

# 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.

## 1.1 Design Level

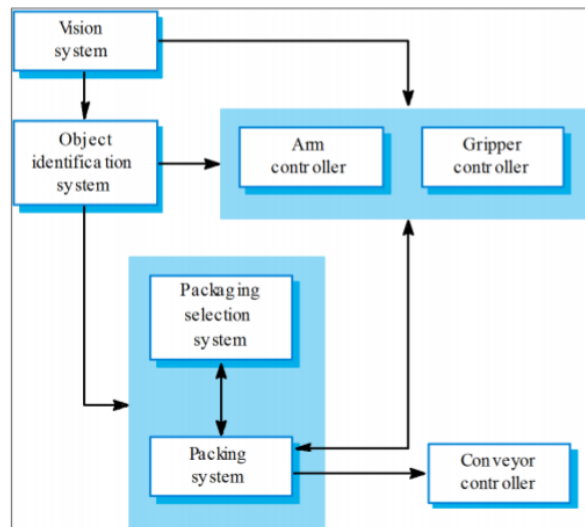
- High-level design or **architectural design**: partition the system into components.
- Detailed design (e.g., object-oriented design): partition each component into classes.
- Design of algorithms and data structures.

## 1.2 Typical outputs

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

## 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

## 1.4 Non-functional requirements

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

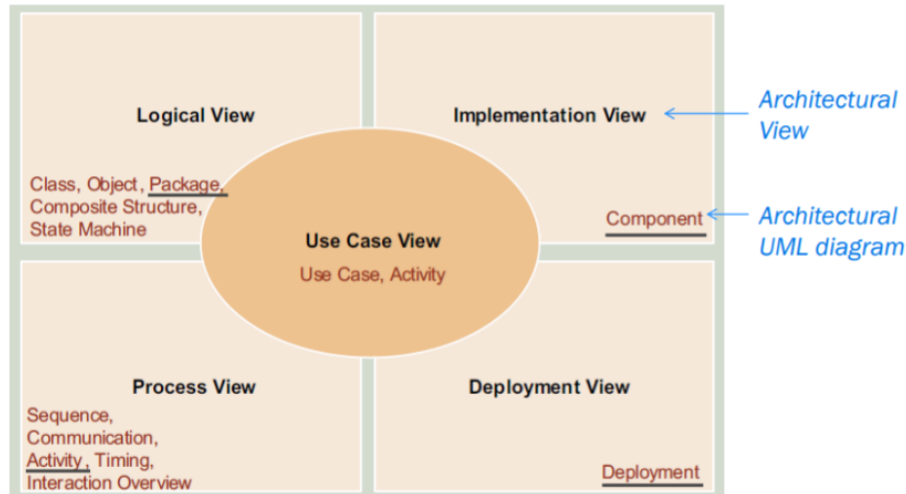
- Performance - Localize critical operations, minimize communications and levels of indirection
- Security - Use a layered architecture with critical assets in inner layers
- Safety - Localize safety-critical features in a small number of subsystems
- Availability - Include redundant components and mechanisms for fault tolerance
- Portability - Isolate platform dependencies in specific components
- Maintainability - Use fine-grained, loosely coupled, replaceable components.

# 2 Architectural Views

## 2.1 Civil engineering analogy

Architecture is best described by considering multiple views

## 2.2 4+1 view model of software architecture



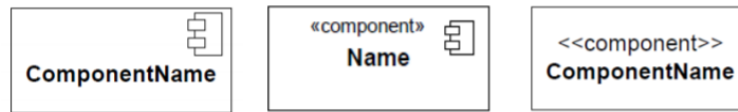
<p><b>Logical View:</b> Package diagrams (and others)</p> <p>Shows logical packages* and their relationships</p> <p>*division of responsibilities, independently of allocation to sw components or hw nodes</p>	<p><b>Implementation View:</b> Component diagrams</p> <p>Shows software components and dependencies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

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

## 3 Component Diagrams

- Components

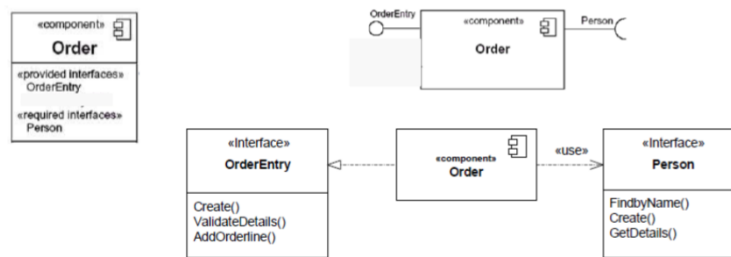
A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.



- Interfaces

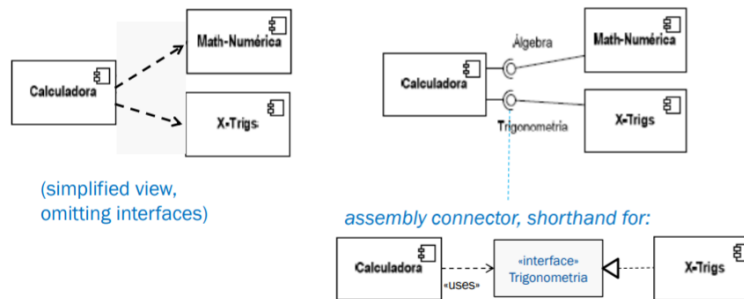
A component defines its behavior in terms of **Interfaces provided (realized)** and **Interfaces required (used)**

Components with the same interfaces are interchangeable.



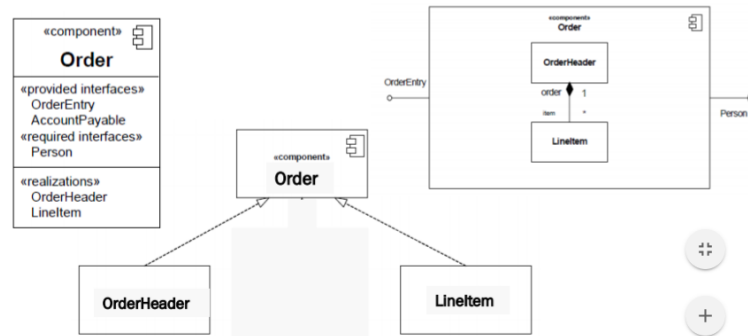
- Dependencies

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

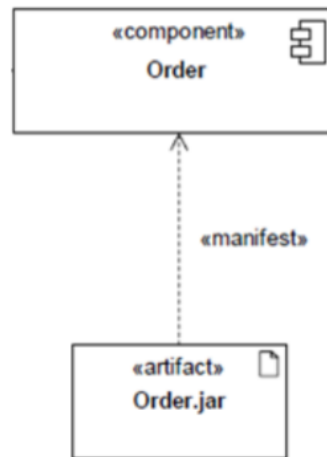


- Components and classes

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)



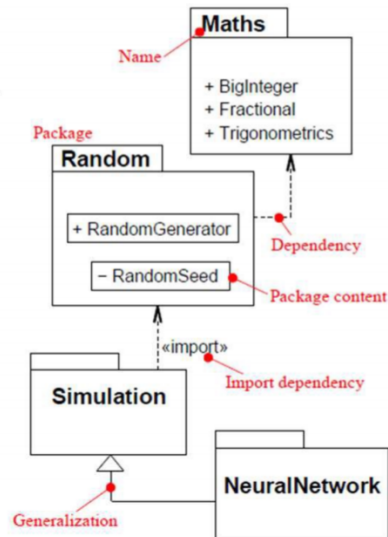
## 4 Deployment Diagrams

- **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).

## 5 Package Diagrams

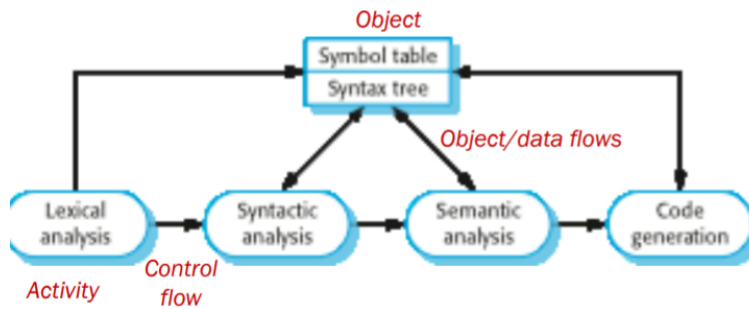
### Package diagrams

- Packages are a grouping mechanism in UML
- They may group elements of any type (even other packages)
- For the logical architecture, packages typically group classes
- May have stereotypes
  - «system»
  - «subsystem»
  - «layer», etc.



## 6 Activity Diagrams

### Compiler Architecture



## 7 Architectural 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.

### 7.1 Mode-view-controller (MCV)

For interactive processing Separates presentation (**V**) and interaction (**C**) from the application data/state (**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.

#### Disadvantages:

- Code overhead for simple data model and interactions.

### 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.

**Disadvantages:**

- Format for data transfer has to be agreed upon.
- Possible overhead in parsing/unparsing input/output data.
- Not really suitable for interactive systems

### 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.

**Disadvantages (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.

### 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.

**Disadvantages:**

- 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.

## 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.

**Disadvantages:**

- 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.

## 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**)

## 8 Process stages

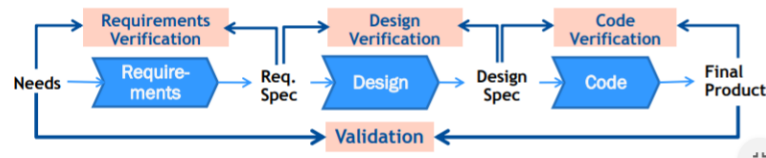
- There are a variety of different object-oriented design processes that depend on the organization using the process.
- Common activities in these processes include:
  - Define the system context and use cases;
  - Design the system architecture;
  - Identify the principal object classes in the system;
  - Develop design models;
  - Specify object interfaces (API).

## 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.

## 10 Verification vs Validation

- Verification – are we building the product right?  
Ensure (mainly through reviews) that intermediate work products and the final product are “well built”, i.e., conform to their specifications
- Validation – are we building the right product?  
Ensure (manly through tests) that the final product will fulfill its intended use in its intended environment  
Can also be applied to intermediate work products, as predictors of how well the final product will satisfy user needs.



## 11 Verification and validation technique

- Static techniques – involve analyzing the static system representations to find problems and evaluate quality
  - Reviews and inspections
  - Automated static analysis (e.g., with lint)
  - Formal verification (e.g., with Dafny)
- Dynamic techniques – involve executing the system and observing its behavior
  - Software testing
  - Simulation (e.g., animation of state machine model)

## 12 Software reviews and inspections

- Analysis of static system representations to find problems
  - Manual analysis of requirements specs, design specs, code, etc.
  - May be supplemented by tool-based static analysis
- Advantages (as compared to testing):
  - Can be applied to any artefact, and not only code
  - Can be applied earlier (thus reducing impact and cost of errors)
  - Fault localization (debugging) is immediate
  - Allows evaluating internal quality attributes (e.g., maintainability)
  - Usually more efficient and effective than testing in finding security vulnerabilities and checking exception handling
  - Very effective in finding multiple defects
  - Peer reviews promote knowledge sharing

## 13 Types of reviews

- Process
  - Personal review
  - Walkthrough
  - Peer review
  - team inspection
- Purpose
  - Technical review
  - Management review
- Target
  - Requirements review/inspection
  - Design review/inspection
  - code review/inspection
  - Reviewing plans and reports

### 13.1 Review best practices

- Use a checklist derived from historical defect data  
Makes the review more effective and efficient, by focusing the attention on the most frequent and important problems.
- Take enough review time (e.g., 200 LOC/hour is a recommended review rate by some authors)
- Take a break between developing and reviewing (in personal reviews)
- Combine personal reviews with peer reviews or team inspections
- Measure the review process and use data to improve  
size, time spent, defects found, defects escaped (found later).

## 14 Capture-recapture

The capture-recapture method is used to estimate the total defects (T) and number of defects remaining (R) based on the degree of overlapping between defects detected by different inspectors (A, B).

## 15 Key points

- Verification shows conformance with specification. Validation shows that the program meets the customer's needs.
- Static verification techniques involve examination and analysis of the program for error detection. Dynamic techniques involve executing the program for error detection.
- Program reviews and inspections are very effective in discovering errors.
- Source code in inspections is systematically checked by a small team to locate software faults.

## 16 Software Engineering Laws

### 16.1 Lei Nº3 - Principio fundamental da Arquitetura de Software

Qualquer problema de estruturação de software resolve-se introduzindo níveis de indireção.

Corolário: Qualquer problema de desempenho resolve-se removendo níveis de indireção.

### 16.2 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.