0.1 Basic Concepts

Software Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

0.1.1 Design Level

- High-level design or **architectural design**: partition the system into components.

- Detailed design (e.g., object-oriented design): partition each component into classes.

- Design of algorithms and data structures.

0.1.2 Typical outputs

"Requirements Capture - "Architectural Design (high-level design)" - "Detailed Design" | "Coding" | "Unit Testing" - "Integration and Testing"

0.1.3 Architectural design notations

- Block diagrams
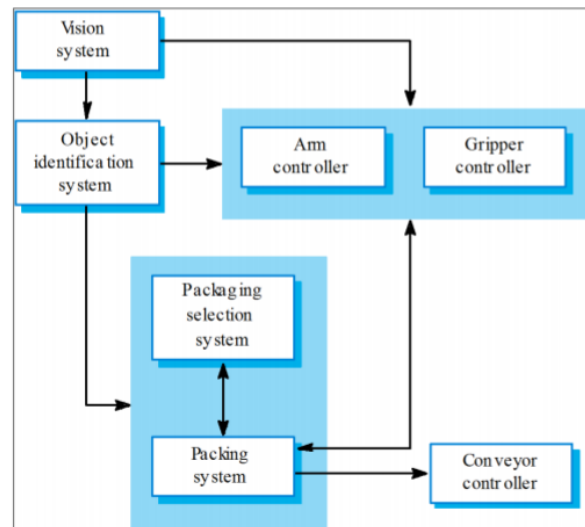
  - Informal, but simple and easy to understand.
  - The most frequently used for documenting software architectures.
  - Lacks semantics and detail.

- Architecture modeling languages (with UML)

– Semi-formal

– Multiple views

- Formal architecture description languages (ADLs)

  – Support automated analysis and simulation

0.1.4   Non-functional requirements

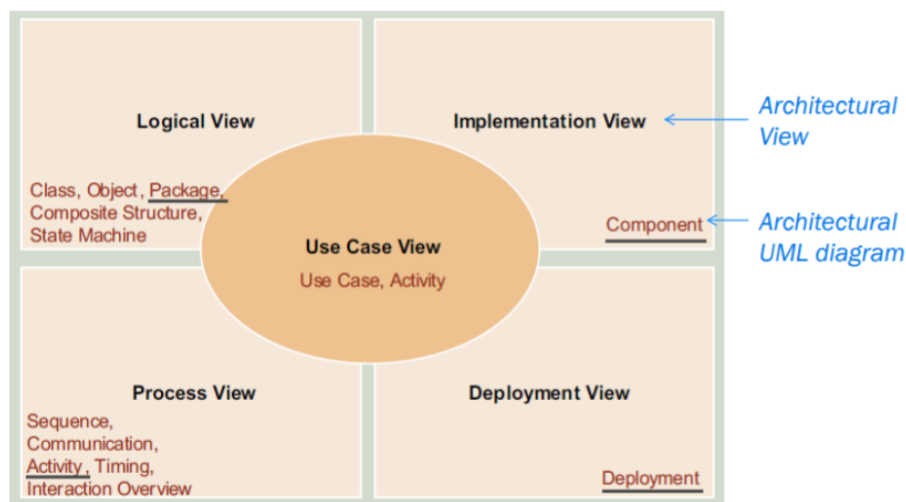Architectural design decision are strongly influenced by non-functional requirements:

- Performance - Localize critical operations, minimize communications and levels of indirection

- Security - Use a layered architecture with critical assets in inner layers

- Safety - Localize safety-critical features in a small number of subsystems

- Availability - Include redundant components and mechanisms for fault tolerance

- Portability - Isolate platform dependencies in specific components

- Maintainability - Use fine-grained, loosely coupled, replaceable components.

0.2   Architectural Views

0.2.1   Civil engineering analogy

Architecture is best described by considering multiple views

0.2.2   4+1 view model of software architecture

Locial View: Package diagrams (and others)	Implementation View: Component diagrams
Shows logical packages* and their relationships	Shows software components and dependencies among them
*division of responsibilities, independently of allocation to sw components or hw nodes	
Process View: Activity diagrams (and others)	Deployment View: Deployment diagrams
Shows processing steps, data/object stores, data/object-flows, and opportunities for parallelization	Shows hardware nodes, communication relationships and software artifacts deployed on them

Use Case View (+1): Relates the other views.

0.3 Component Diagrams
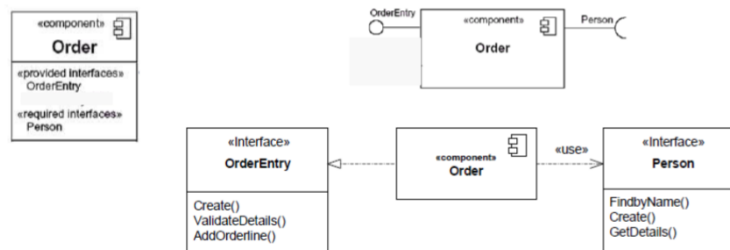
- Components
  A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.
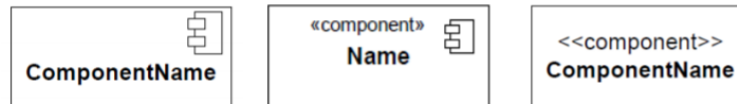

- Interfaces
  A component defines its behavior in terms of **Interfaces provided (realized)** and **Interfaces required (used)**
  Components with the same interfaces are interchangeable.

3

- Dependencies
  To promote intercangeability, components should not depend directly on other components but rather on interfaces (that are implemented by other components)

(simplified view, omitting interfaces)

assembly connector, shorthand for:
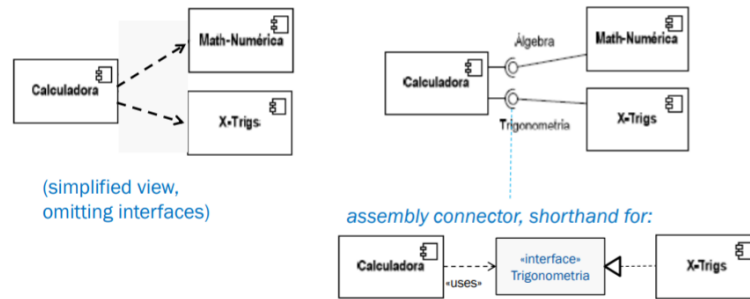
- Components and classes
  The behavior of a component is usually realized by its internal classes
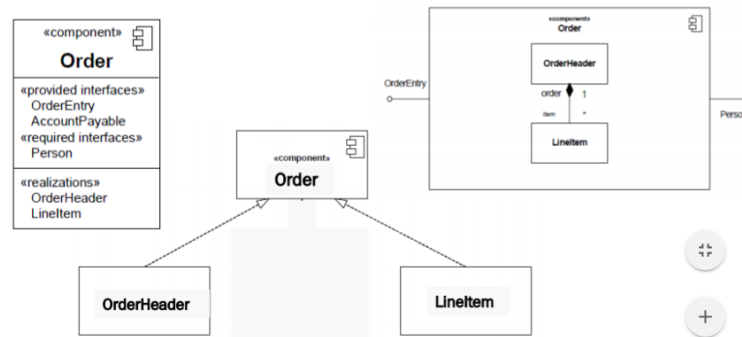
- Components and artifacts
  Components manifest physically as artifacts (that may be deployed in hardware nodes)
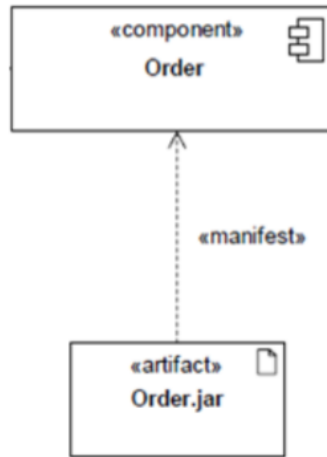
0.4 Deployment Diagrams

- **Nodes**
  Nodes are computational resources where artifacts may be deployed

- **Artifacts**
  Artifacts are physical information elements used or procedure by a software development process. (example: model files, source code files, executable files, scripts, etc).

0.5 Package Diagrams

0.6 Activity Diagrams

Compiler Architecture

6

0.7 Architecural Patterns

- Patterns are a means commonly used in software engineering of representing, sharing and reusing knowledge.
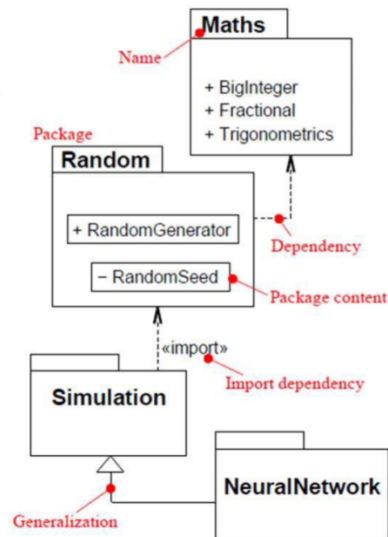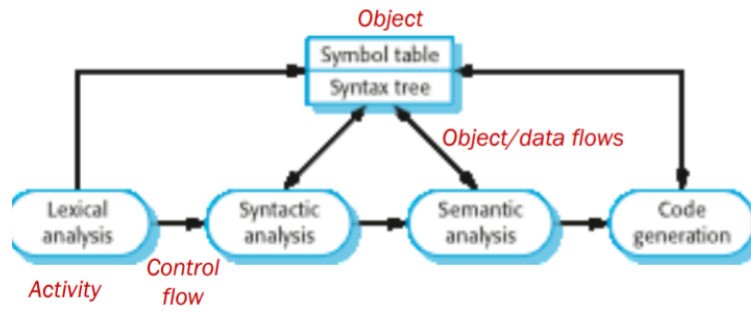
- A pattern describes a proven solution for a recurring problem in a context "Pattern = (Context, Problem, Solution)"

- An architectural pattern (or architectural style) is a stylized description of good architectural design practice, which has been tried and tested in different environments.

0.7.1 Mode-view-controller (MCV)

For interactive processing Separates presentation **(V)** and interaction **(C)** from the application data/sate **(M)**.
Example: Ruby on Rails.

When used:

- There are multiple ways to view and interact with data.

- Future requirements for interaction and data presentation are unknown.

Advantages:

- Allows the data and its representation to change independently.

- Supports presentation of the same data in different ways.

- Changes made in one data representation are shown in all of them.

Disavantages:

- Code overhead for simple data model and interactions.

0.7.2 Pipes and filters (data flow)

- For batch processing

Organizes the system as a set of data processing components (filters), connected so that data flows between components for processing (as in a pipe).
Example: Compiler Architecture, Test Generation Tool.

When used:

- In data processing applications (both batch- and transaction based) where inputs are processed in separate stages to generate outputs.

Advantages:

- Easy to understand and supports transformation reuse.

- Workflow style matches the structure of many business processes.

- Evolution by adding transformations is straightforward.

- Can be implemented as either a sequential or concurrent system.

Disavantages:

- Format for data transfer has to be agreed upon.

- Possible overhead in parsing/unparsing input/output data.

- Not really suitable for interactive systems

0.7.3 Layered architecture

For complex systems with functionalities at different levels of abstraction

Organizes the system into a set of layers, each of which groups related functionality and provides services to the layer above. (Strict, Relaxed)
Example: Three-Layered Services Application, CASE Tool.

When used:

- When building new facilities on top of existing systems.

Advantages:

- Supports the incremental development layer by layer.

- Lower layers provide isolation from system/platform specificities.

Disavantages (strict layering):

- Providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers.

- Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

0.7.4 Repositories (data centric)

For accessing AND manipulating shared data by multiple subsystems

All data in a system is managed in a central repository that is accessible to all system components or subsystems. Components or subsystems do not interact directly, only through the repository. (Variants: passive, active)
Example: IDE.

When used:

- In systems in which large volumes of information are generated that have to be shared and/or stored for a long time.

- In data-driven systems where the inclusion of data in the repository triggers an action or tool (active repository).

Advantages:

- All data can be managed consistently as it is all in one place.

- Components can be independent (don't need to know each other).

- Changes made by one component can be propagated to all others.

Disavantages:

- The repository is a single point of failure for the whole system.

- Possible inefficiency in having all communication through the repos.

- Distributing the repos. across several computers may be difficult.

0.7.5 Client-server and n-tier systems

For accessing shared data and resources from multiple locations

Asymmetrical distributed system in which clients request services from servers through a shared network or middleware.
N-tier systems are a generalization of client-server (2-tier) systems, in which servers may in turn act as clients. The tiers may also be implemented on a single computer.
Example: Film Library, Four-Tiered Web Application.

When used:

- When shared databases or other resources have to be accessed from a range of locations.

- Because servers can be replicated, may also be used when the load on a system is variable.

Advantages:

- Servers can be distributed and replicated across a network.

- General functionality (e.g., printing) can be available to all clients.

Disavantages:

- Each service is a single point of failure so susceptible to denial of service attacks or server failure.

- Performance may be unpredictable because it depends on the network as well as the system.

- Possible management problems if servers are owned by different organizations.

0.7.6 Design models and design views

An object-oriented design model may in general cover 4 inter-related design views, represented by means of appropriate UML diagrams. **(External, Internal, Static, Dynamic)**

0.8 Process stages

- There are a variety of different object-oriented design processes that depend on the organization using the process.

- Common activities in these processes include:

  - Define the system context and use cases;
  - Design the system architecture;
  - Identify the principal object classes in the system;
  - Develop design models;
  - Specify object interfaces (API).

0.9 Common use cases of UML sequence diagram in detailed design

- show interactions between the system and its environment

- show internal interactions between objects the system

- show a dynamic view of the system.

0.10 Software Engineering Laws

0.11 Lei Nº3 - Principio fundamental da Arquitetura de Software

Qualquer problema de estruturaçao de software resolve-se introduzindo niveis de indireção.
Corolário: Qualquer problema de desempenho resolve-se removendo niveis de indireção.

0.12 Lei nº4 - Lei de Arquimedes da Arquitetura de Software

Um sistema de software fundado numa má arquitectura afundar-se-á sob o peso do seu próprio sucesso.