Resumo para o Teste 1 - RAS

- Chapter 2 - Software Engineering
    - **SWEBOK Knowledge areas:**
    - **Software and System**
    - **Domains**
    - **Development Process Models**
- Chapter 3 - Requirements
    - **Funcional Requirements**
    - **Non-funcional Requirements**
    - **User and System Requirements**
- Chapter 4 - Requirements Engineering
    - **Activities**
- Chapter 7 - Writing in a Natural Language
    - **Standardized format for user requirements**
    - **Standardized format for system requirements**
    - **Standardized format for non-functional requirements**
- Chapter 5 - Elicitation
    - **Identification of Entities**
        - **Identification of the Stakeholders**
        - **Identification of the Users**
        - **Identification of the Clients**
        - **Identification of the Costumers**
    - **Techniques for requirements elicitation**
- Chapter 8 - Modelling
- 1 - Introduction to software architecture
    - **Major Concepts**
    - **Good Architecture**
- 2 - Architectural Risks
    - **Risk-driven approach**
    - **Styles of Design**
- 3 - Design Tactics
- 4 - Design Patterns
- 5 - Architectural Styles
    - **Constraints**
    - **Patterns vs. Styles**
- Coisas que saíram no teste
    - Diagrama de Atividades
    - Correção de Requisitos
    - Diagramas UML
    - Class Models
    - Design Patterns

- Extras do livro
    - Sequence Diagram
    - State Machines
    - Use Case Diagram
    - Domain Model

Chapter 2 - Software Engineering

Software engineering: application of a systematic, disciplined and quantifiable approach in the context of the planning, development and exploration of software systems.

SWEBOK Knowledge areas:

1. Software requirements
2. Software design
3. Software construction
4. Software testing
5. Software maintenance
6. Software configuration management
7. Software engineering management
8. Software engineering process
9. Software engineering models and methods
10. Software quality
11. Software engineering professional practice
12. Software engineering economics
13. Computing foundations
14. Mathematics foundations
15. Engineering foundations

Software and System

Software is:

- Intangible
- Cost is determined by the cost of human resources
- Does nor wear out (does not lose qualities over time)

A system is an identifiable and coherent set of components that cohesively interact to achieve a given objective.

Classification of software systems:

Name	Description
Tailor-made/Custom	Made by request of a given client for satisfying their own necessities and expectations.
Software/Mass-market Product	Produced to be commercialized/made for the public in general.
Embedded Software	Unique device (satellite) or appliances (printers) that contain software.

A software product is composed of:

- programs
- data structures
- documentation

Type of Software Product	Description
System Software	Manages the hardware response of the computer.
Software Application	Performs tasks that are useful for the end users.

Domains

A domain is an area of human knowledge or activity that is characterised by possessing a set of concepts and terms that the respective players know. For example: telecommunications, transports, banking.

A *problem domain* is the context where one feels the necessities that need to be satisfied by the system to be developed.

A *solution domain* refers to the activities that are executed and the artefacts that are handled and constructed to solve the problem.

Classification of software systems domains:

- difficulty of the problem class (how the real problem can be framed into the conceptual problem that covers it)
- temporal relationship between data and processing (availability of input data and processing - static or dynamic)
- number of tasks to be executed simultaneously (sequential or parallel)
- relative difficulty of the data, interaction, and control aspects of the problem (whether the application is data-centered or interactive-centered)
- determinism level (whether the system is deterministic, like a simple calculator, or non-deterministic, like a chess game)

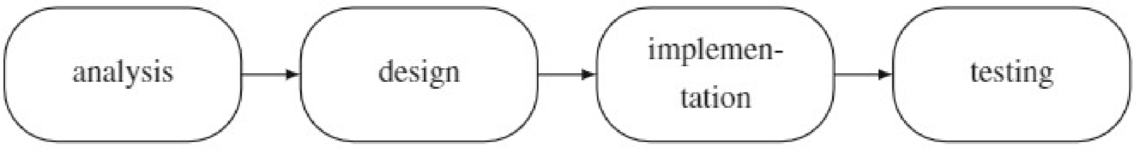
Development Process Models

A process model represents a development process and indicates the way it must be organised. It aims to help engineers in establishing the relationship between the activities and the techniques that are part of the development process.

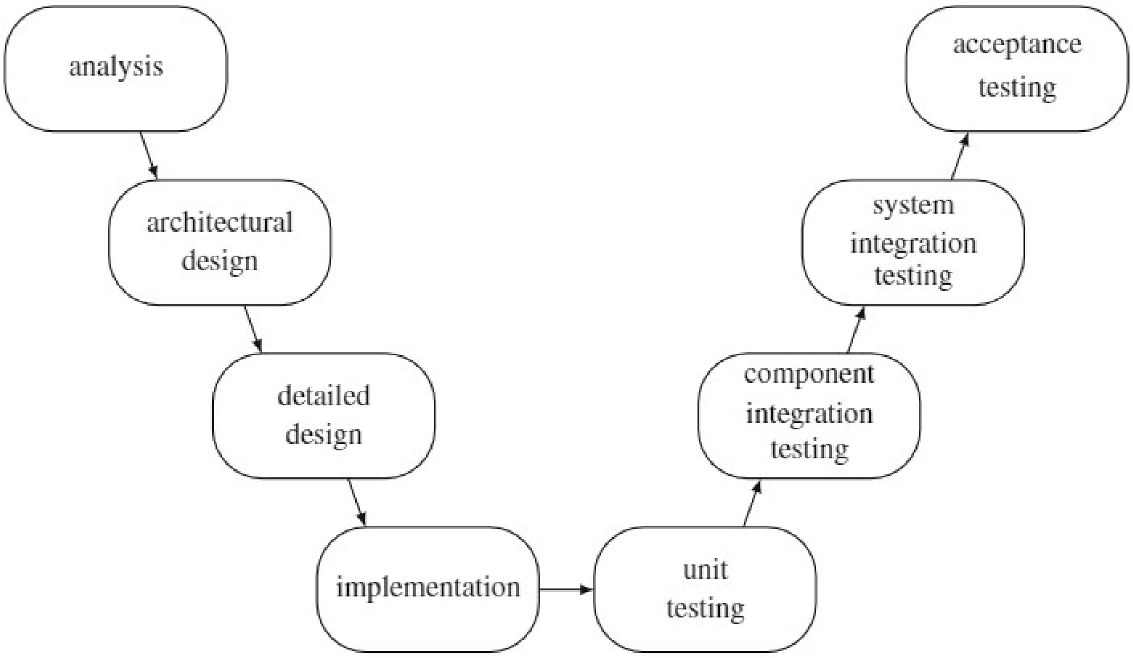
Objectives of development process systematisation:

- Clearly identify the activities that must be followed to develop a system
- Introduce consistency in the development process, ensuring that the systems are developed according to the same methodological principles
- Provide control points to evaluate the obtained results and to verify the observance of the deadlines and the resources needs
- Stimulate a bigger reuse of components, during the design and implementation phases, to increase the productivity if the development teams

Waterfall Process Model

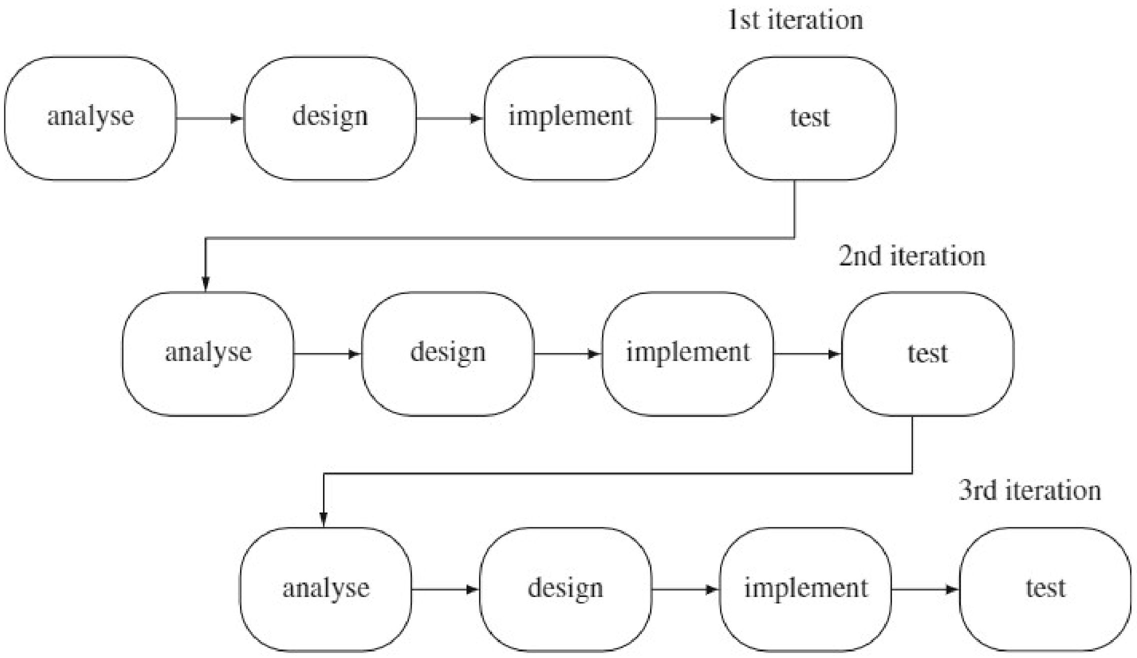


V Process Model

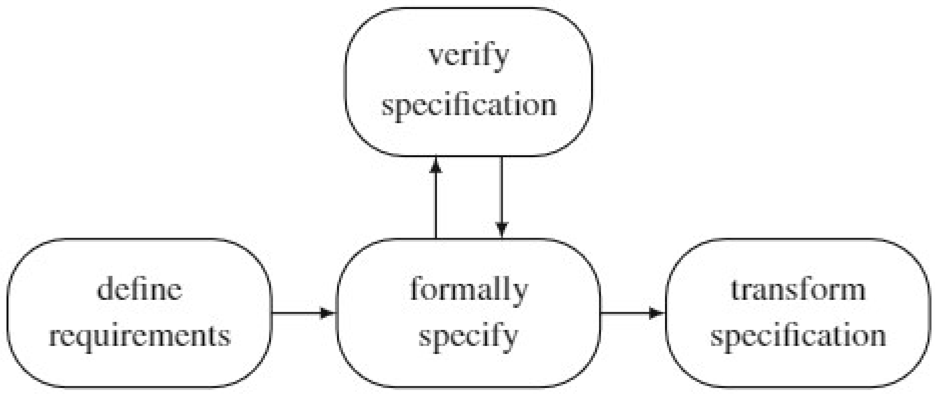


Incremental and Iterative Model

In each iteration the complexity of the functionalities implemented increases (manipulate files -> search and replace text -> figures and tables -> use of the thesaurus and auto spellers)

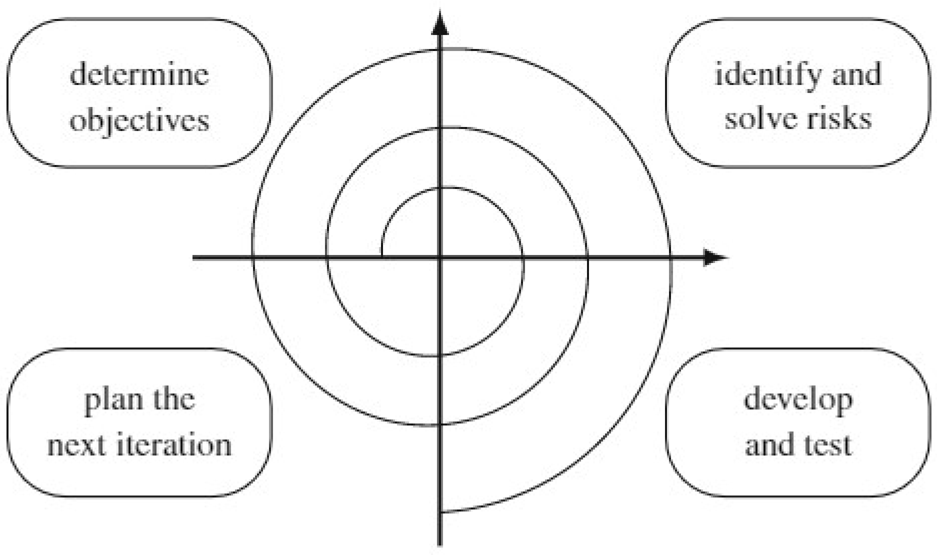


Transformational Model



Spiral Model

Minimises the number os *risks*: a potentially adverse circumstance that can have negative or pervese effects in the development process and in the final quality of the system.



Chapter 3 - Requirements

Requirement: express the users' necessities and restrictions that are placed on the system, and are seen as properties that the system must possess when built.

- Condition/capacity that someone needs to solve a problem or to achieve an objective
- Condition/capacity that must be verified or possessed by a system or by a system component to satisfy a contract
- Documented representation of a condition/capacity like in (1) or (2)

Type of Requirement	Description
Primary	Originates directly from some stakeholder.
Derived	Obtained by refining a primary requirement.
Candidate	Was identified by some elicitation technique.

Funcional Requirements

- describes a functionality to be made available to the users of the system

- characterises partially the system behavious as an answer to the stimulus that it is subject to
- does not mention any technological issue
- ideally independent of design and implementation aspects

A *set* of functional requirements is **coherent** if there are no contradictions, and **complete** if it considers all the necessities that the client wishes to see satisfied.

Implicit requirements: not specifically asked for, but that must be included according to the domain knowledge of the development team.

Explicit requirements: requirement requested by clients.

Non-funcional Requirements

- set of restrictions imposed on the system to be developed
- establishes how attractive, fast, or reliable the system is
- includes time constraints, restrictions in the development process, or adoption of standards
- alternativaly called a quality requirement
- cannot be modularised
- cannot be fulfilled in an isolated way

Emergent property: is a system property that can be associated with the system as a whole, but not individually to each of its components (for example: a bike can transport people, but each of its components alone cannot).

- *Product requirements:* characterise aspects of the behaviour of the system itself (reliability, efficiency, etc).
- *Organisational requirements:* come from strategies and procedures estabilished in the context of the manufacturing process of the system, or the client organisation.
- *External requirements:* have origin in external factors to the system/development process (legal, ethical, etc).

Classification of non-functional requirements:

1. Appearance
2. Usability (ease of use, personalisation, ease of learning, accessibility)
3. Performance
4. Operational
5. Maintenance and Support (preventive, corrective, perfective, adaptive)
6. Security
7. Cultural and Political
8. Legal

User and System Requirements

User requirement: functionality that the system is expected to provide to its users, or a restriction that is applicable to the operation of that system. They are expressed with the problem domain terminology, using natural language and informal diagrams, and not with the technical domain terminology.

System Requirements: more detailed specification of a requirement, being generally a formal model of the system. They are orientes towards the solution domain, helping in system design and construction, and documented in a more technical language than user requirements.

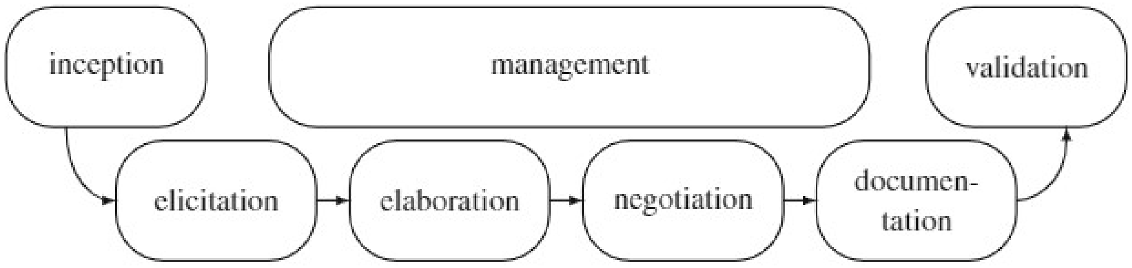
Chapter 4 - Requirements Engineering

Requirements Engineering: all the activities related to requirements discovery, negotiation, documentation, and maintenance in engineering projects. Must be kept strictly separated from their own solutions.

It seeks to ensure that:

- all the relevant requirements are explicitly known and comprehended at the intended level of detail
- a reasonable and wde agreement about the requirements is obtained among the stakeholders
- all the requirements are duly documented, in conformity with the established formats and templates

Activities



Chapter 7 - Writing in a Natural Language

Standardized format for user requirements

1. A subject that indicates the type of **users** that benefit from that requirements
2. An intended result to achieve with the requirement (using a predicate)
    1. Verb (functionality to be performed)
    2. Other elements to complete the predicare: objects, predicatives, adjuncts, etc.
3. A mechanism to allow a test for the requirement to be defined

Standardized format for system requirements

1. A subject, either the system under development or a design entity that is related to the requirement
2. An intended result to achieve with the requirement (using a predicate)
    1. verb (functionality to bring about)
    2. other elements to complete the predicate: objects, predicatives, adjuncts, etc.

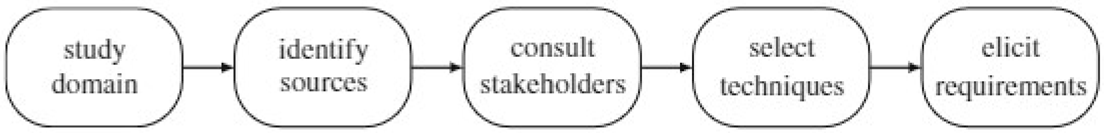
Standardized format for non-functional requirements

1. The system under development or a design entity that is related to the requirement
2. A quality to be achieved with the requirement
    1. verb ("to be", "to have", etc)

2. object (description)

Chapter 5 - Elicitation

**Requirements elicitation:* permits to understand what are the requirements of a given system. It allows comprehending the necessities and expectations that stakeholders have with respect to a given sysytem.



Identification of Entities

Identification of the Stakeholders

System Stakeholder: someone that can be materially affected by the implementation of that system.

A *negative stakeholder* is someone that desires that the system is not developed, and may be in peaceful opposition to active hostility towards the development of the system.

Identification of the Users

User: any person that operates and interacts directly with the system, whenever it is in effective operation in its environment.

Identification of the Clients

Client: the entity that orders and pays for the development of a system.

Identification of the Costumers

Costumer: someone who pays for aquiring a system, whenever available. Consumer.

A group of customers represents a distinct *market segment* if:

- their necessities demand and justify a different offer
- they are approached through different distribution channels
- they need different types of relations
- they have significantly distinct levels of profitability
- they are willing to pay for different aspects of the offer

Techniques for requirements elicitation

Competencies:

- Questioning
- Observing
- Discussing
- Negotiating
- Supposing

Groups of Techniques:

- Marketing
- Psychology and sociology

- Participative design
- Human factors and human-machine interaction
- Quality
- Formal methods

Techniques according to the group of persons:

- Individuals:
  - Interviews
  - Surveys
  - Introspection
- Groups:
  - Group dynamics
- Artefacts:
  - Domain analysis
  - Prototyping
  - Persona (fictitious person that represents the target user)

Chapter 8 - Modelling

Modelling: is the process of identifying adequate concepts and selecting adequate abstractions to construct a model taht reflects a given universe of discourse appropriately.

Characteristics:

- Abstraction
- Understandability
- Accuracy
- Reasoning
- Inexpensiveness

Dimensions:

- Form:
  - Physical - small cardboard model of a house
  - Symbolic - equations, topological maps, topographical maps
- Representativeness:
  - Descriptive - weather forecast
  - Prescriptive - target system does not yet exist
- Perspective:
  - Strictural - UML diagrams
  - Behavioural - finite state machines

Modelling language by what they are oriented towards:

- State: set of states and set of transitions
- Activity: set of activities related by data or execution dependencies
- Structure: representation of the physical models or components of a system
- Data: system is represented as a collection of entities related by attributes, properties, and classes

- Heterogenous: use of several characteristics

Models for requirements (UML):

- Domain
- Use Case
- Interaction
- Class
- State
- Activity

1 - Introduction to software architecture

Major Concepts

Principle: a comprehensive and fundamental law, doctrine, or assumption:

- abstraction
- coupling and cohesion
- decomposition and modularization
- encapsulation and information hiding
- separation of interface and implementation
- sufficency, completeness and primiriveness
- separation of concerns

Software design: the conception or invention of a scheme for turning a specification for computer software into operational software; the activity that links requirements to coding and testing.

Software Architecture: set of structures needed to reason about the system, which comprise of software elements, relations among them, and properties of both.

- *Static structures:* define a system's internal design-time elements and their arrangement.
- *Dynamic structures:* define its run-time elements and their interactions.
- *Externally visible behaviour:* defined the functional interactions between the system and its environment.
- *Quality property:* is a non-functional property.

Functional requirements don't determine architecture.

Architectures are influenced by:

- the stakeholders
- the developing organization
- the background and experience of the architects
- the technical environment

Design

Architecture design: the macroscopic parts of the design, such as modules and how they are connected.

Detailed design: covers everything else.

The output of design is a set of artefacts that record the decisions that were taken; the rationale of each non-trivial decision is explained.

Dependencies

Eeach component plays a significant role in the system, and together the components provide the required functionality.

Minimizing dependencies between components is important to create loosely couple architecture, so that changes are localized and don't propagate through the system.

Good Architecture

Skeletons

A 3 tier software architecture (e.g. the MVC system) enables IT systems to localize change and handle transactional loads. A cooperating-processes architecture is well suited to operating systems because it isolates faults.

Quality attributes:

Developers must pay attention to functionality, but a system also has quality attributes. A system's architecture enabled or inhibits qualities such as security or performance. Functional requirements that change are a challenge, but evolving quality attributes can force drastic changes (e.g. a system that supports 100 users may be impossible to scale up to 10000 without a drastic architectural change).

Architecture is orthogonal to functionality

A system's architecture can be changed, but yet keeping its functionality. The same architecture can be used on a system with different functionality. A poor architecture choice can make functionality and quality attributes difficult to achieve.

Architecture is likely to require more attention in large scale of high complexity systems.

For a good architecture:

- Should be a product of a small group of architects with an identified leader.
- The architect should base the architecture on a prioritized list of quality attributes.
- Should be documented using views, which should address the concerns of the most important stakeholders.
- The architecture should be evaluated for its ability to deliver the system's important quality attributes.
- The arhitecture should lend itself to incremental implementation.

Presumptive architecture: family of architectures that dominates a particular domain. They are a good match for the common risks in the domain.

Reference architecture: family of architectures that describes an architectural solution to a problem.

2 - Architectural Risks

Risk-driven approach

Failure is central to the design process, and it is by thinking in terms of avoiding failure that successful designs are achieved.

A risk-driven approach will help developers build quality software quickly and at low cost:

1. identify and prioritize risks
2. select and apply a set of techniques
3. evaluate risk reduction

Kinds of risks:

- Project management risks:
  - lead developer hit by bus
- Software engineering risks
  - the server may not scale to 800 users
- Prototypical risks:
  - IT domain: complex, poorly understood problem
  - Systems domain: performance, security
  - Web domain: security, scalability

Risk techniques:

- Software engineering:
  - design patterns
  - domain modeling
  - throughput modeling
  - security annalysis
  - prototyping
- Other engineering fields:
  - stress calculation
  - breaking point testing
  - therman analysis
  - reliability testing
  - prototyping

Styles of Design

Evolutionary design: the design system grows as the system is implemented.

Planned design: architecture is detailed before construction. Few people advocate doing planned design for an entire software system.

Minimal planned design: balanced style that joins evolutionary and planned design.

3 - Design Tactics

A tactic is a design decision that impacts on specific quality attributes.

Types of tactics:

- Availability
  - Aim to avoid faults from becoming failures or at least reduce the effects of the

    fault and make repair possible.
  - Involve redundancy, health monitoring, and recovery.
- Modifiability
- Performance
  - generate a response to an event arriving at the system within some time constraint
- Security
- Testability
- Usability

Technique	Examples
Fault Detection	Ping/echo Heartbeat Exceptions
Fault Recovery	Voting Active redundancy Passive redundancy
Fault Prevention	Removal from service Transactions Process monitor

4 - Design Patterns

Design principles:

- Identify the aspects of your application that vary and separate them from what stays the same.
- Program to an interace, not the implementation.
- Open-closed principle: a class/module should be open for extension, but closed for modification.

A pattern is a textual description of a generic solution for a recurring problem in a given context.

Types of design patterns:

- Creational
- Structural - strategy
- Behavioural - observer

The *strategy pattern* defines a family of algorithm, encapsulates each one, and makes then interchangeable. It lets the algorithm vary independently of the client that uses it. Duck example.

The *observer pattern* defines a one-to-many dependency between objects. Whenever one object changes state, all its dependents are notified and updated automatically. Animals example, lensing.

The *decorator pattern* attaches additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality. Beverages example (matrioska).

5 - Architectural Styles

An **architectural style** consists of:

- a set of components that perform some function at runtime (process, procedure)
- a topological layout of the components showing their runtime relationships
- a set of semantic constraints
- a set of connectors that mediate communication among components (sockets, streams)

Constraints

Constraints act as guide rails that point a system where you want it to go. The consistency brough about by the constraints of the style can encourage clean evolution of the system, which can make maintenance easier.

Platonic style	Embodied style
Idealization.	Exists in real systems.
In books but rarely in code.	Often violates the strict constraints found in platonic styles.

Example: the platonic client-server requires that servers be unaware of clients. However embodied versions of the style can have server occasionally push data to clients.

Patterns vs. Styles

Design patterns are at a smaller scale than architectural styles. Patterns can appear anywhere in your deisgn, and multiple patterns could appear in the same design.

However, *a system usually has a single dominant architectural style*.

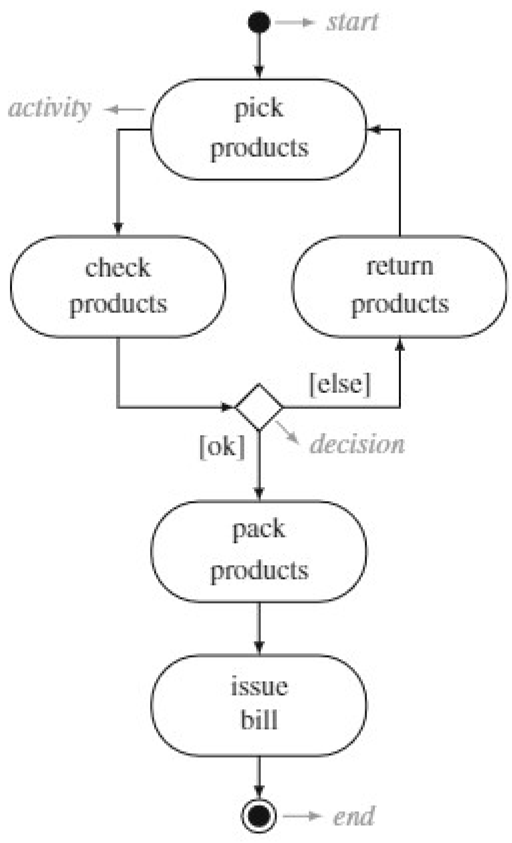
Types of styles:

- Layered - layers
- Big ball of mud - spaghetti
- pipe-and-filter - continually processing and incrementing data
- batch-sequential - like pipe-and-filter but with stages and between stages data is tipically written to file, not to next stage's output
- Model-centered (shared data) - independent components interact with a central model (data store)
- Publish-Subscribe - independent components publish events and subscribe to them
- Client-Server & N-Tier - hierarchical. Clients synchronously request services from servers, clients initiate the communication (not the server)
- Peer-to-Peer - egalitarian. Nodes communicate with eachoter as peers
- Map-Reduce - good for large datasets, enables computation to be spread across multiple computers.
- Mirrored, Farm & Rack - not runtime viewtype like the others, but from the allocation viewtype.

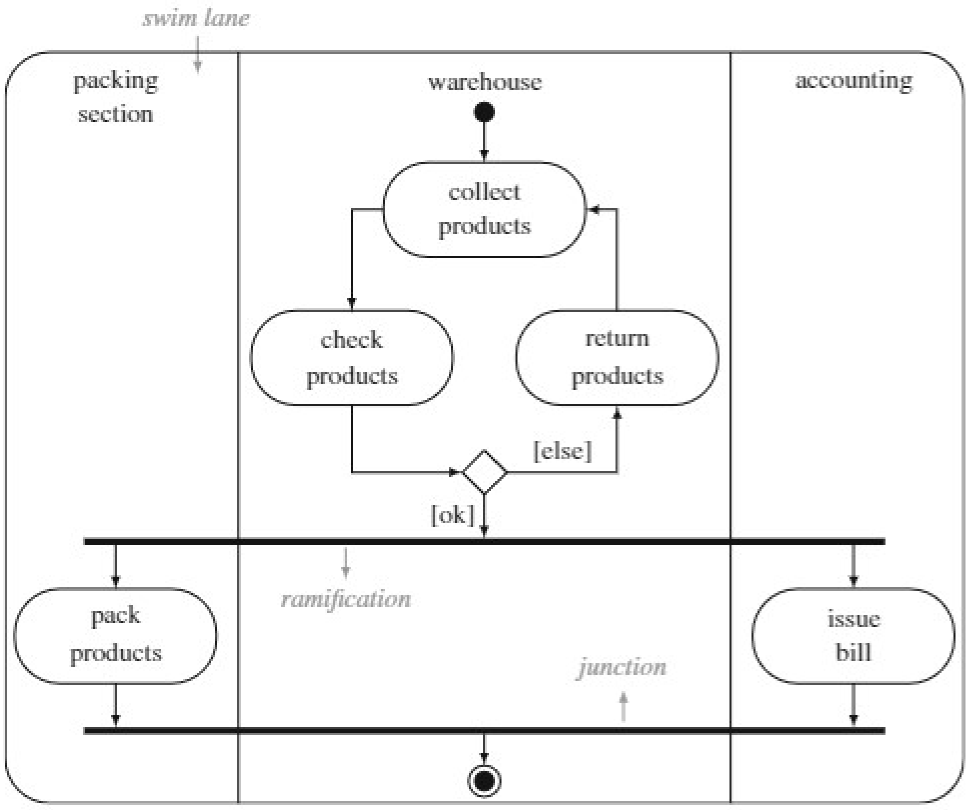
Coisas que saíram no teste

Diagrama de Atividades

Exemplo simplificado:



Exemplo completo (*swim lanes*):



No exemplo completo, cada *swim lane* (partição) identifica o departamento que realiza a ação. No exemplo complexo, duas ações podem ser realizadas em paralelo uma vez que são entidades diferentes a fazê-las.

Resolução do exercício do teste:

- tostar o pão
- se fresco, não tostar
- pôr queijo ou fiambre
- aquecer o chá
- beber o chá
- comer o pão
- lavar a loiça

Correção de Requisitos

Formato dos requisitos:

Tipo	Formato	Exemplo
User	Subject + Result (verb + object) + Mechanism for test	
System	Subject + result (verb + object)	
Non-functional	System/Entity + Quality (verb + object)	The product shall be easy to use for illiterate persons.

Cheatsheet: **User:**

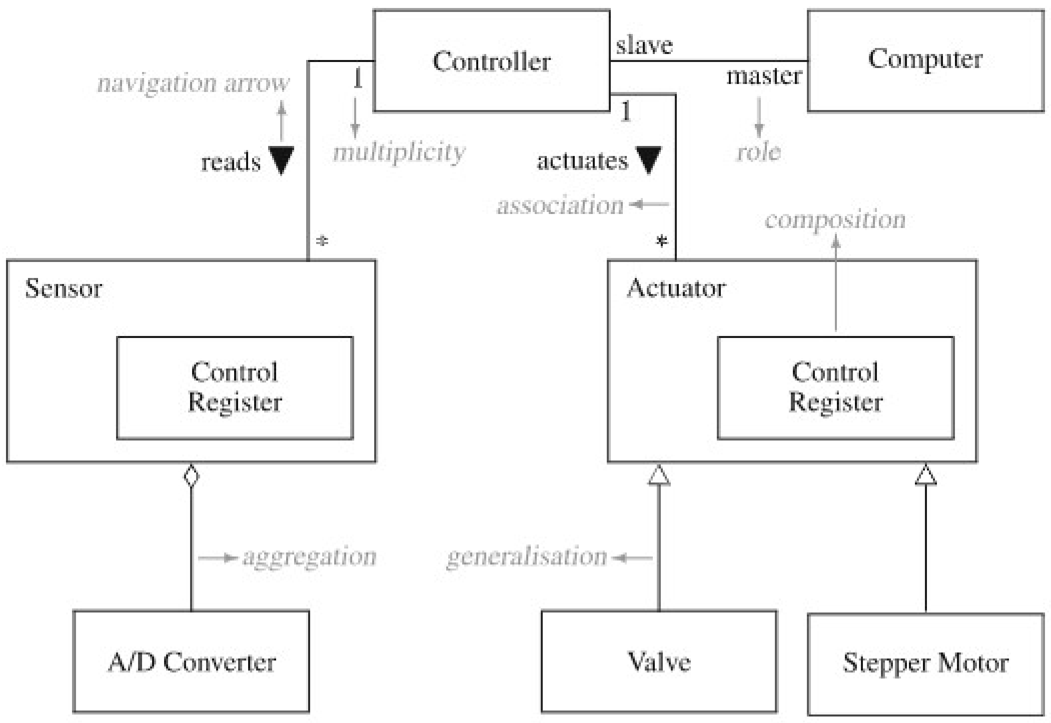
- functionality that the system is expected provide to users

- restriction that is applicable to the operation of that system

System:

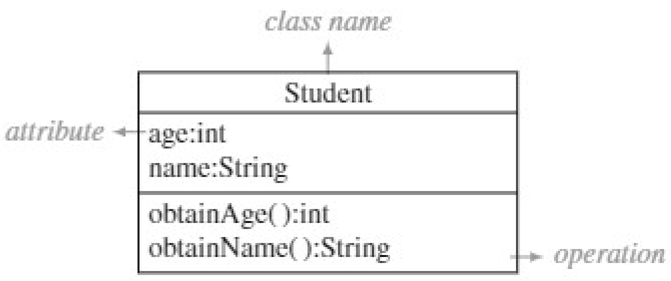
- detailed specification of the requirement
- oriented towards the solution domain
- more technical
- independent from design

Diagramas UML



página 194

Class Models



página 192

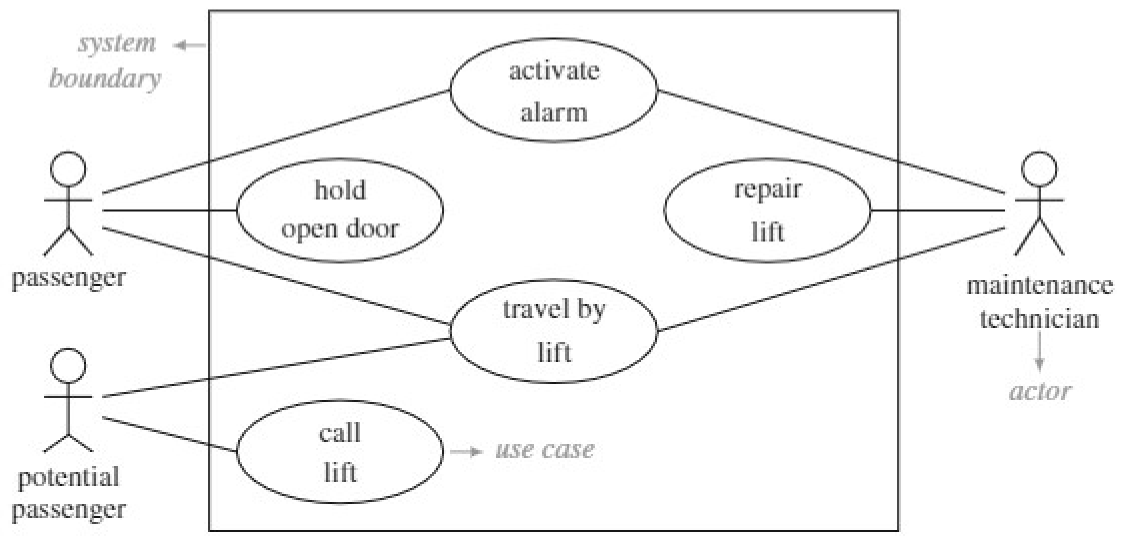
Design Patterns

Extras do livro

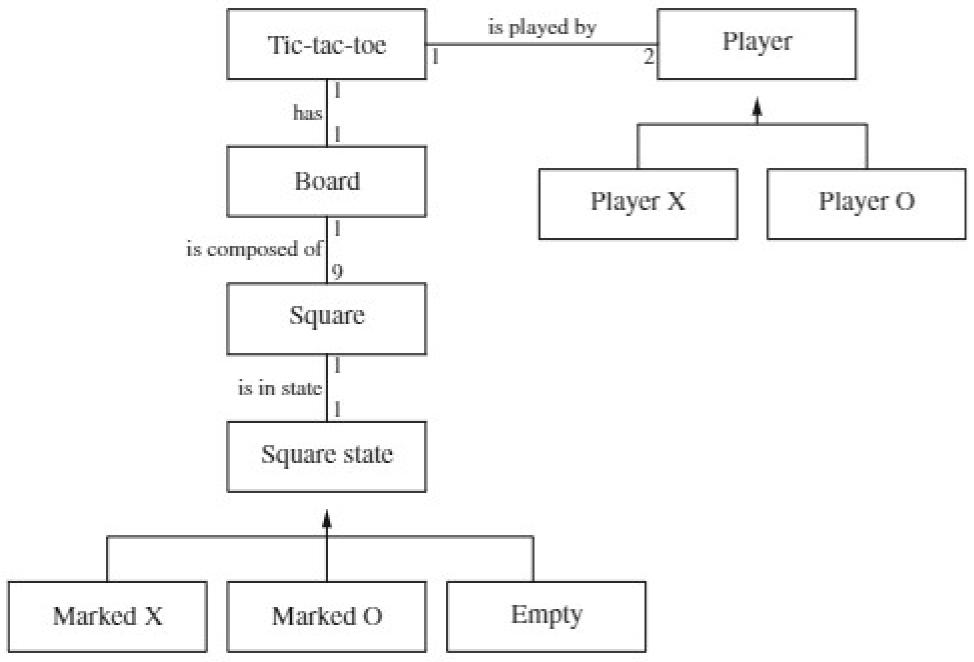
Sequence Diagram

State Machines

Use Case Diagram



Domain Model


Tic-tac-toe
is played by
1
2
Player
has
1
1
Board
is composed of
1
9
Square
is in state
1
1
Square state
Player X
Player O
Marked X
Marked O
Empty