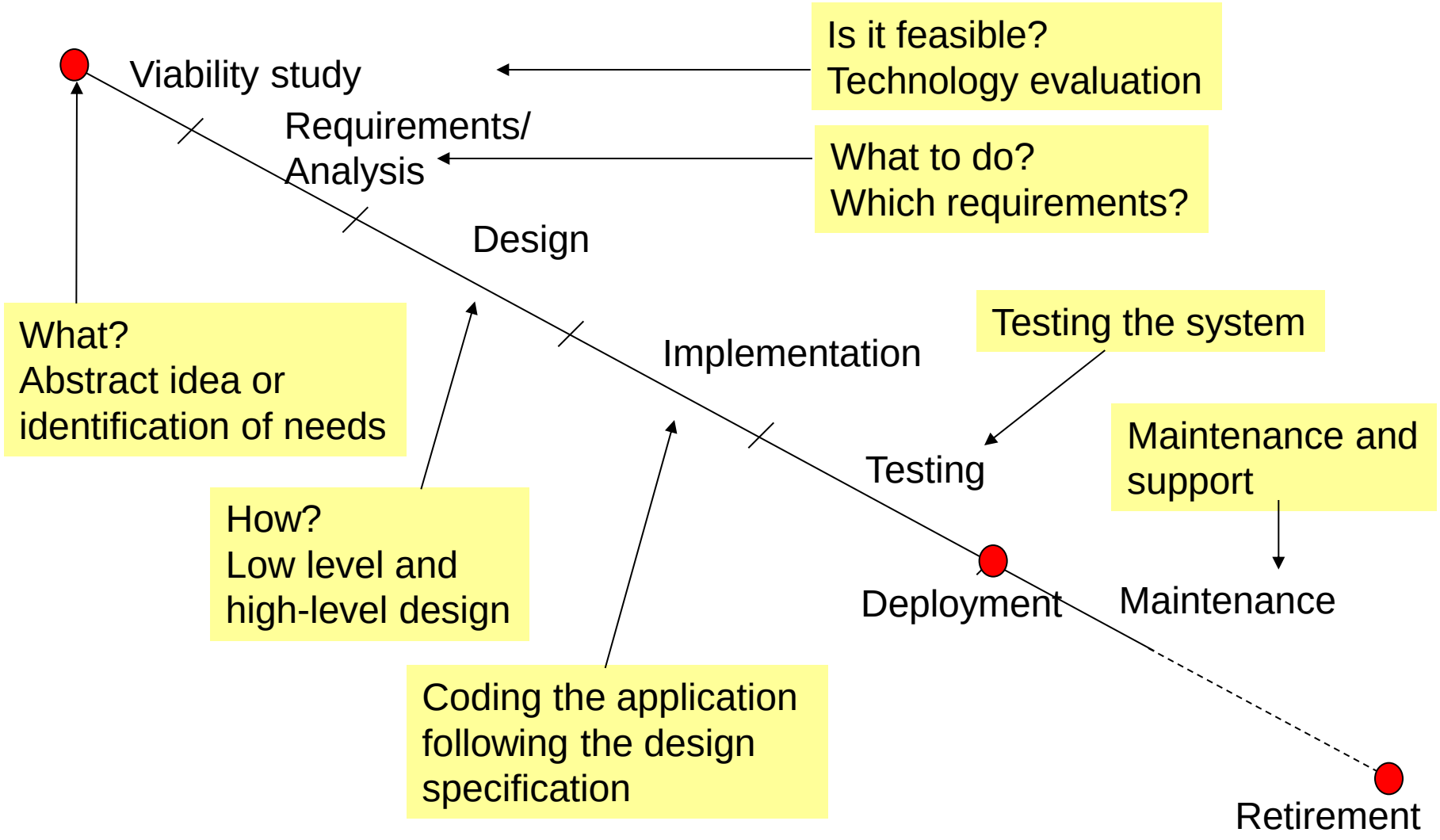
Software life-cycle

- Building software is not only about programming
- Additional phases: feasibility (viability) study, requirements, analysis, design, coding, testing and maintenance
- All these phases are enacted (executed) in conformance to a project plan
- Similar to other (engineering) disciplines
  - building a car is not just welding metal and tightening screws
  - building a house is not just laying bricks
  - …

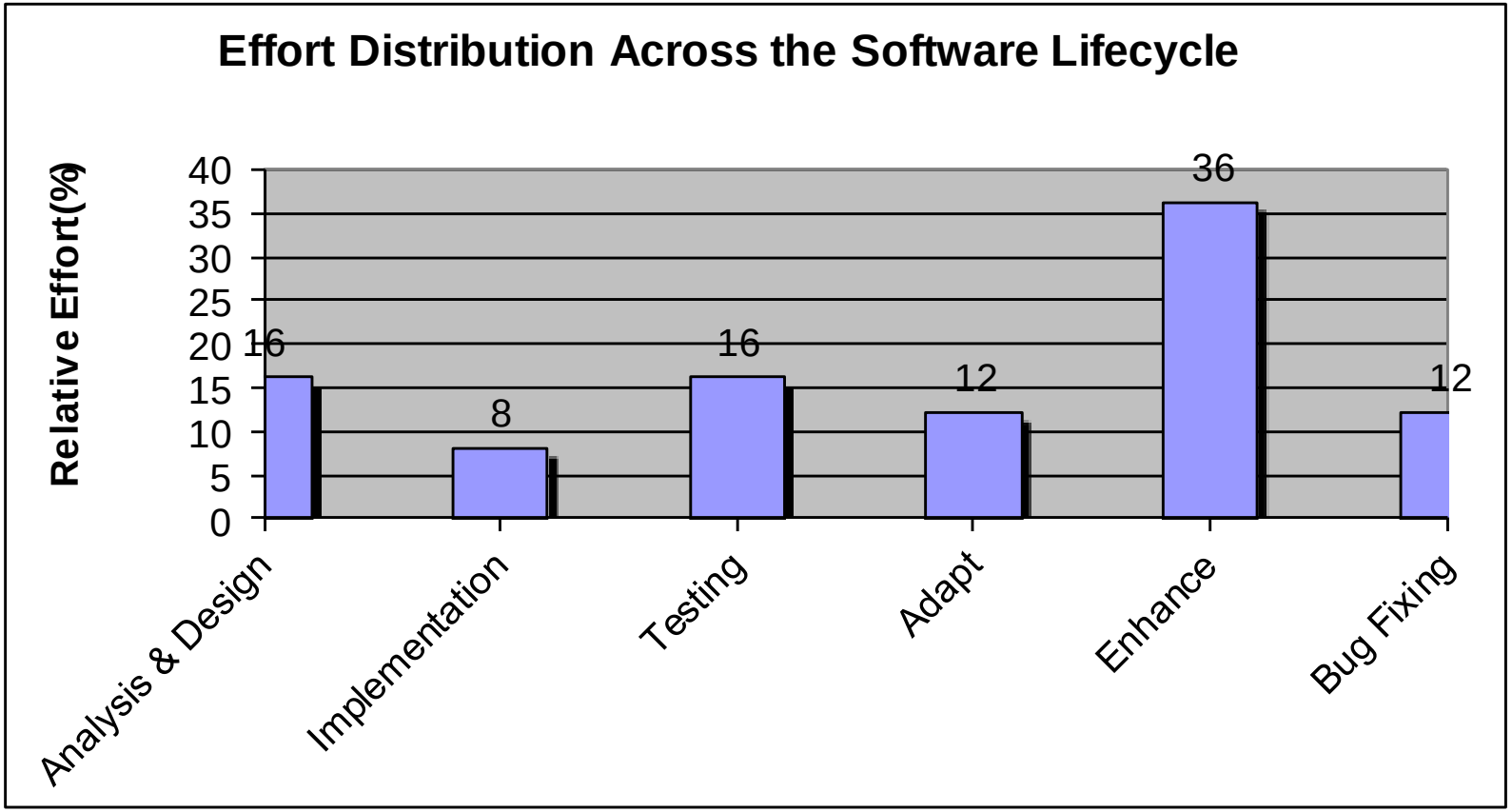
Software life cycle

- Phases the software goes through
  - from the moment its conception starts
  - until it is retired

Software life-cycle



Software life-cycle



Feasibility study

- Technical, operational and economic analysis prior to a project to determine if it is profitable and decide if the problem is approachable

- Basis for decision-making about the continuity or not of the project, as well as for the analysis of the ***risks*** involved in its execution

Feasibility study

- **Technical** feasibility
  - Analysis of the functionality, performance and constraints
- **Economical** feasibility
  - Cost/benefit analysis
- **Legal** feasibility
  - Violation or illegality resulting from the development of the system
- **Alternatives**
  - Evaluation of possible alternatives for system development

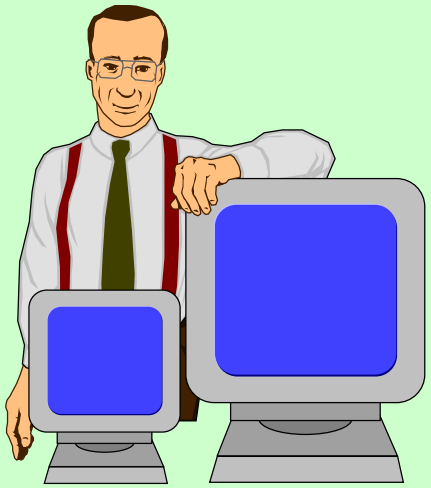
Requirements analysis

- Find answers for the following questions
  - **What** needs to be done?
  - **What** functionality do we need to implement?
  - **What** are the non-functional requirements? (e.g. performance, reliability)

"*Problem analysis and complete specification of the external behaviour of the software system to be built, and the flow of information and control.*"

Requirements analysis

Customers and users describe their current problem, and the results and conditions they expect.



The software engineer makes questions, analyses, assimilates and presents the right solution.

Output: Requirements Analysis Document (RAD)

Requirements analysis

Tasks

- Requirements **Elicitation (gathering)**
  - Identify requirements obtained from users and customers
- **Analysis** of the problem and the requirements
  - Reasoning about requirements, combining related requirements, prioritize them, determine their feasibility, etc
- **Representation** (modelling)
  - Register requirements using some method, including natural language, formal languages, models, mockups, etc
- **Validation**
  - Detect inconsistencies between requirements, determine their correctness, ambiguity, etc. Establish criteria to ensure that the software meets the requirements as soon as possible (criteria which the client, user and developer should agree on)

Requirements analysis

Types of requirements

- Functional and non-functional
- Functional requirements
  - Fundamental actions that have to take place in the execution of the software
- Non-functional
  - Operational (eg., recovery, back-ups)
  - Security (access levels, security levels,…)
  - Mantainibility and portability
  - Resources (memory, storage, etc)
  - Performance (response time, number of users,…)
  - …

Requirements analysis

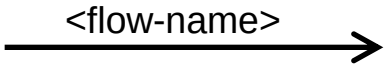
Representation

- Structured analysis
  - Analysis techniques oriented to **data**
  - Analysis techniques oriented to **functions**
  - Analysis techniques oriented to **states**
- Object-oriented analysis
- Formal languages
- Mock-ups (“*maquetas*”)

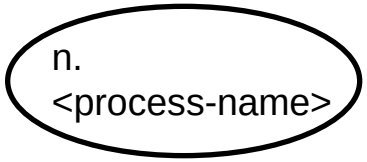
Structured analysis

Data-flow diagrams (DFDs)

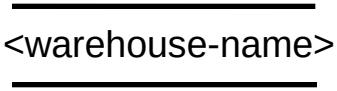
- Shows the information flow and the transformations of the data when moving from the input to the output
- Elements
  - Data flows
    - Sets of system values at a certain moment



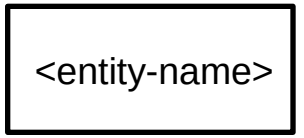
  - Processes
    - Transformations of the input flows into output flows



  - Data warehouses (datastore)
    - Data that should be persisted for their use by processes

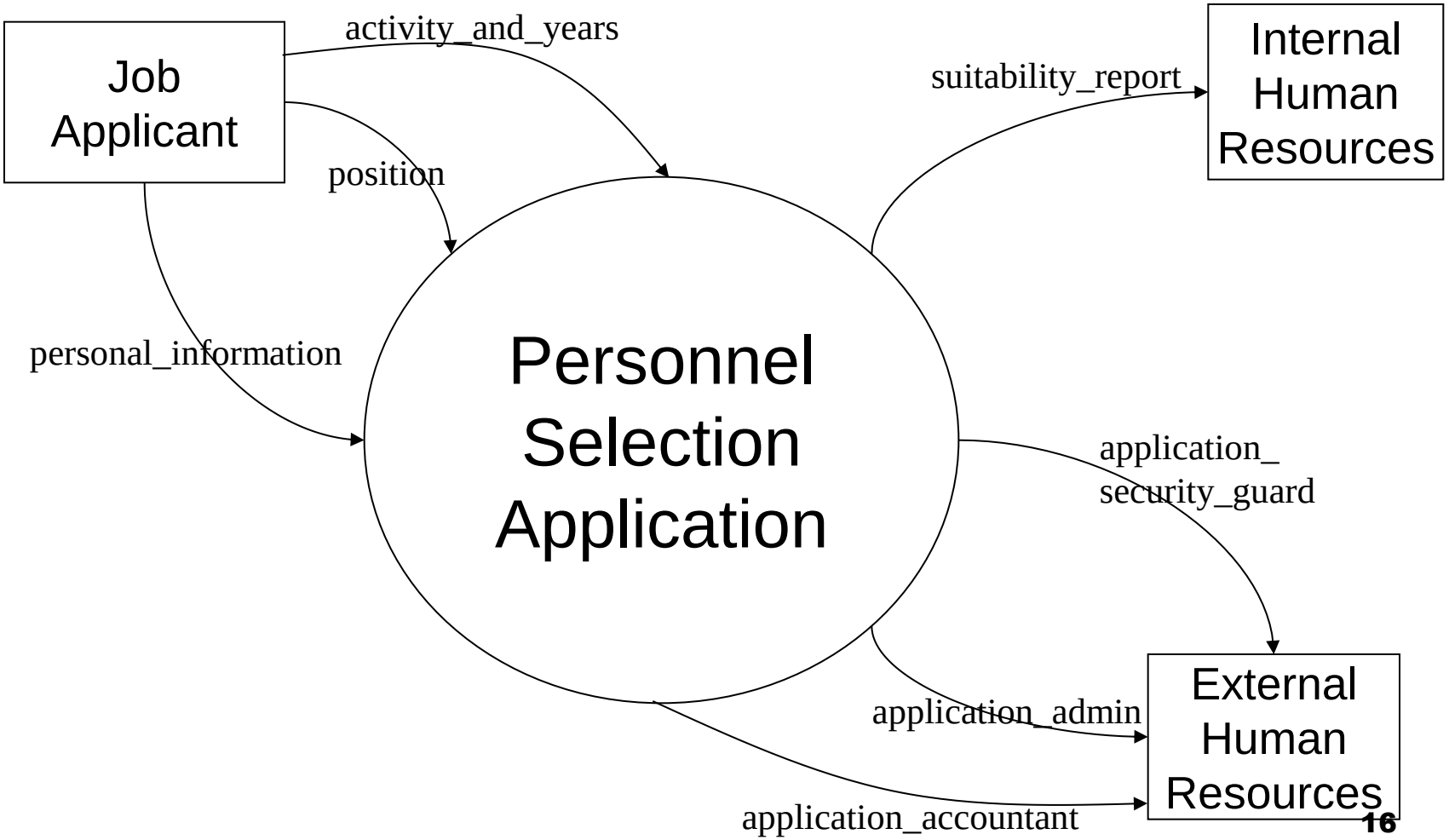


  - External entities
    - Data producer or consumer, which lives outside of the system the DFD describes, but that is related to it



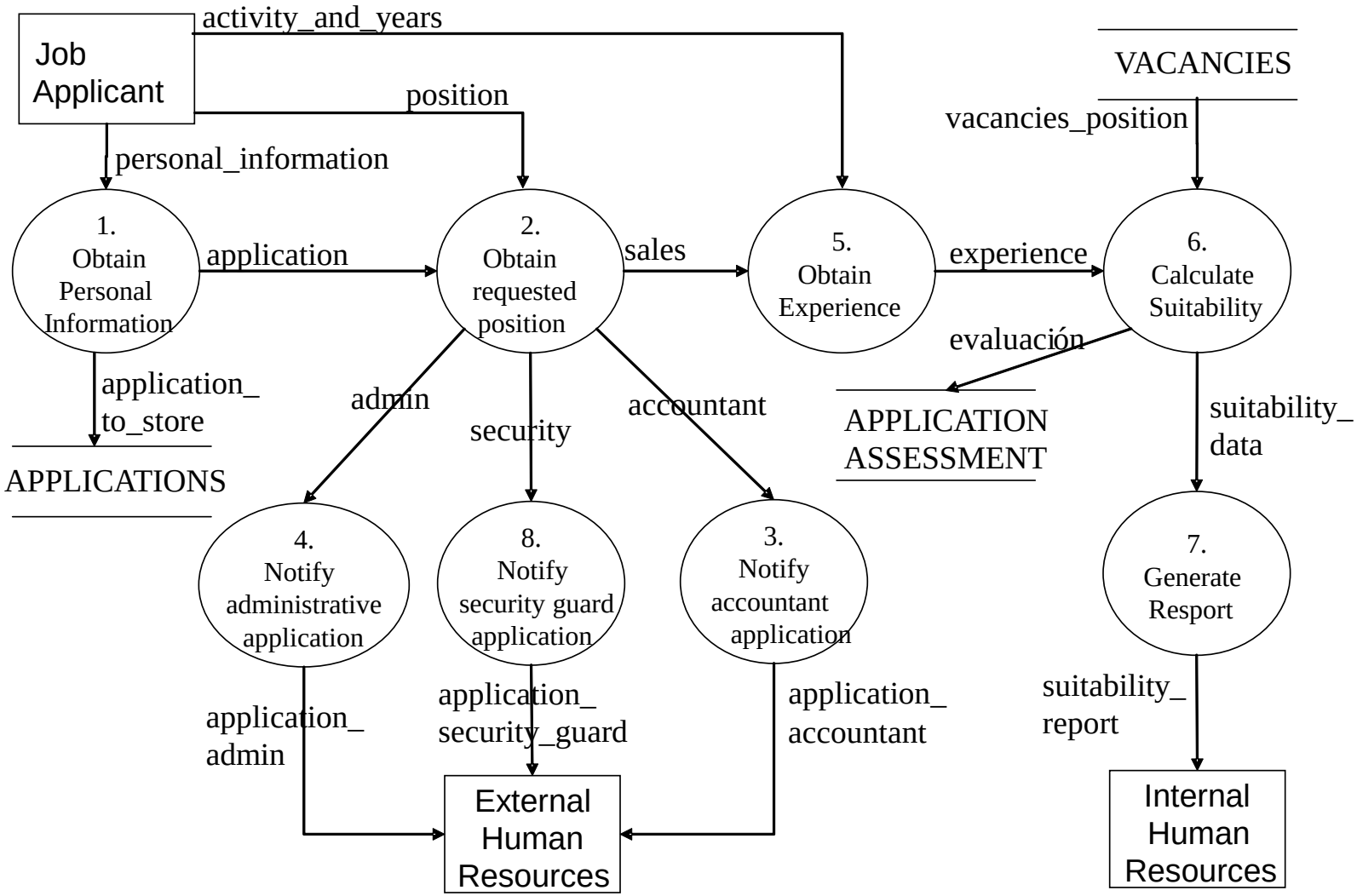
Data-flow diagrams

Level 0



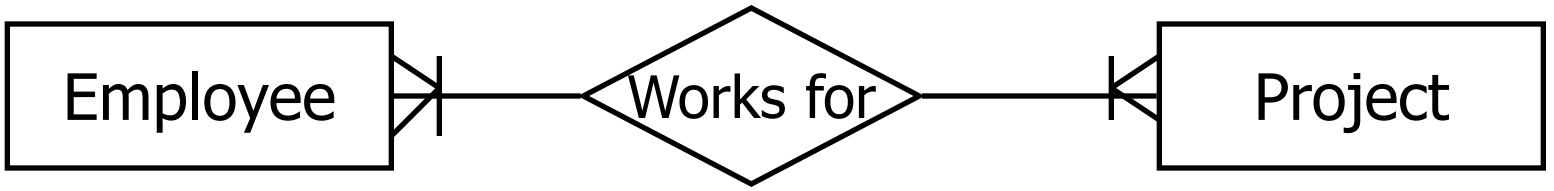
Data-flow diagrams

Level 1



Data

- Which information the application needs
- Data dictionary
- Entity-relatiomship diagrams



- Name
- ID
- Address
- …

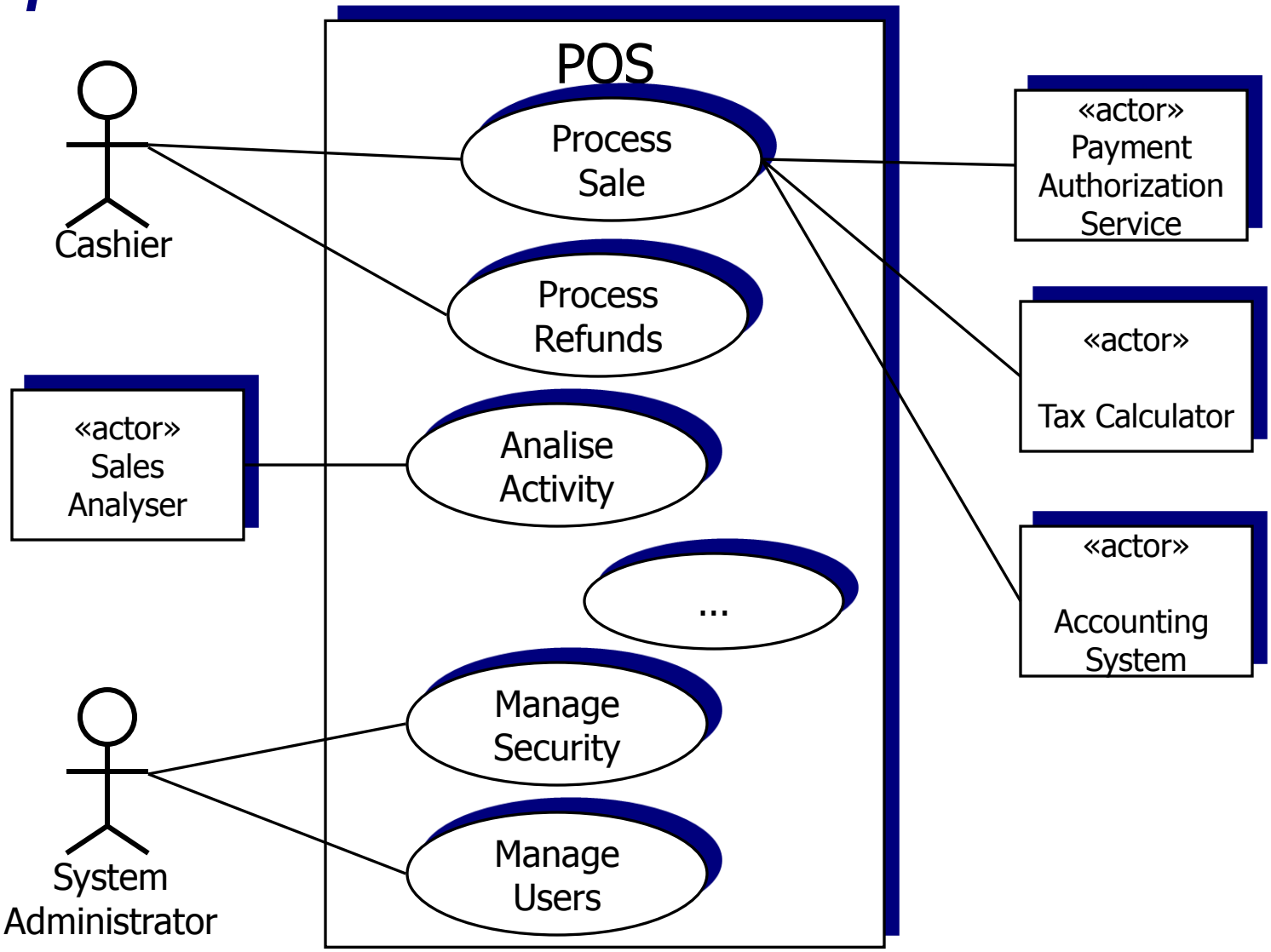
- Name
- Start date
- …

Object-oriented analysis

- **Use cases**: set of scenarios describing different ways to use the software, from the point of view of each type of user
- **Scenarios:** sequence of interactions that describe success or failure conditions (eg., errors)
- **Actors**: Active external elements (users, another system) that interact with the system

Requirements analysis

Representation: Use cases



Requirements Analysis

Representation: Use cases

USE CASE 1: Process sale

Primary actor: Cashier.
Preconditions: The cashier has been identified and authenticated.
Success guarantee (postconditions): The purchase is registered in the system. The applicable tax is calculated. The accounting system is updated. Commissions are recorded. A receipt is generated. Card payment approval is recorded.

Success path:

1. A customer arrives at the POS with goods or services to buy.
2. The cashier starts a new sale.
3. The cashier inputs the identifier of a product.
4. The system registers the product and presents a description, its price and the current total. The price is calculated from a list of rules.

The cashier repeats steps 3-4 until there are no more elements

5. The system displays the total, including taxes
6. The cashier tells the total to the client, and asks her to pay
7. The client pays and the system processes the payment
8. The system registers the completed sale and sends the information to the external accounting system.
9. The system generates a receipt.
10. The client leaves.

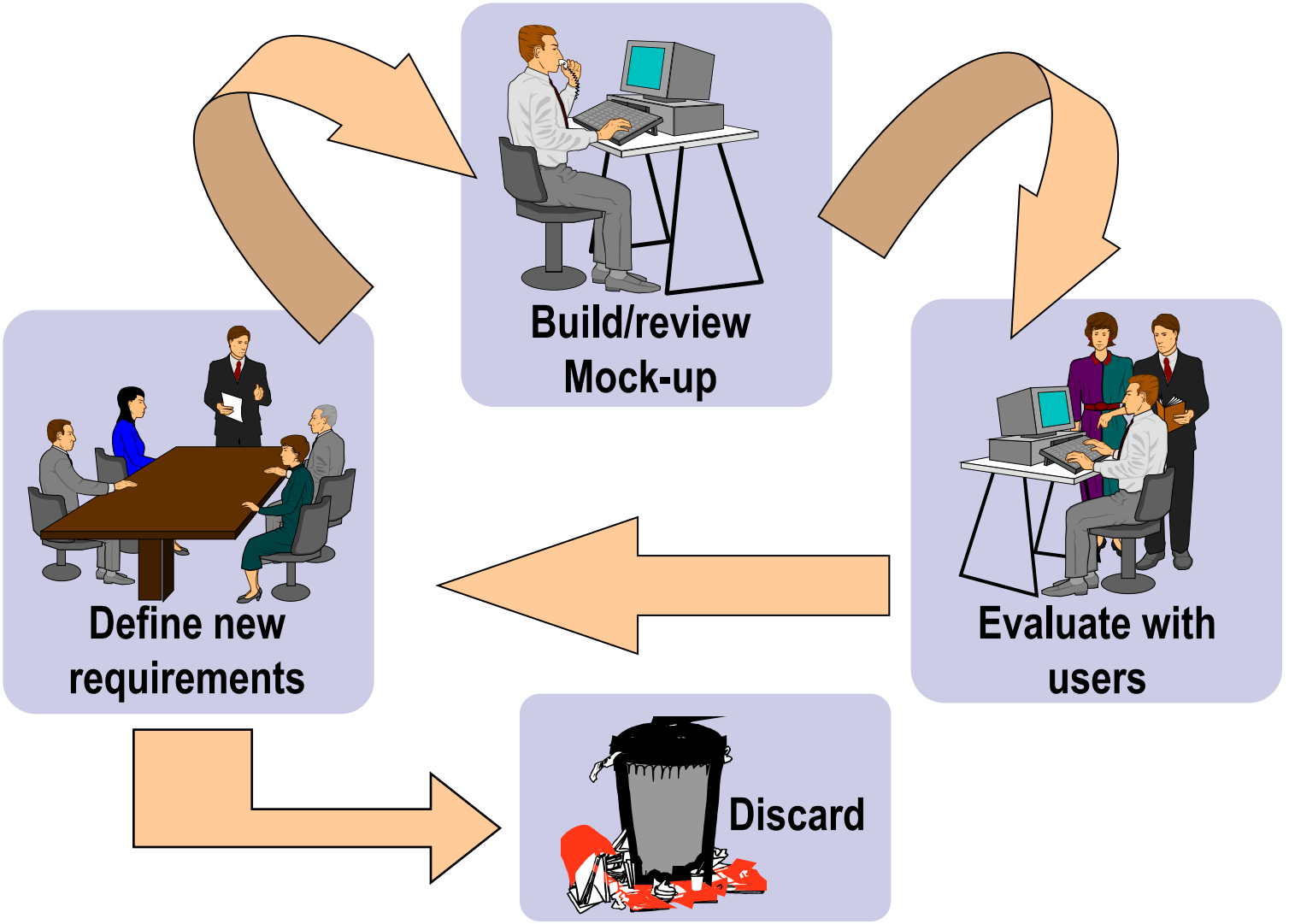
(in addition, we need to specify exceptions and alternative paths)

Mock-ups

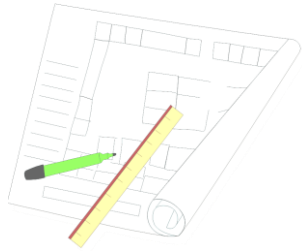
- Representation of requirements capture in the form of interfaces, which allow a better understanding with the user

- From drawing programs (powerpoint, visio) to specialized applications (ex: Mockupscreens, Balsamiq, etc)

Mock-ups: Life-cycle



Design



- Change the attention from **what** to do to **how** to do it

- From the problem domain to the solution domain

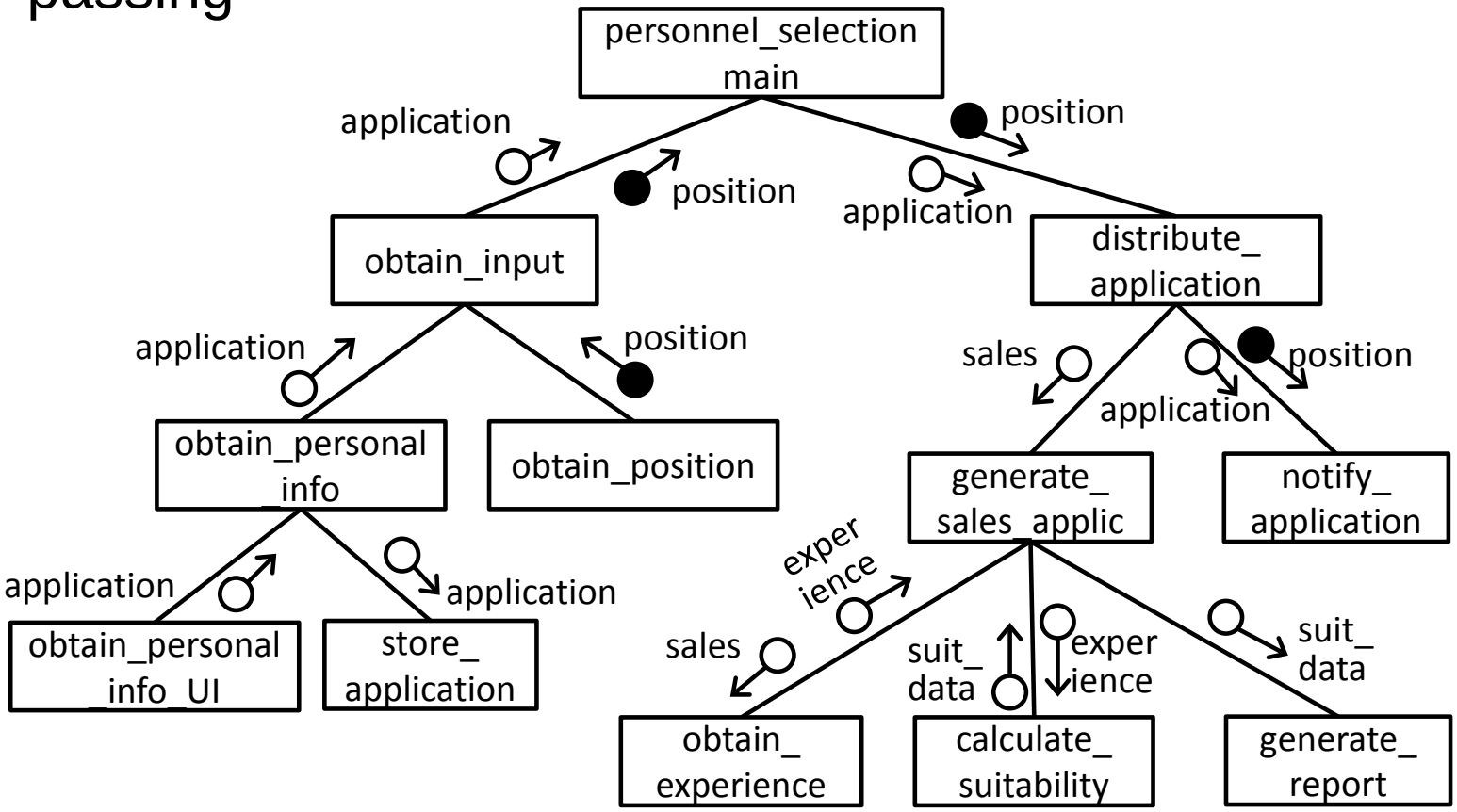
  "*It is the process of defining the architecture, components, modules, interfaces, test procedures and data of a software system to meet the specified requirements*."

Design

- **Architectural** design
  - Definition of the components of the system and its interfaces
- **Detailed** design
  - Detailed description of the logic of each module, the data structures they use and the control flows
- Basic principles
  - Abstraction
  - Refinement
  - Modularity
  - Information hiding
- Structured design vs. Object-oriented

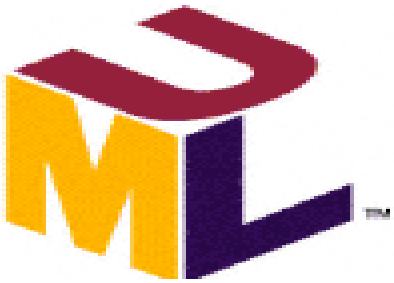
Structured design

- Hierarchy of calls between functions, and parameter passing



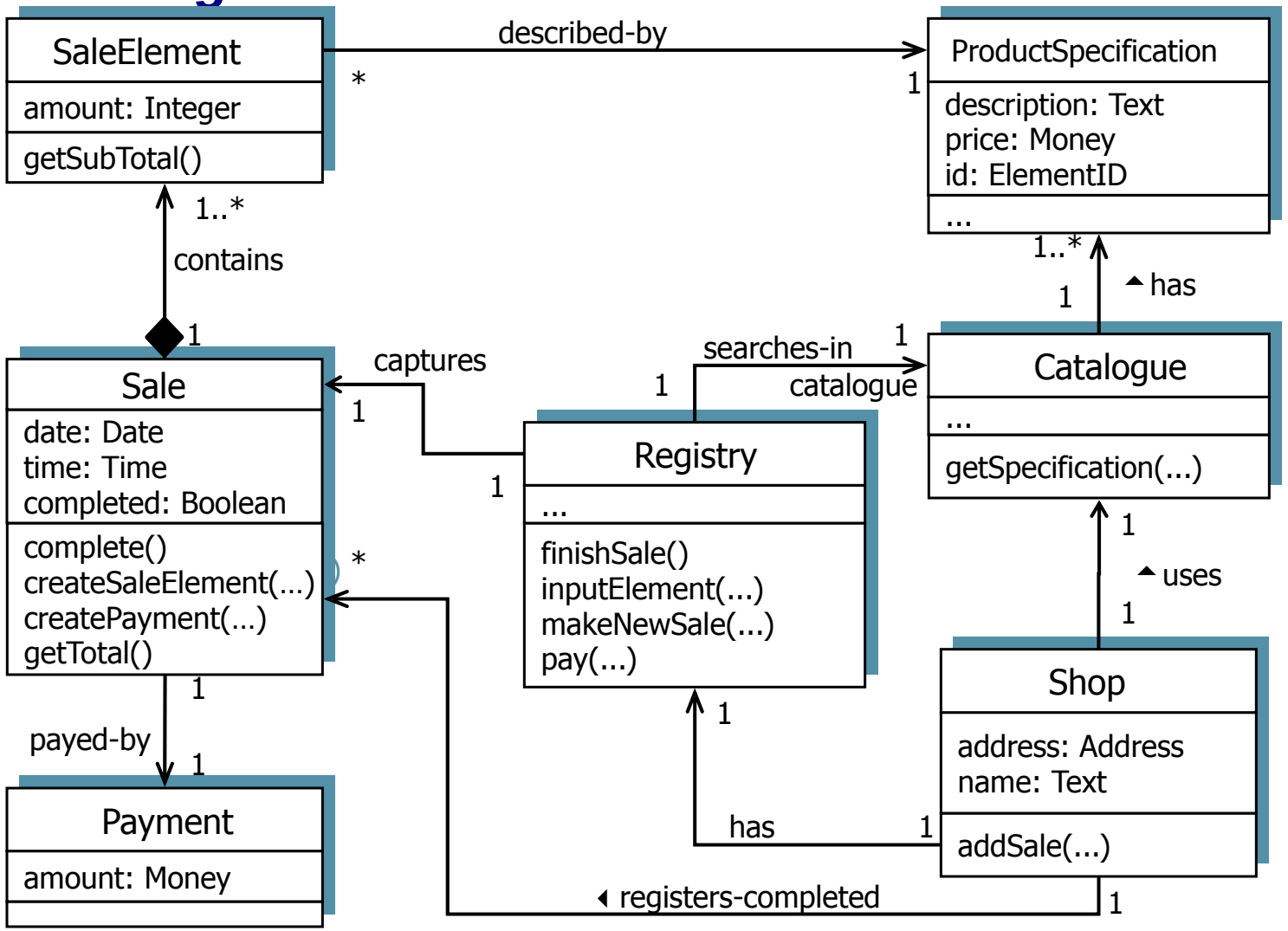
Object-oriented design

- Structure
  - Class design
- Behaviour
  - Of each class
  - Of object interactions
- A standard notation is UML (Unified Modelling Language)
  - http://www.uml.org



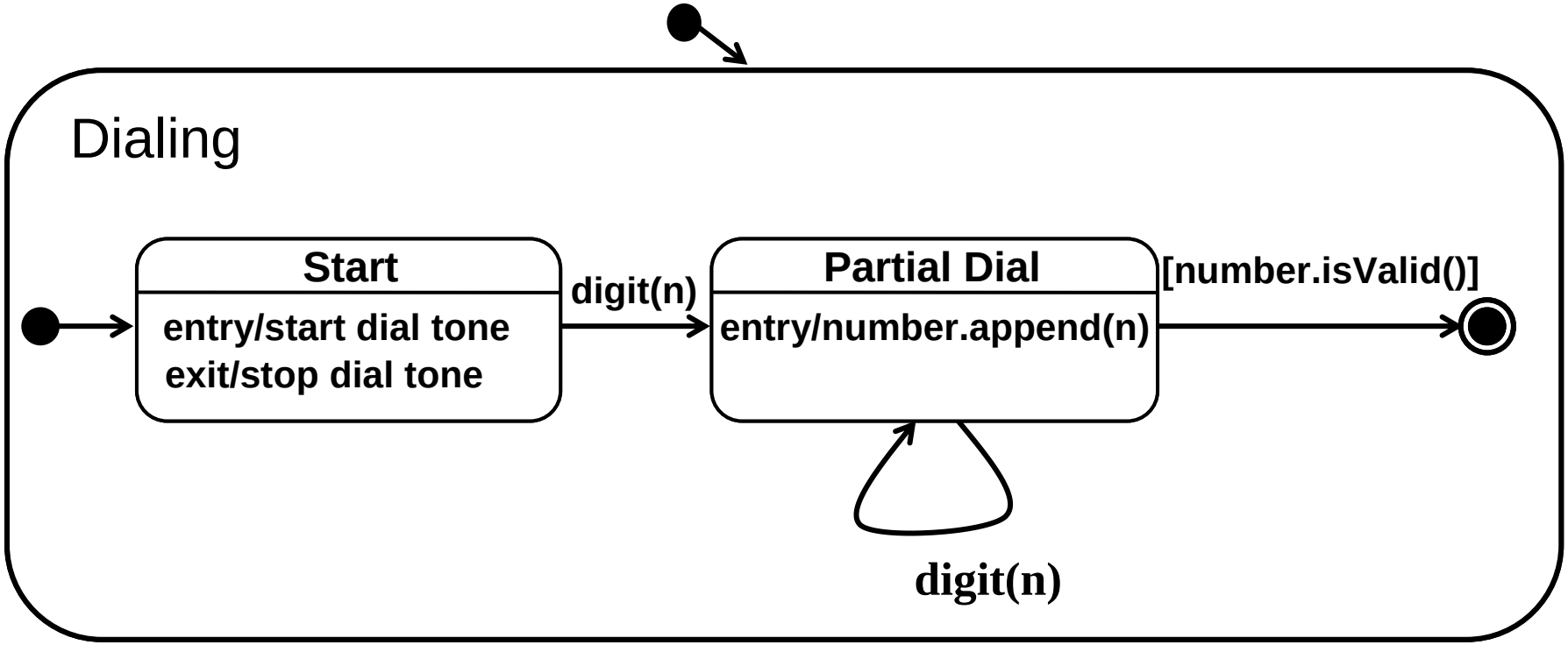
Object-oriented design

Class diagram



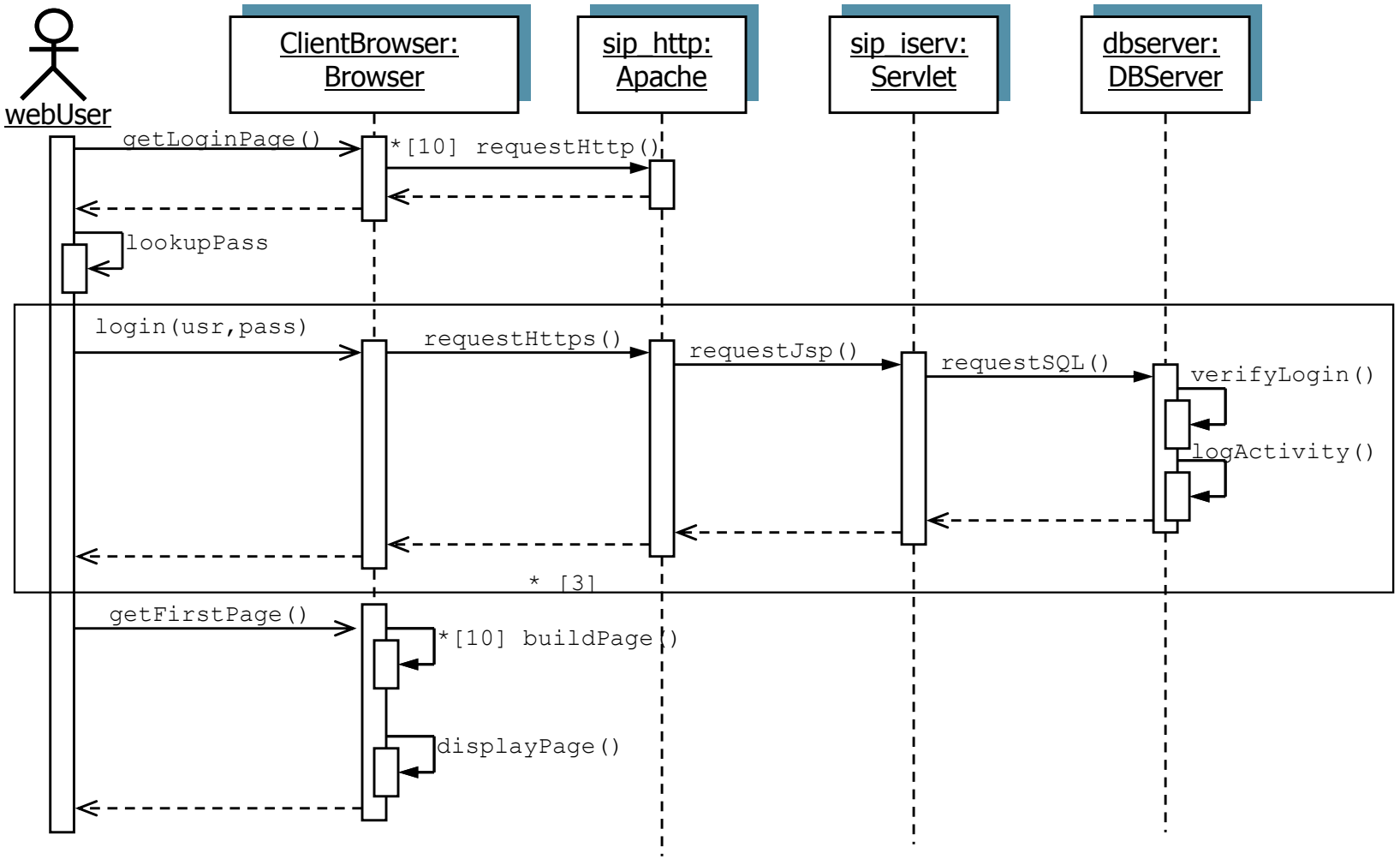
Object-oriented design

State machines



Object-oriented design

Sequence diagram



Coding

- Translate design specifications into an implementation language (C, Java, etc.)
- Also includes
  - Unit tests (e.g., tests of each function or method in isolation to check that they work properly)
  - Technical manual (e.j.: Javadoc)
  - User manual
- Style guides
- Structured programming vs. object-oriented

Testing

- Execute the program to find errors
  - **It is not** proving that there are no errors, or proving that the program works
  - Ensuring correctness is practically impossible for large programs
  - Design test cases maximizing the probability of finding errors
- Software error: when the program does not behave as expected, as stated in the requirements
  - Programming error
  - Communication problems with the users

Testing

Types

- **White box**
  - Exercise the program taking its logic into account
  - Coverability criteria
  - White box tests execute
    - all sentences (at least once)
    - all independet paths of each module
    - all boolean decisions
    - all loops

- **Black box**
  - Driven by the input/output data
  - Considers the software as a black box without taking its inner logic into account

Testing

Strategies

- **Unit tests**
  - Check the logic, and functionality of the basic building blocks (functions, methods)
- **Integration testing**
  - Consider groups of modules, and the flow of information between the interfaces
- **Validation testing**
  - Check the conformance of the software with respect to the requirements
- **System testing**
  - The application is integrated with its software/hardware environment
- **Acceptance testing**
  - Check that the product meets the user requirements

Maintenance

- Activities that the developer performs on the software once it is operational (after the delivery)

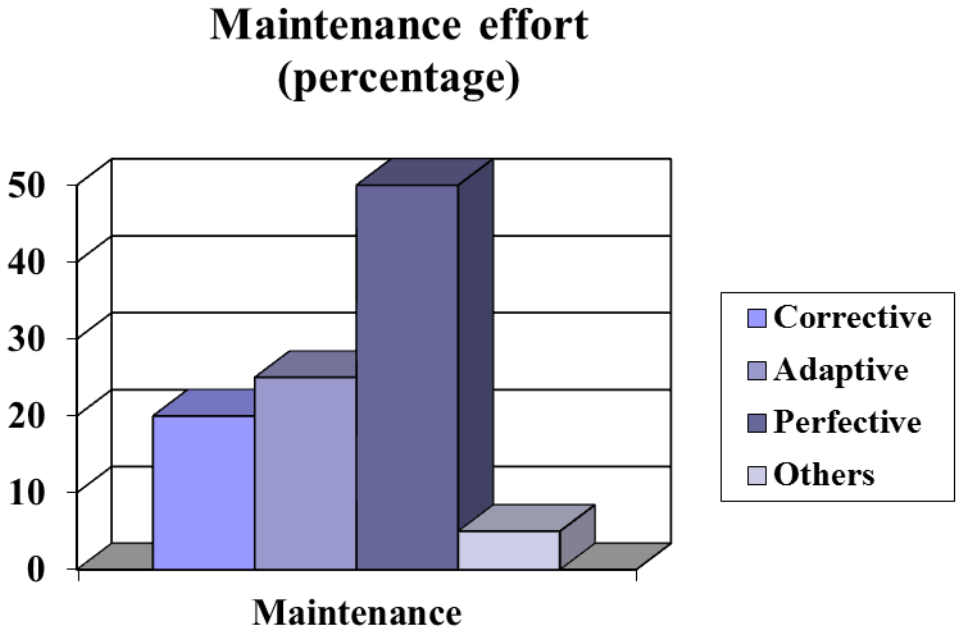
- Necessary modifications to meet new or old requirements

"The modification of a software product after its delivery, to correct errors, improve its performance or other attributes, or adapt the product to modifications of its operating environment"

Maintenance

Types

- **Corrective** (≅ 20%)
  - Fix errors
- **Adaptive** (≅ 25%)
  - Adapt to new environment
- **Preventive** (≅ 5%)
  - Prevent errors
  - Structural (type of preventive main.)
    - Modify the inner architecture
- **Perfective** (≅ 50%)
  - Improve, expand implemented requirements



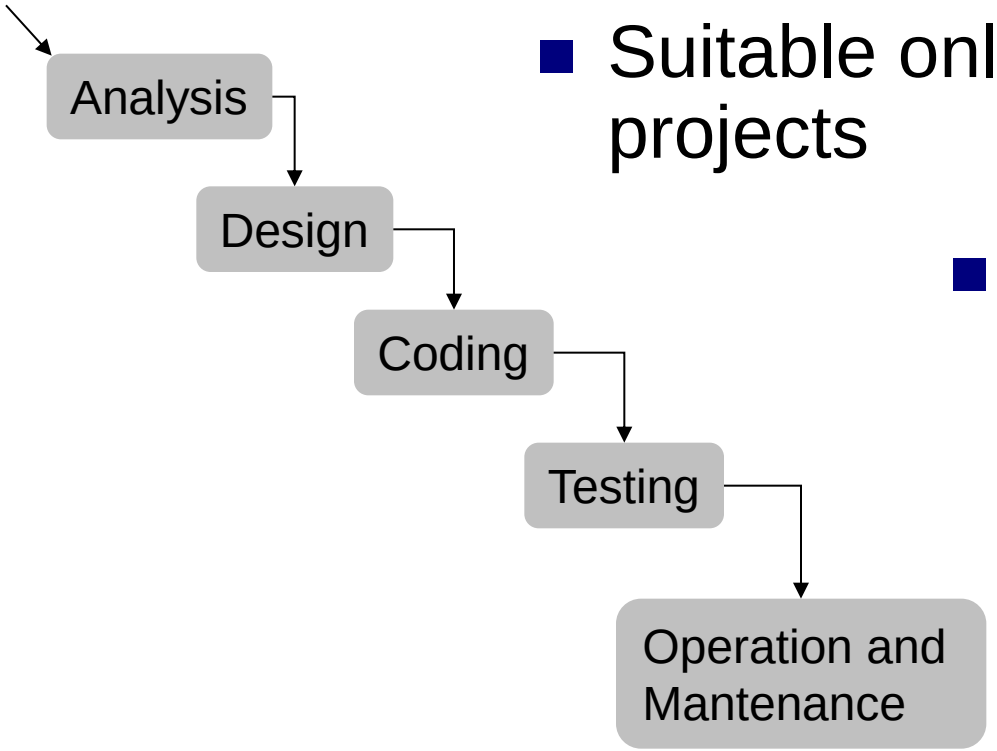
Management activities

- As projects in other disciplines, software development requires a series of management activities
  - **Estimate**: prediction of duration, effort and costs to carry out the project
  - **Planning**: selection of a strategy, define activities, assign a calendar and resources, coordination, etc.
  - **Negotiation**
  - Project tracking/**monitoring**
  - **Management**
  - **Coordination** of the team
  - **Technical leadership**
  - …

Life-cycle models

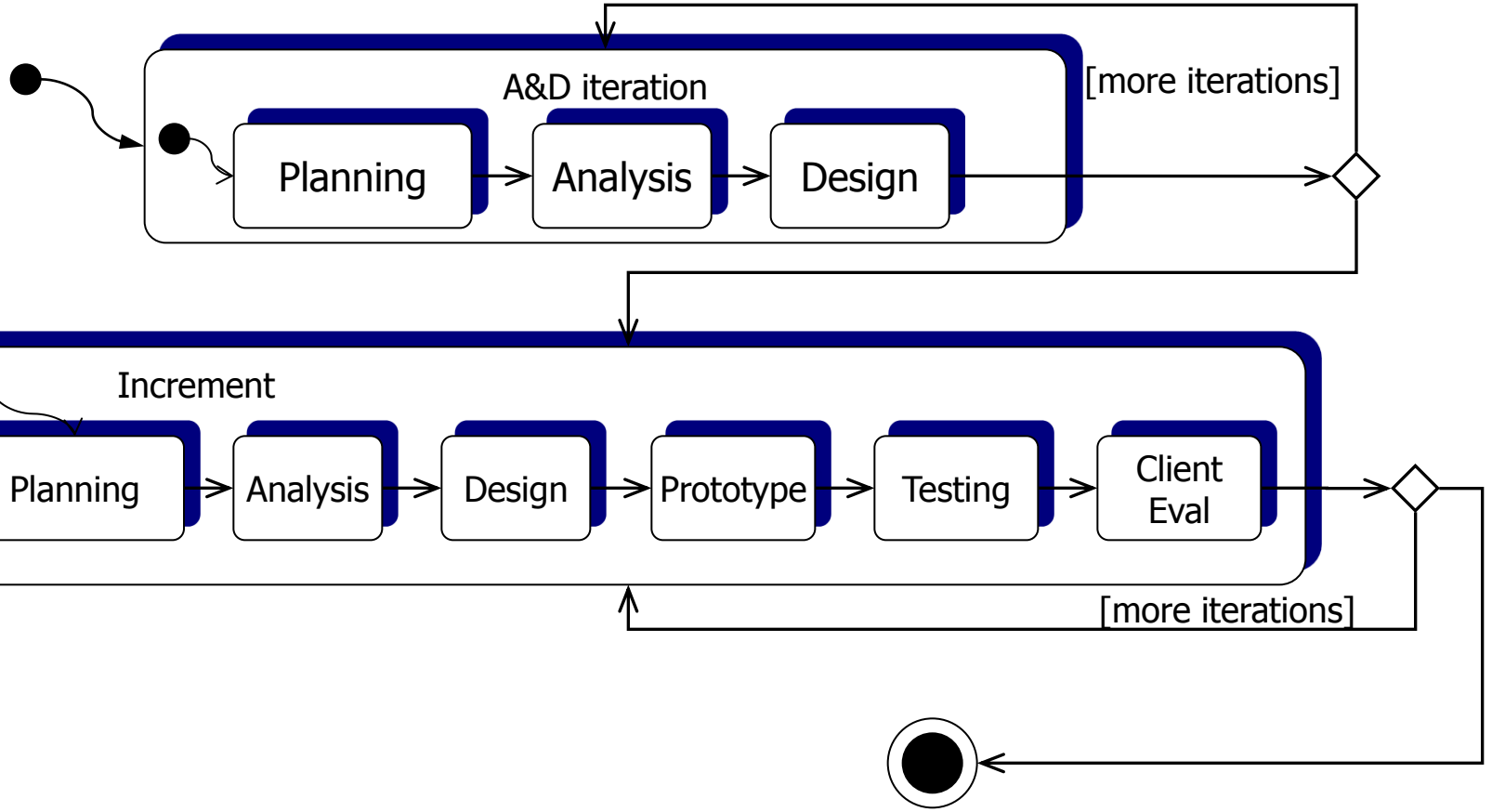
- Except in very simple cases, software construction does not follow a linear phase distribution
- Sometimes it is more convenient to perform iterations and increments
- ***Life-cycle model***. Scheme describing
  - Phases the project goes through
  - Transition criteria from a phase to the following one
  - Inputs and outputs of every phase

Waterfall



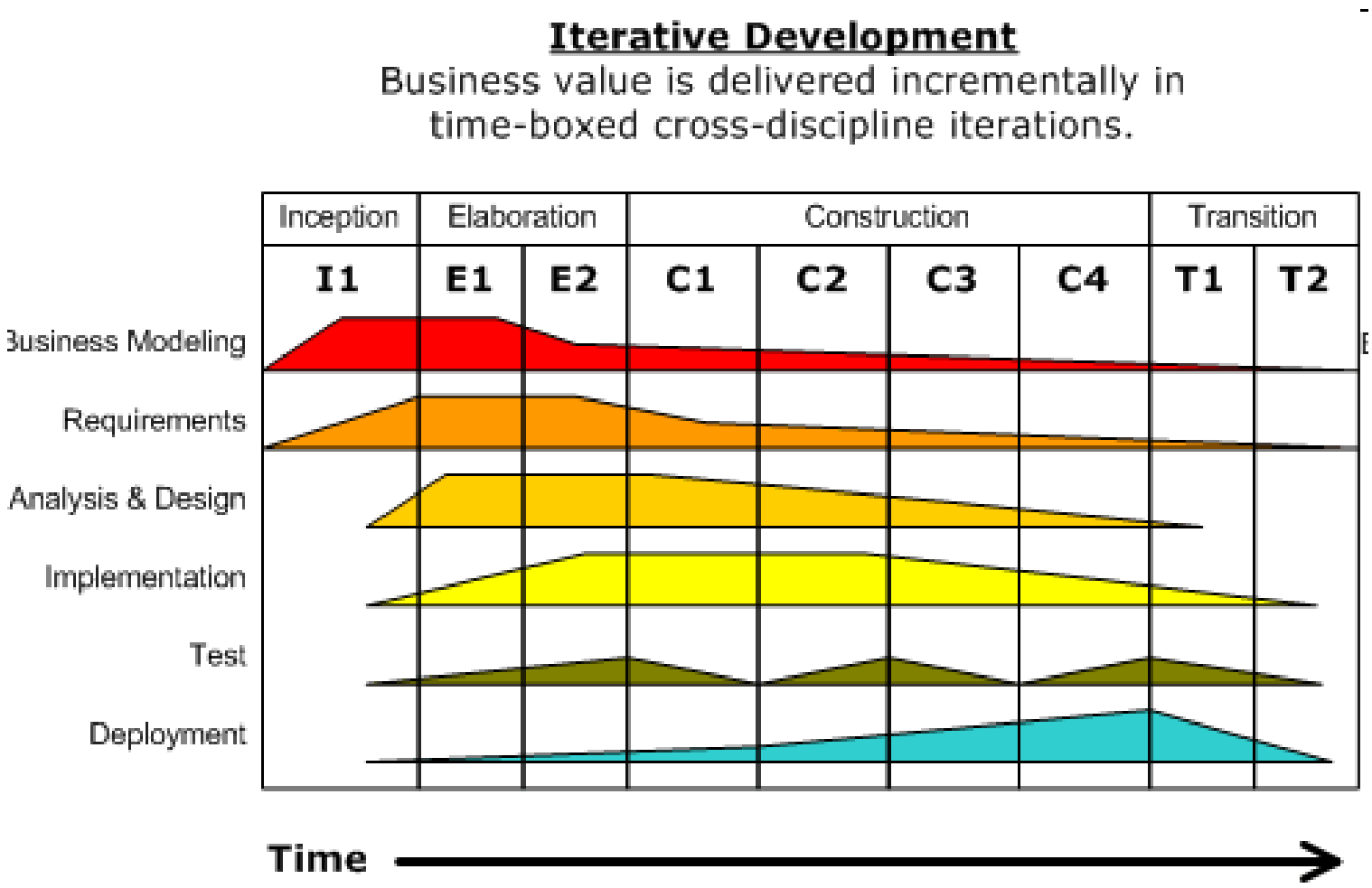
- Suitable only for very simple projects
- Disadvantages
  - Iterations are not allowed
  - Requirements are frozen at the beginning of the project
  - The application is not shown to the user until the end of the project

Incremental models



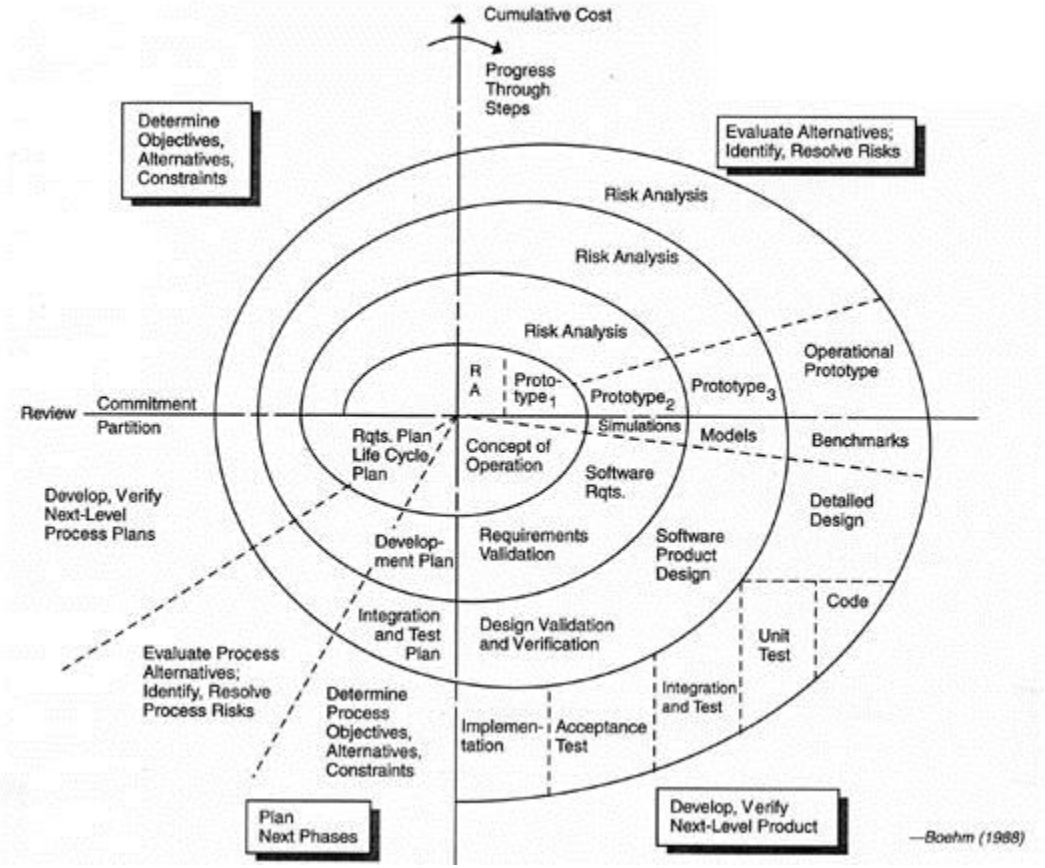
- Rational Unified Process
- Other “agile methodologies”
  - Extreme Programming (Xp)
  - Scrum

Incremental models

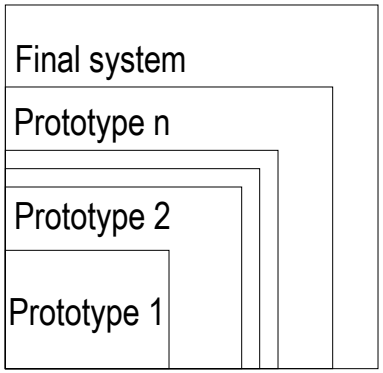


Other models

- Spiral life-cycle model



- Product development models
  - Prototyping



Methodologies

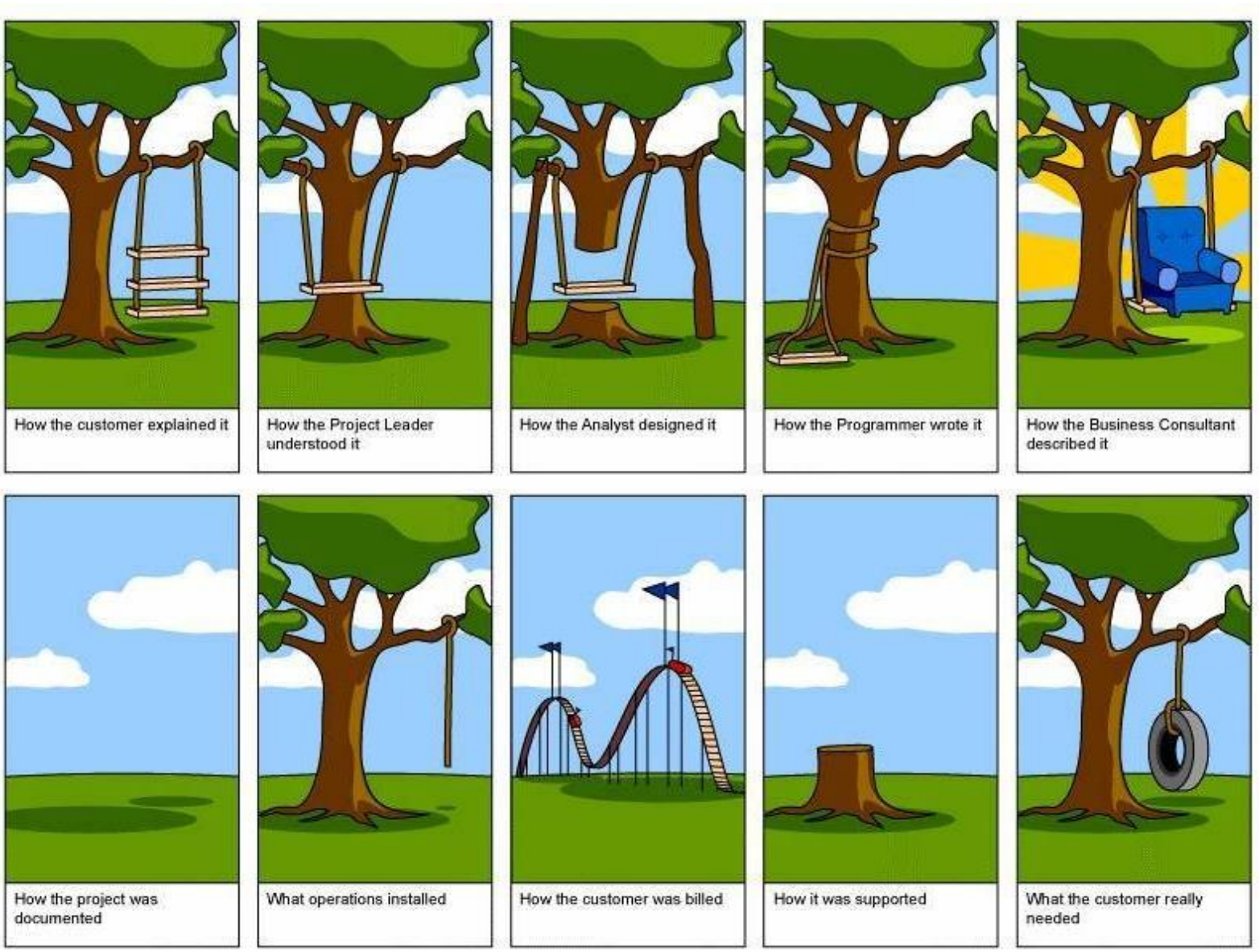
- In addition to phases, they indicate exactly which methods we need to use in each one of them

- Examples
  - METRICA (https://administracionelectronica.gob.es/pae_Home/pae_Documentacion/pae_Metodolog/pae_Metrica_v3.html)
    - For companies working for the Spanish asministration

Summary

- Developing software is not only about programming
- Other activities
  - Feasibility, Requirements, Analysis, Design, Programming, Testing and Maintenance
- Organization of the software development
  - Life-cycle models
  - Methodologies

In real life…


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

Bibliography

- Basic bibliography
  - Software engineering a practitioner's approach, 7ªed. Roger Pressman. McGraw Hill Higher Education, 2010. INF/681.3.06/PRE.
  - Software engineering, 9ª ed. Addison Wesley. Ian Sommerville. INF/681.3.06/SOM.
- Recommended lectures
  - Mary Shaw. "*Prospects for an engineering discipline of software*". IEEE Software, Vol.7(6), Nov 1990, pag. 486-495.
  - Mary Shaw. "*Continuing Prospects for an Engineering Discipline of Software*". IEEE Software. Vol. 26(6). Nov 2009. pag. 64-67.
  - IEEE Standard 1074-2006. IEEE Standard Software Life-Cycle Processes. 2006.