Software Engineering

FEUP-MIEIC-ESOF-2019-20

Ademar Aguiar, Filipe Correia

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Airbus A380
120 million lines of code (*avionics* and onboard support systems)

https://www.computerweekly.com/news/2240233948/Airbus-Interview-research-innovation-and-the-future-of-aviation

The History of Software Engineering

Grady Booch

IBM Fellow & Chief Scientist for Software Engineering

Email: gbooch@us.ibm.com
Twitter: @grady_booch
Web: computingthehumanexperience.com

V1.0


NOUN
RSET
0 1 2 3
KEY
REL

Waterfall

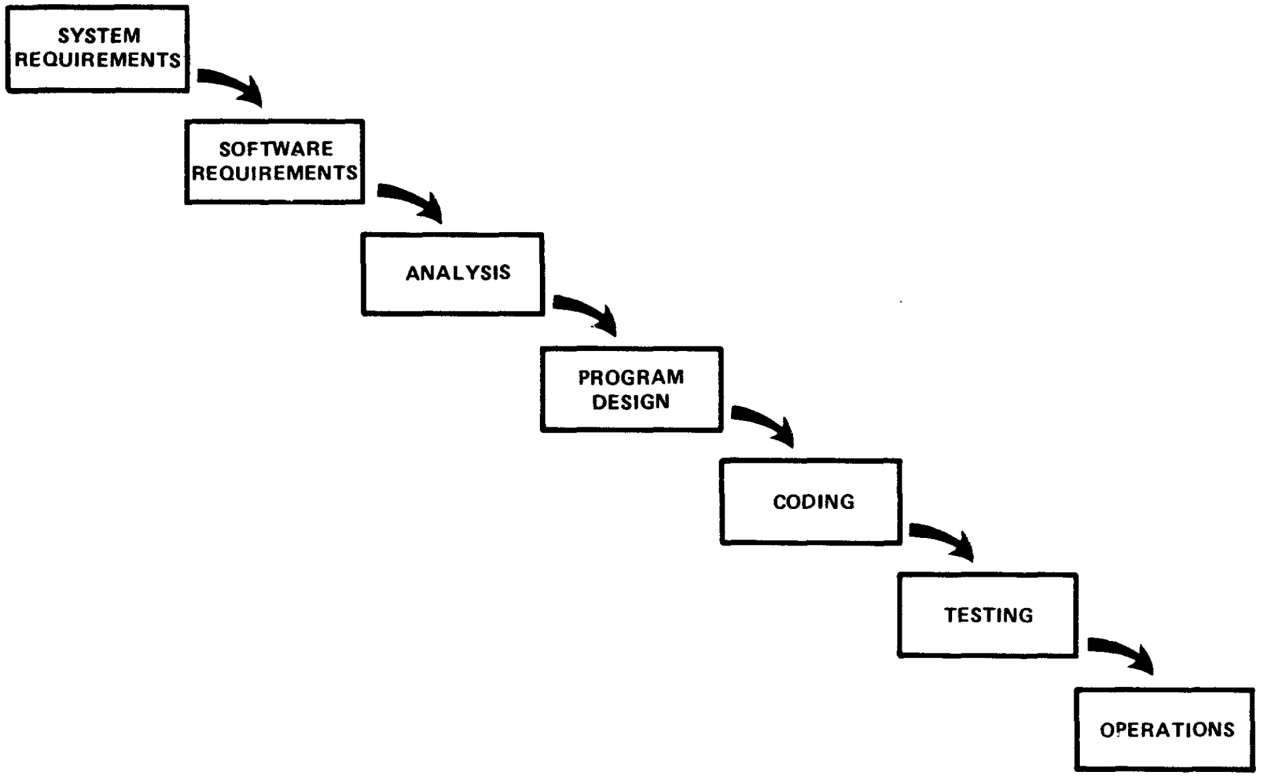
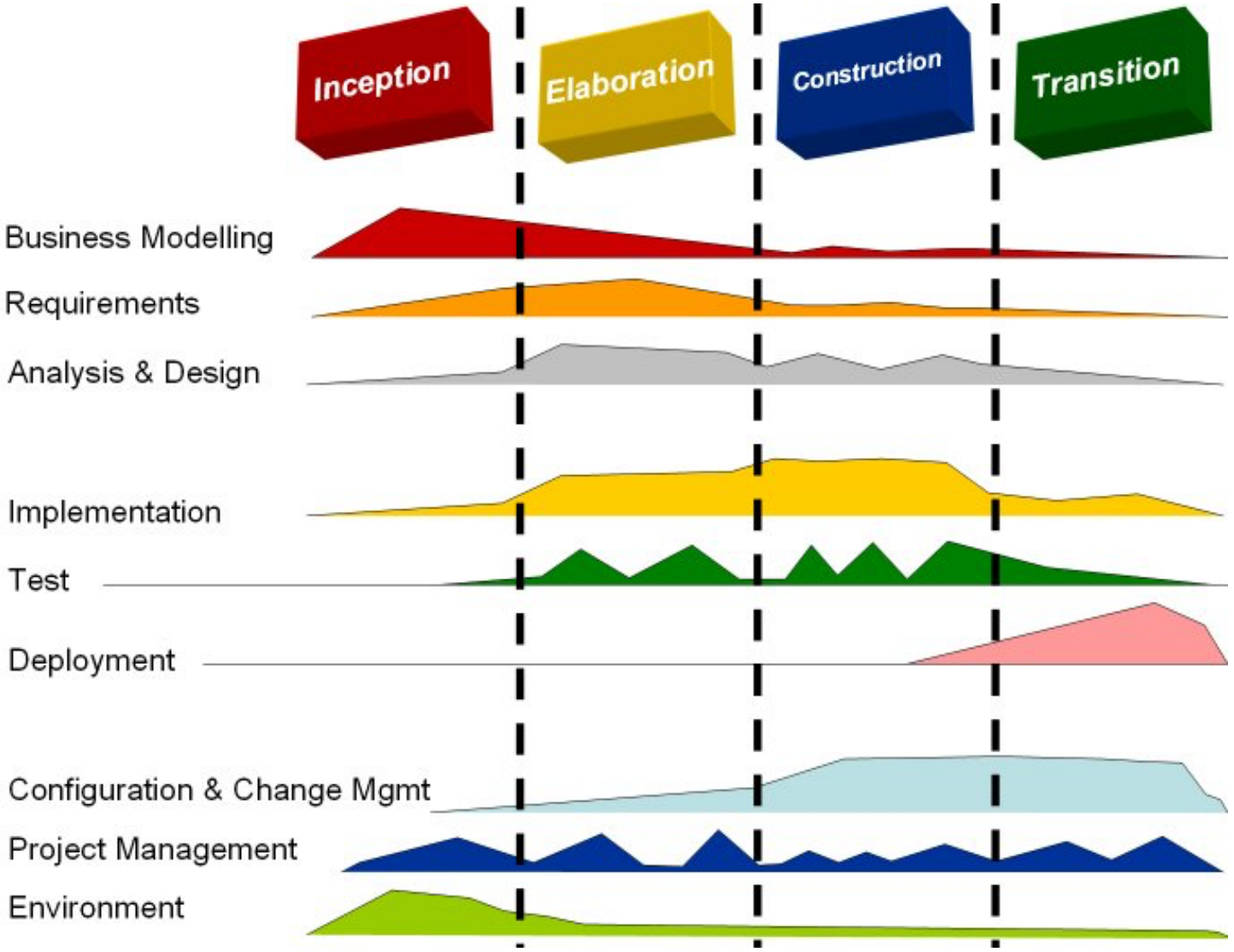
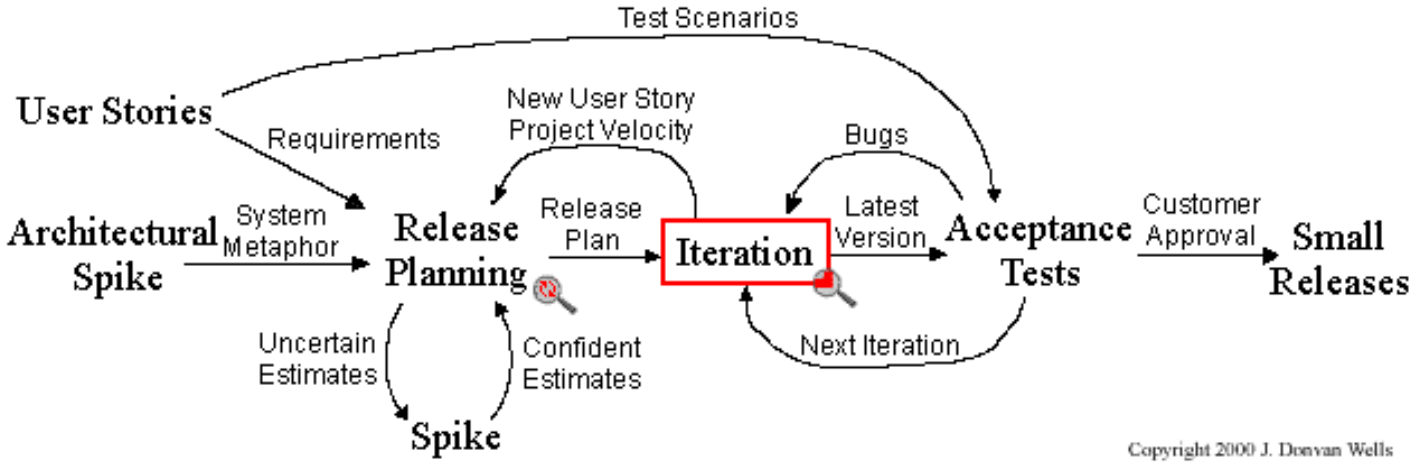


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

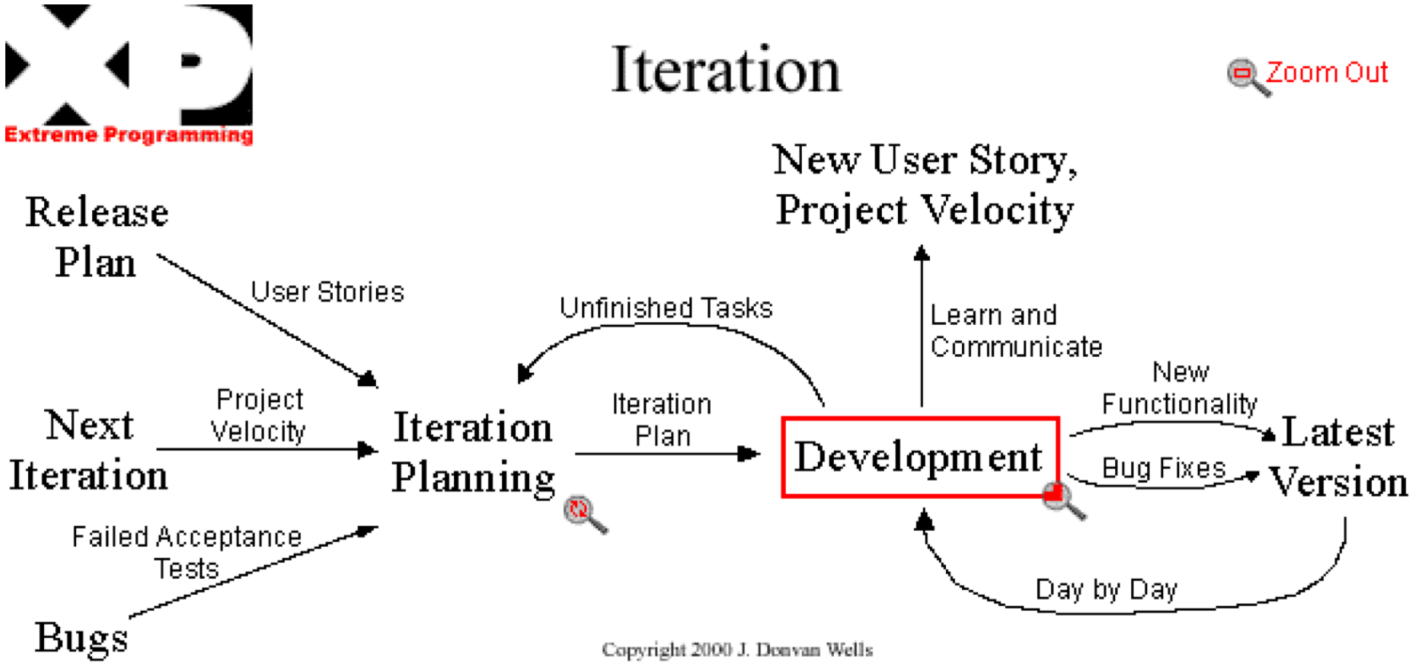
Rational Unified Process



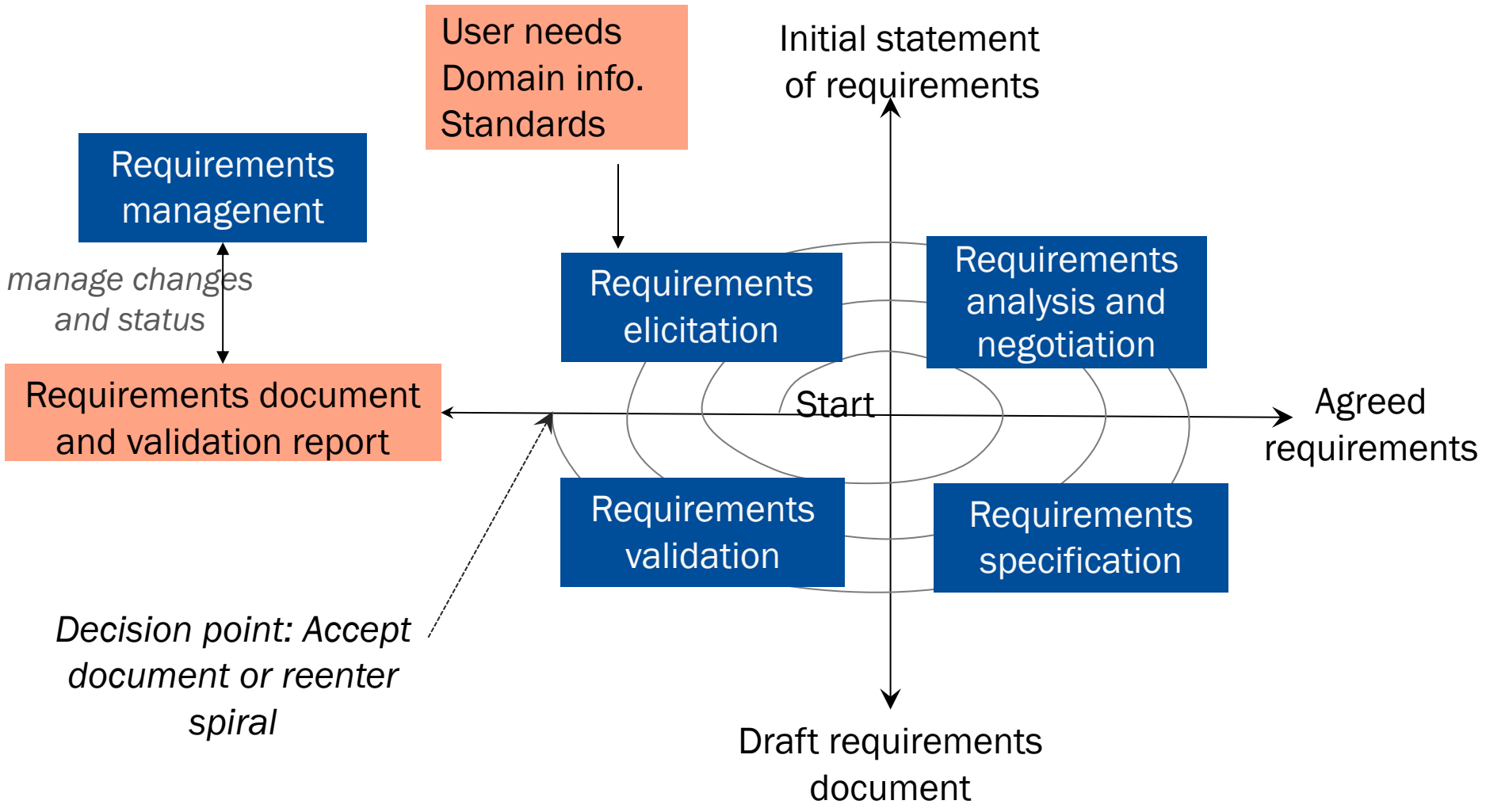
Extreme Programming Project

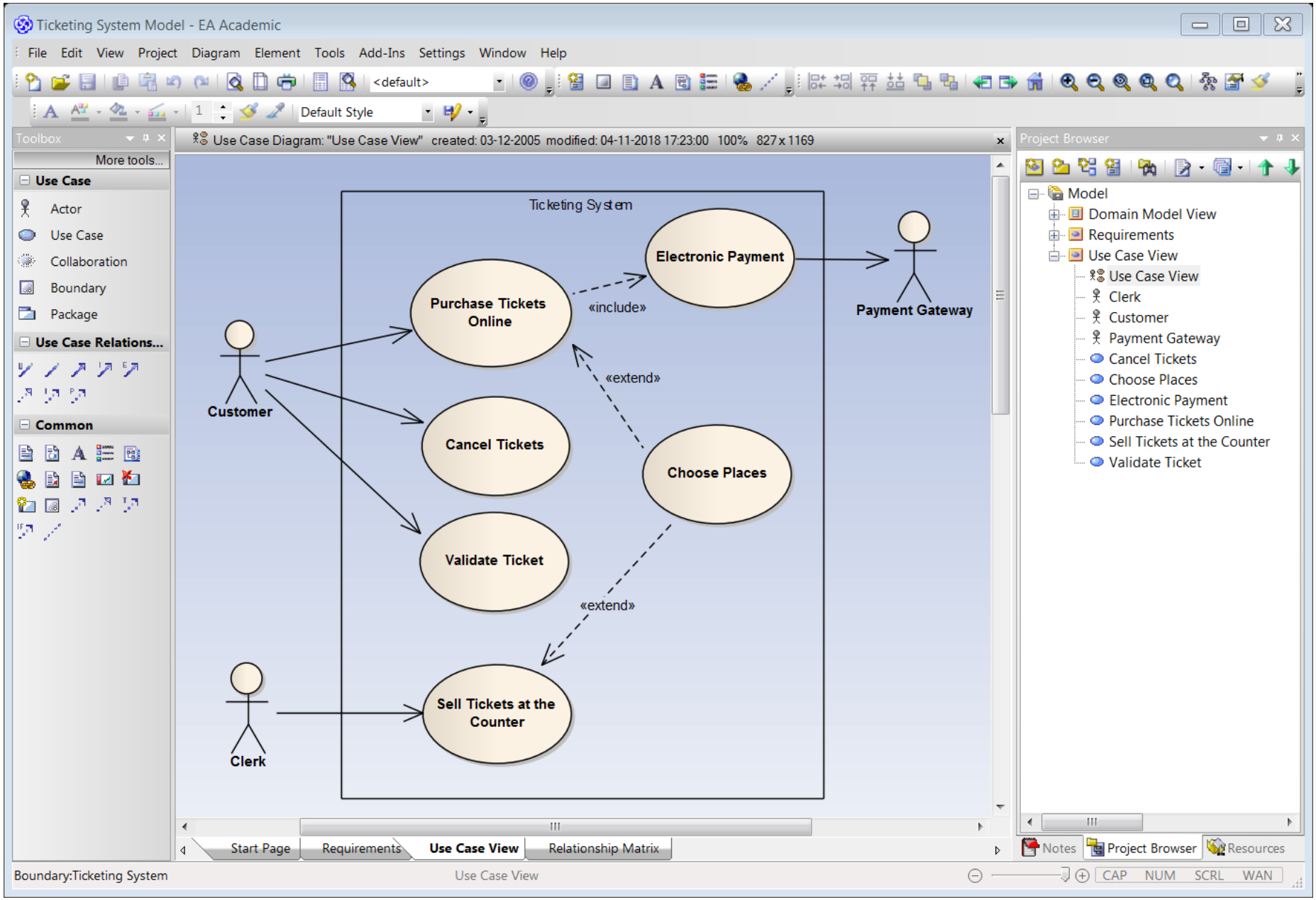


Iteration

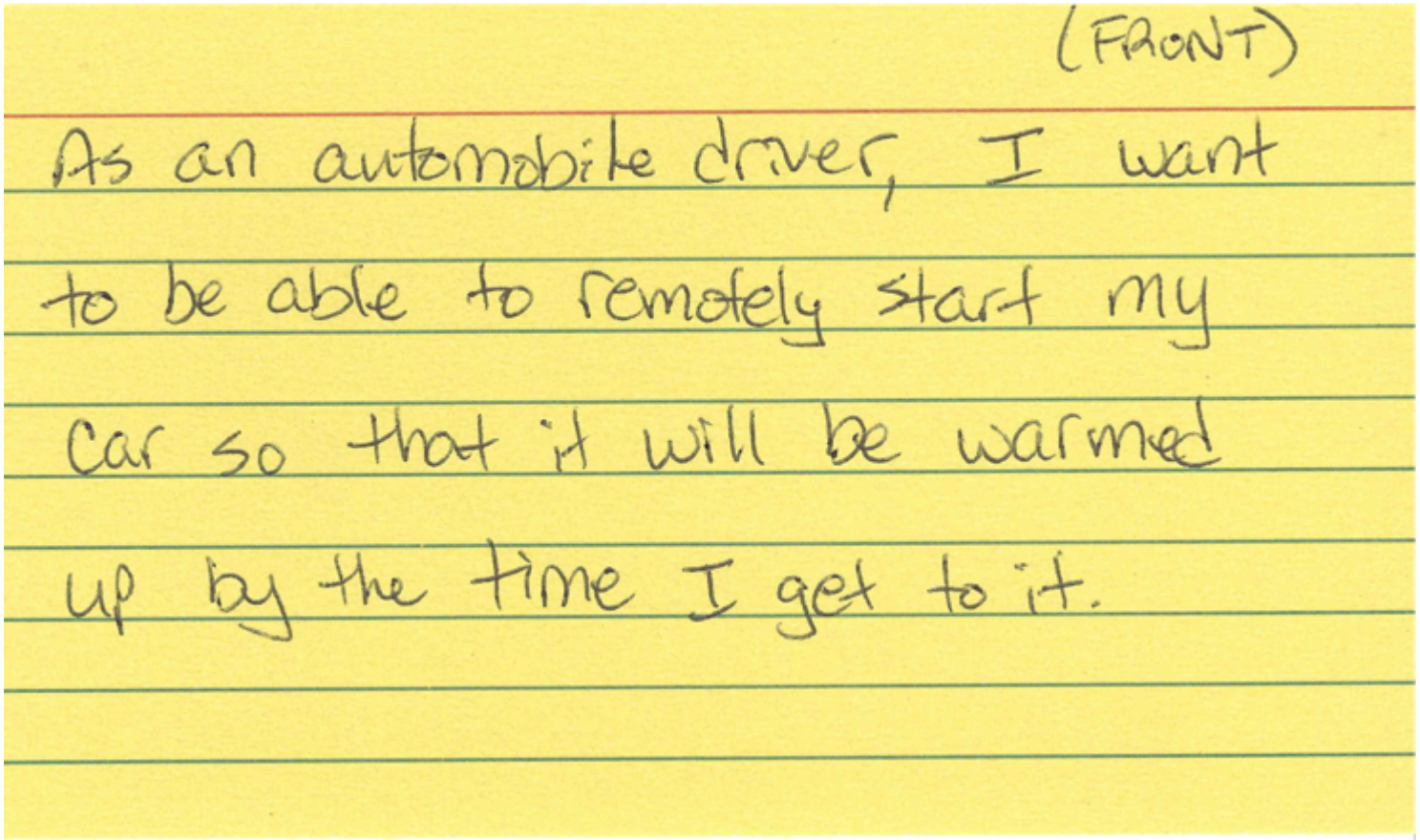


Requirements enginering process




Ticketing System Model - EA Academic
File Edit View Project Diagram Element Tools Add-Ins Settings Window Help
Toolbox
More tools...
Use Case
Actor
Use Case
Collaboration
Boundary
Package
Use Case Relations...
Common
Use Case Diagram: "Use Case View" created: 03-12-2005 modified: 04-11-2018 17:23:00 100% 827 x 1169
Ticketing System
Electronic Payment
Purchase Tickets Online
«include»
Payment Gateway
Customer
«extend»
Cancel Tickets
Choose Places
Validate Ticket
«extend»
Clerk
Sell Tickets at the Counter
Project Browser
Model
Domain Model View
Requirements
Use Case View
Use Case View
Clerk
Customer
Payment Gateway
Cancel Tickets
Choose Places
Electronic Payment
Purchase Tickets Online
Sell Tickets at the Counter
Validate Ticket
Start Page
Requirements
Use Case View
Relationship Matrix
Notes
Project Browser
Resources
Boundary:Ticketing System
Use Case View
CAP NUM SCRL WAN

User Stories – Example



Adapted from https://techbeacon.com/how-build-performance-your-user-stories

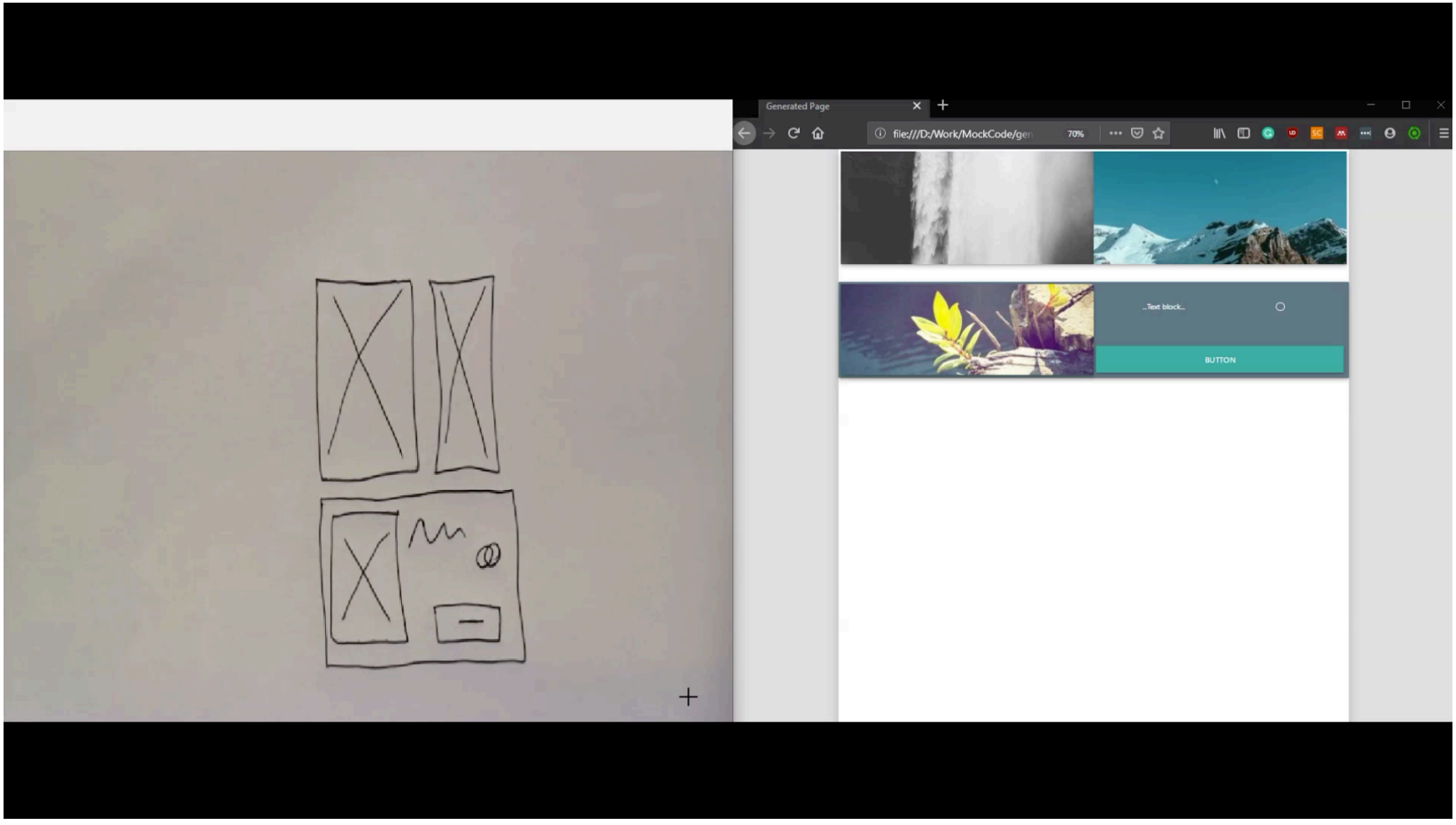
Prototyping

http://www.youtube.com/watch?v=5Ch3VsautWQ

“Live Web Prototypes from Hand-Drawn Mockups”

João Ferreira, André Restivo, Hugo Sereno Ferreira

Speeding up web development by reducing the feedback loop between designers and developers.



U.PORTO

Today...

- Architectural Design
- Coaching
  - João Damas & David Reis, MIEIC-MADS-2019-20
  - Discussions @ github
- Techonologies
  - Flutter
  - Things, beacons, microbits, by JP Dias
- Evaluation/Dashboard
- Q&A

Architectural Design

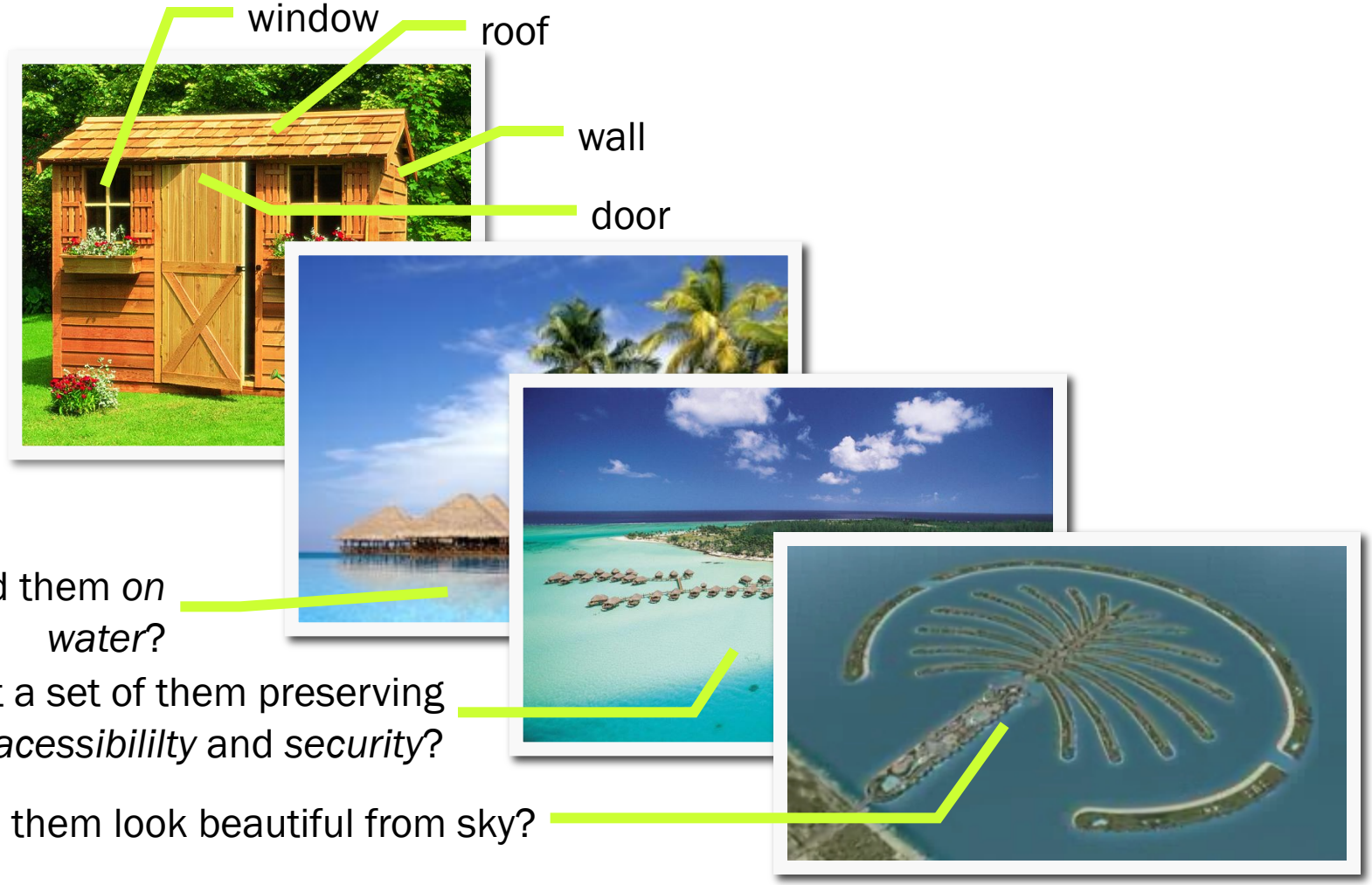
FEUP-MIEIC-ESOF-2019-20

Ademar Aguiar, Filipe Correia

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

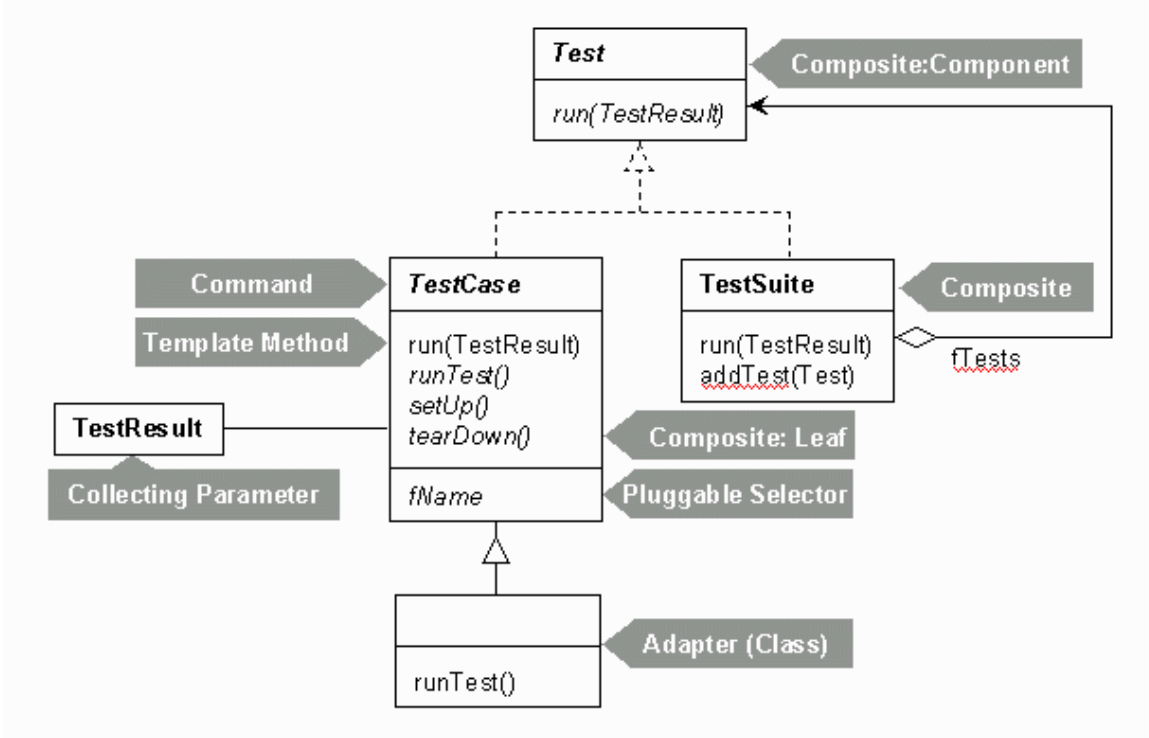
Architecting...

- To architect a small house seems to be very easy...



Architecting…

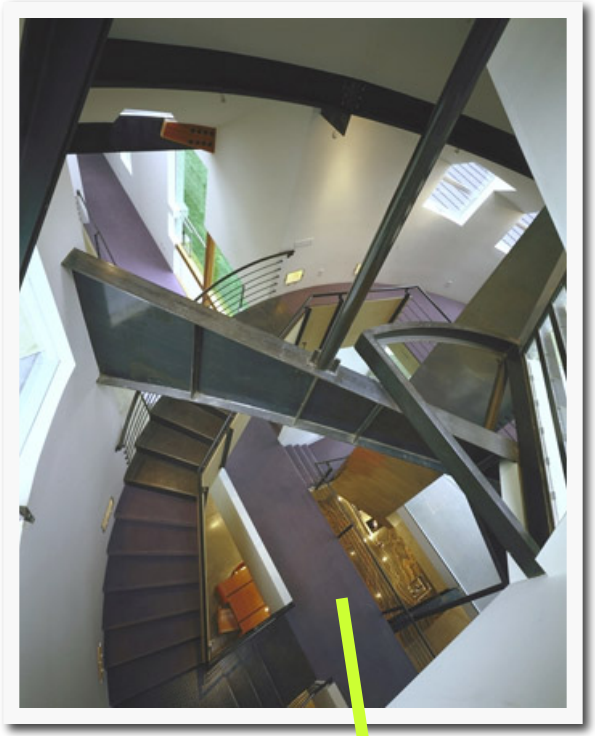
- “Voilá” a well know“little software house”: xUnit



- Which building elements can we identify here? Hmm…

Architecting…

- Not so easy would be to architect a family house…



How to layout and connect different rooms and spaces?

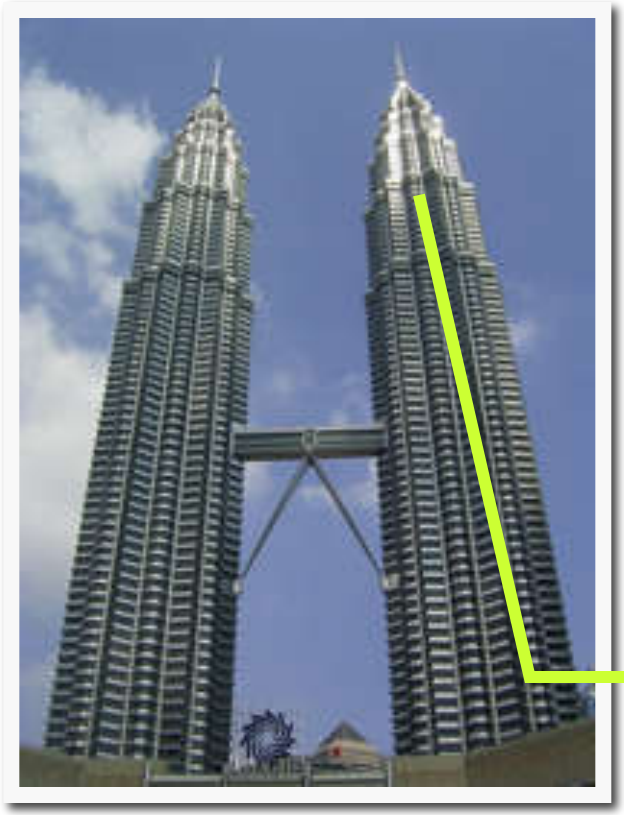
Architecting…

- What about a 6-story building…

How to layout and connect several houses?

Architecting…

- What about a skyscrapper of 88 levels and 452 m high…



How to build and maintain such building, and make it live?

Architecting

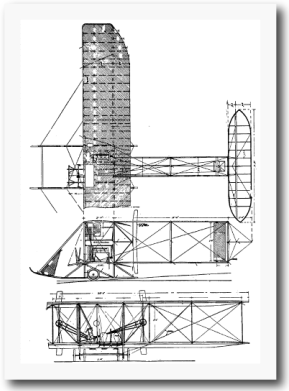
- and even more… 828 meters / 2,717 feet, 162 stories

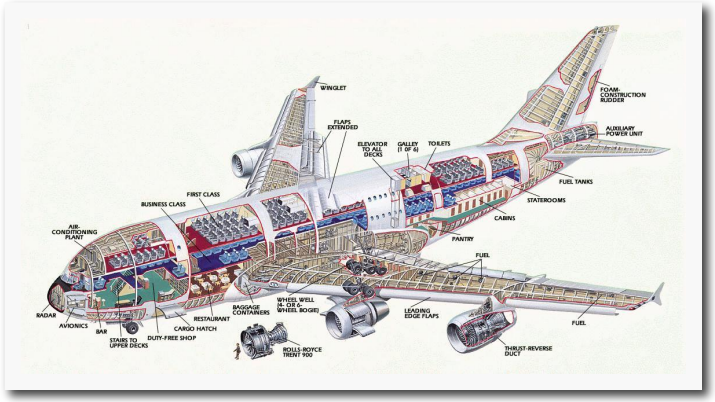
Architecting…

- Main diferences between these buildings
  - Dimensions
  - Cost
  - Time
  - Process
  - Skills and teams
  - Materials and technologies
  - Risks
  - Robustness
  - Longevity…

Traditional Engineering

- Traditional engineering have evolved based on “stable systems”, mature, good references
- Stable systems have evolved over time
  - By trial and error
  - By constant adaptation and specific needs
  - By reuse and refinement of well-proven solutions
  - By several advances in materials, processes and standards





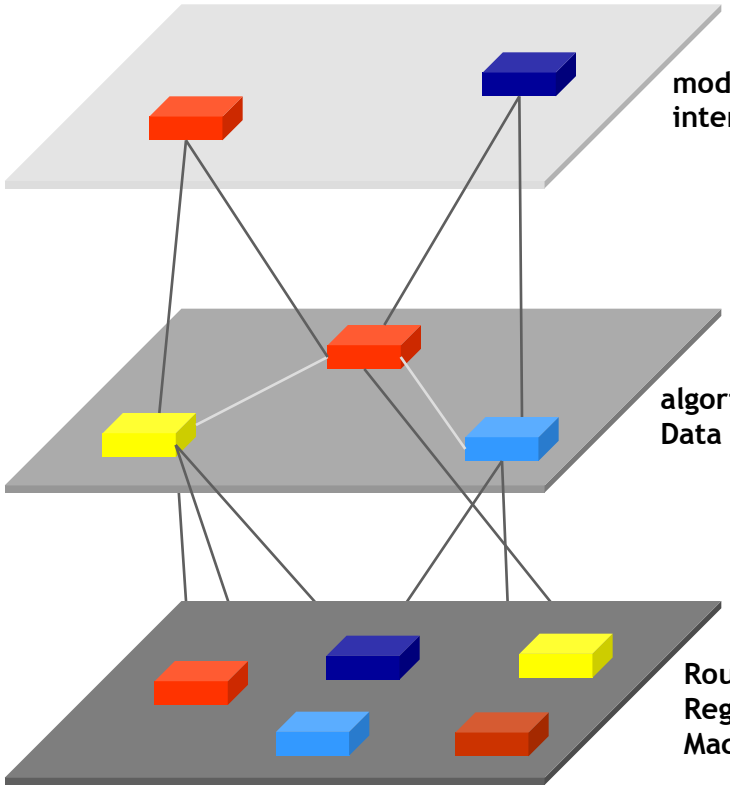
Software Engineering: is it harder? Yes!

- Invisible constructions (logical product)
- Temporal nature of software is complex and hard to understand (discrete systems)
- Big needs to change and evolve along lifespan
- Software must be very easy to adapt; each case is a case
- Underlying technologies evolve very fast
- Short history (since 1960?)
- No “nature laws” or “phyical laws” to conform to…

The Role of Software Architecture

- To control overall system complexity (technological & scale)
- To ensure system integrity
- To ensure required quality attributes: scalability, interoperability, usability, performance, cost, schedule, modularity, etc.
- To improve development predictability
- To establish tradeoffs that influence system development and future evolution
- To ease the collaboration between different development teams
- The Key Challenge: “to architect and design timeless and (ultra) large-scale software…” (not only buildings, but cities of buildings…)

Software design: key levels



Architecture

Source code

Executables

modules
interconnections

Which modules?
How to connect them?

algorithms
Data structures

Which data structure?
Which algorithm?

Routine stack
Register assignment
Machine-code

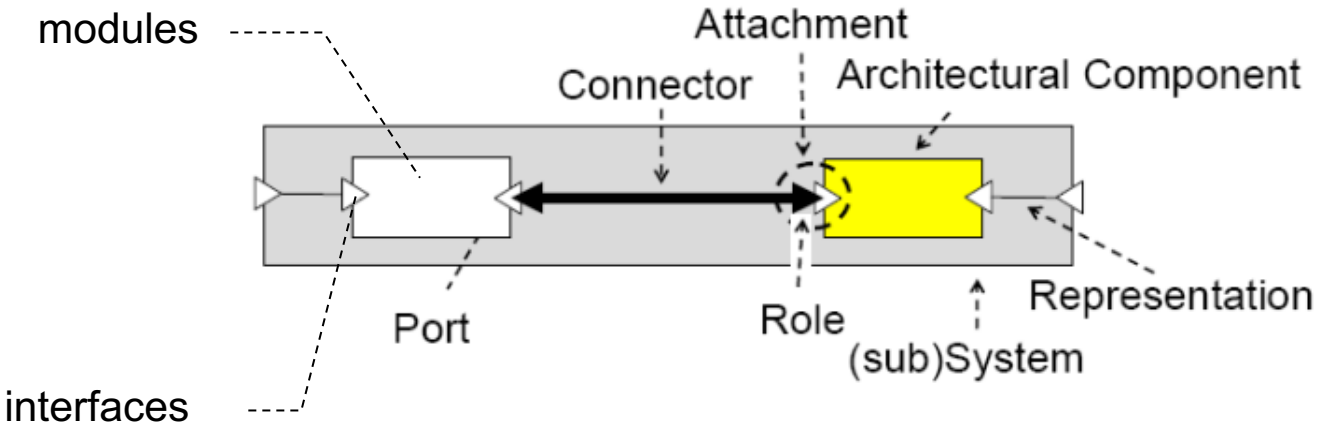
How to manage memory?
Which instructions?

Software Architecture

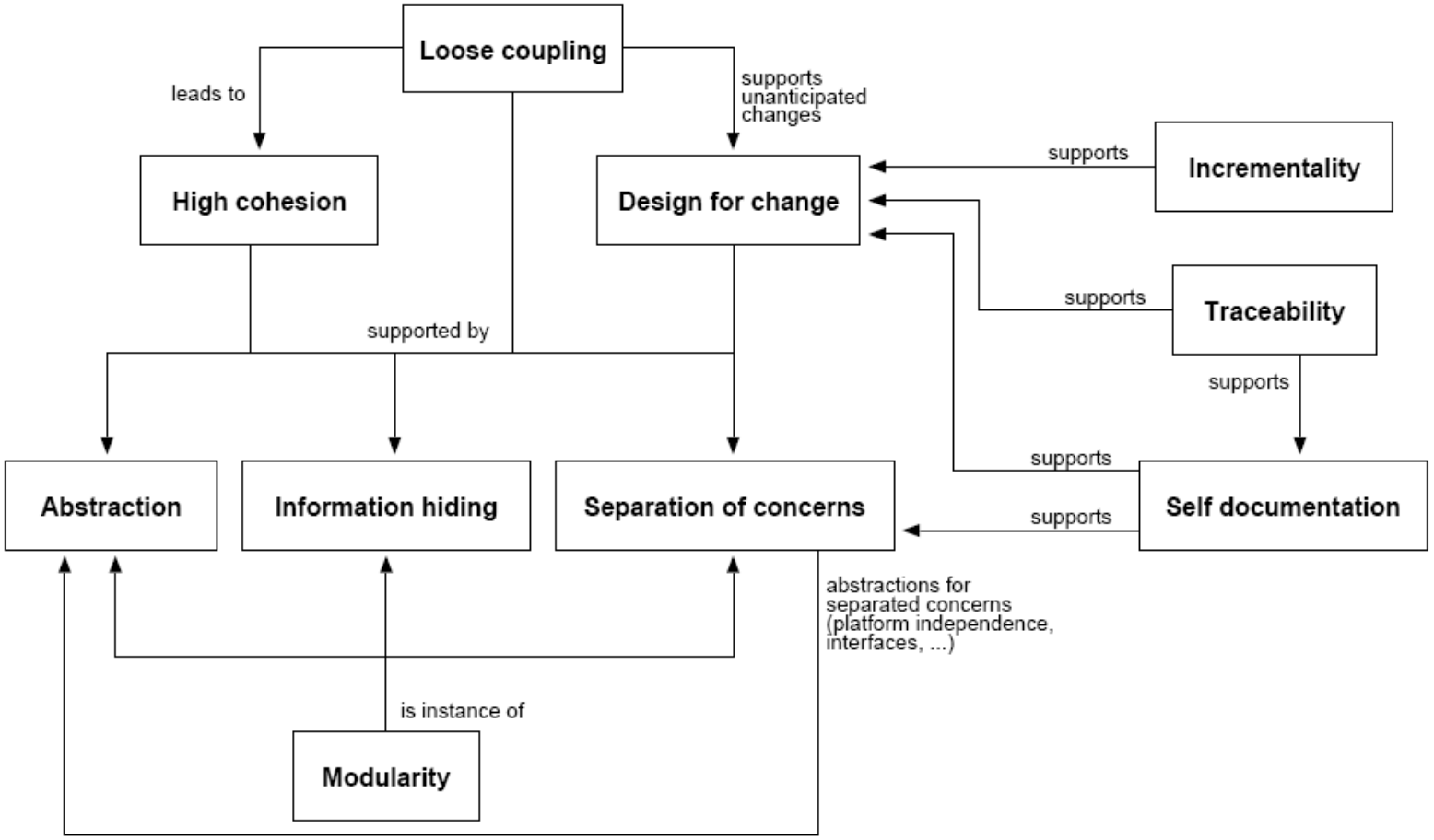
- Software Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

  IEEE Recommended Practice for Architectural Description of Software Intensive Systems (IEEE 1471-2000)

  - Which structural elements and interfaces is the system made of?
  - Which composition mechanisms are used to compound elements?
  - Which colaboration behaviors between elements?
  - Which main architectural style enforms the system?



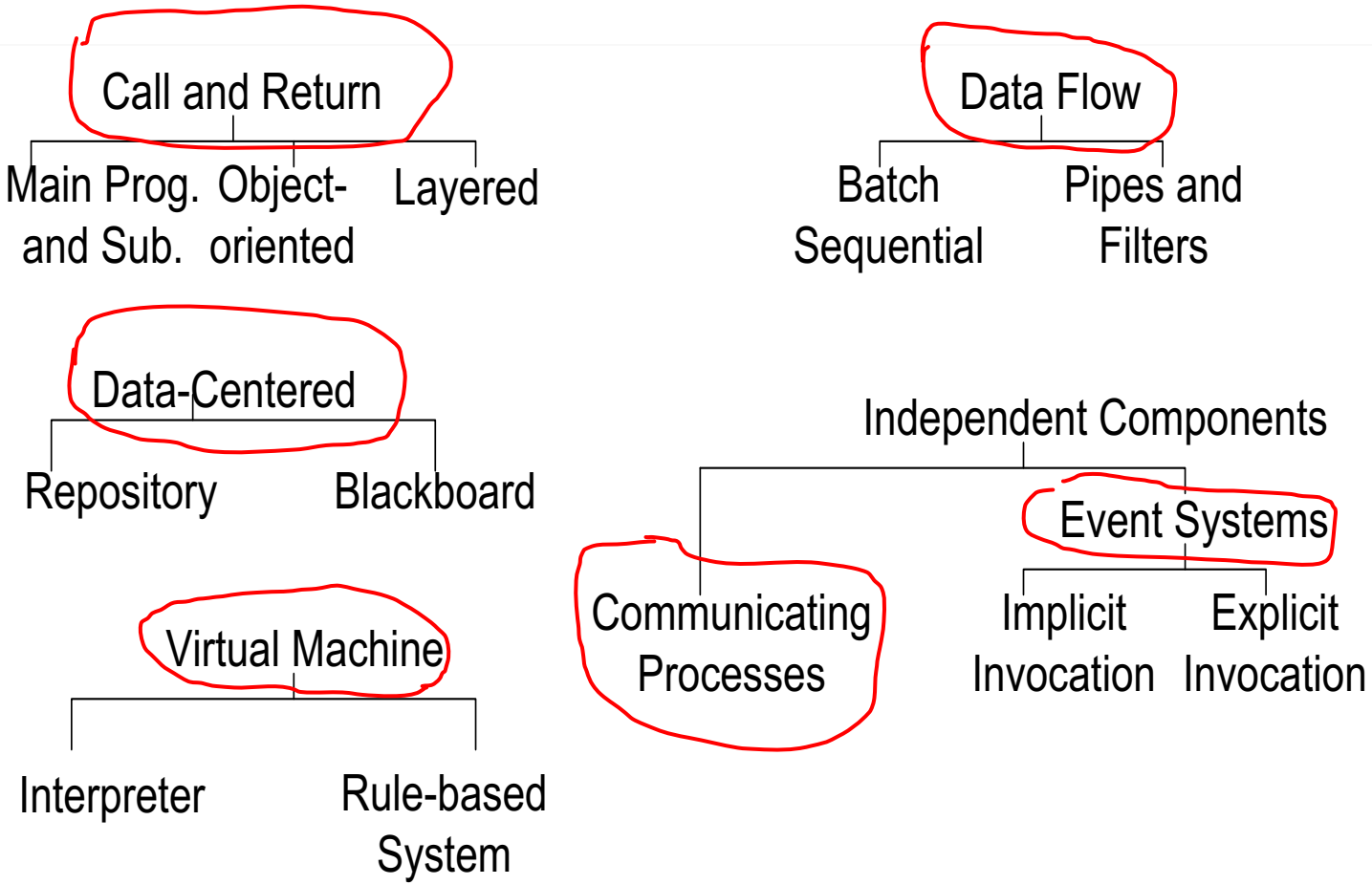
Software Architecture: key principles



Architectural Knowledge: Reusing

- To architect solutions from scratch is hard and not effective in most cases
- Good architects reuse solutions that worked well before and adapt them smoothly and incrementally to new situations.
- Therefore it is very important:
  - to know well proven solutions
  - to reuse generic solutions
  - to adapt existing solutions
  - and then to add innovation over them to fit the needs.

Architectural styles



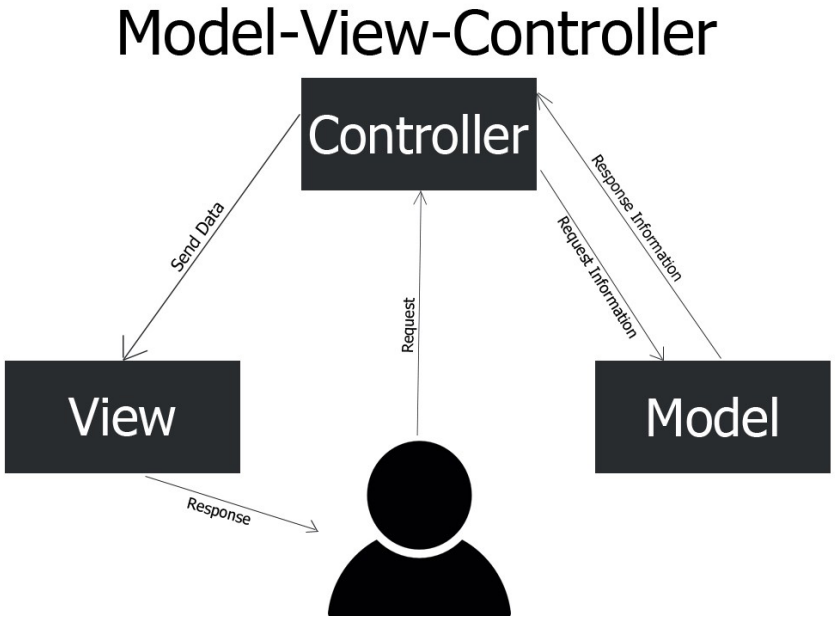
From Chapter 5, Software Architecture in Practice, p. 95

Architectural Patterns

- Mode-view-controller (MVC)
  - For interactive processing
- Pipes and filters (or data flow)
  - For batch processing
- Layered architecture
  - For complex systems with functionalities at different levels of abstraction
- Repositories (data centric)
  - For accessing & manipulating shared data by multiple subsystems

Model-View-Controller (1)

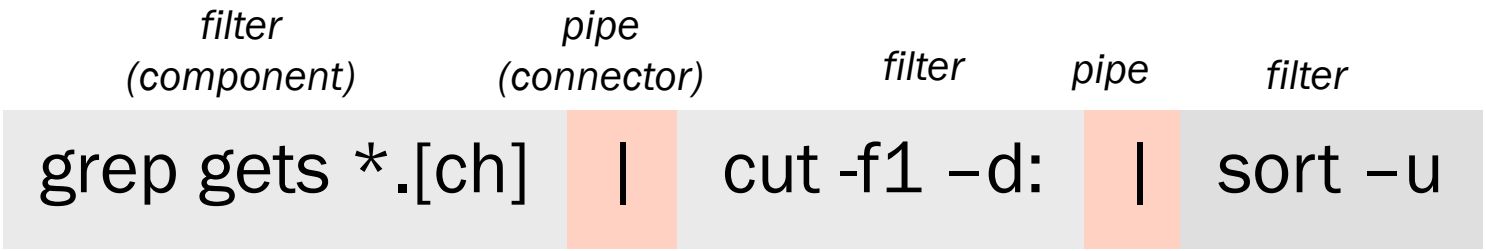
- Separates presentation (V) and interaction (C) from the application data/state (M), by structuring the system into three logical parts that interact with each other.
- Commonly used in GUI & Web development frameworks (Swing, Ruby on Rails, etc.).



Pipes and Filters (or Data Flow)

- Organizes the system as a set of data processing components (filters), connected so that data flows between components for processing (as in a pipe).

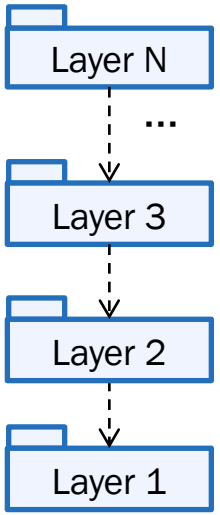
- The name comes from Unix, e.g. (shell command):



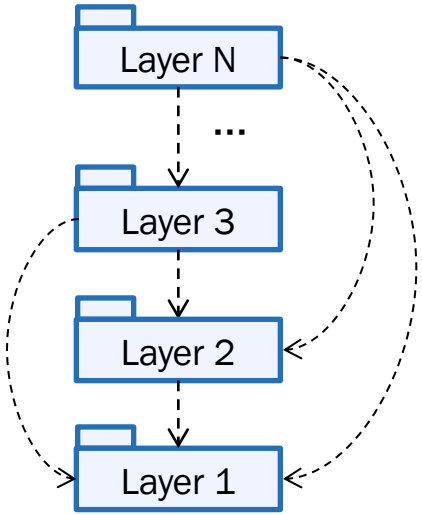
Layered Architecture

- Organizes the system into a set of layers, each of which groups related functionality and provides services to the layer above.

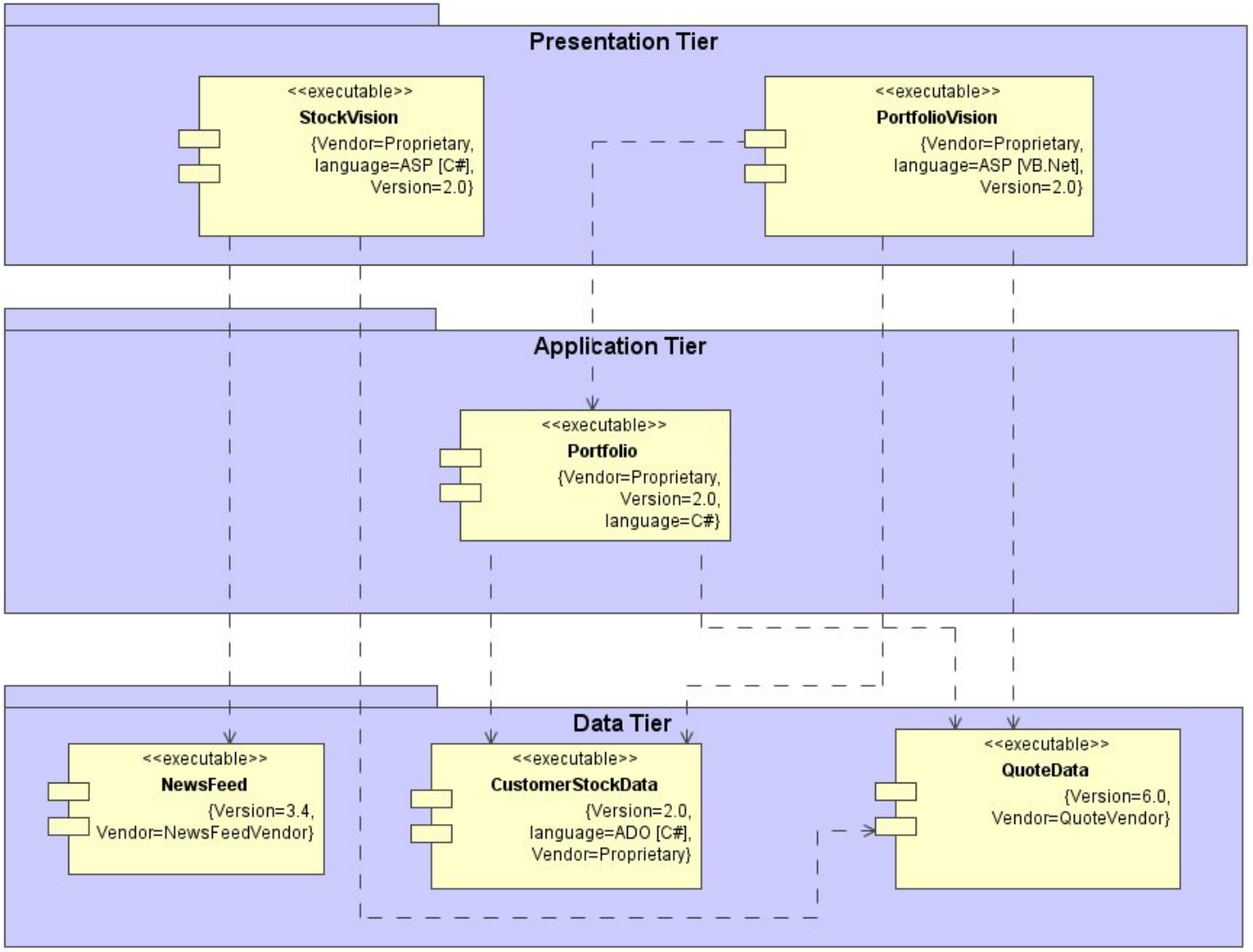
Strict: each layer can only interact with the layer directly below (default).



Relaxed: each layer can interact with any lower layer (to avoid).

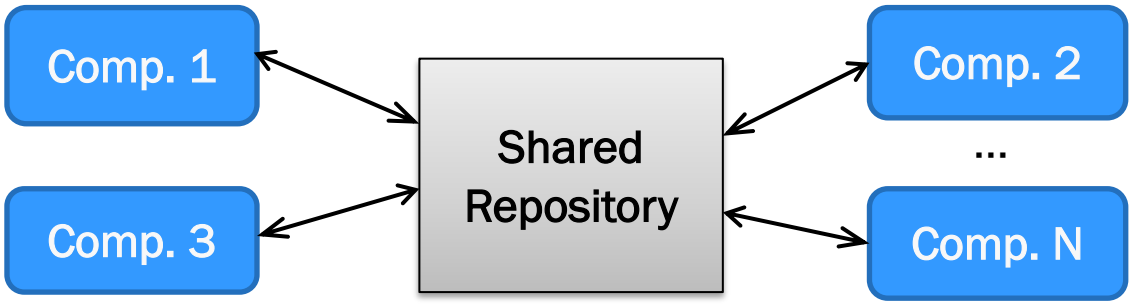


Example: Three-Layered Services Application



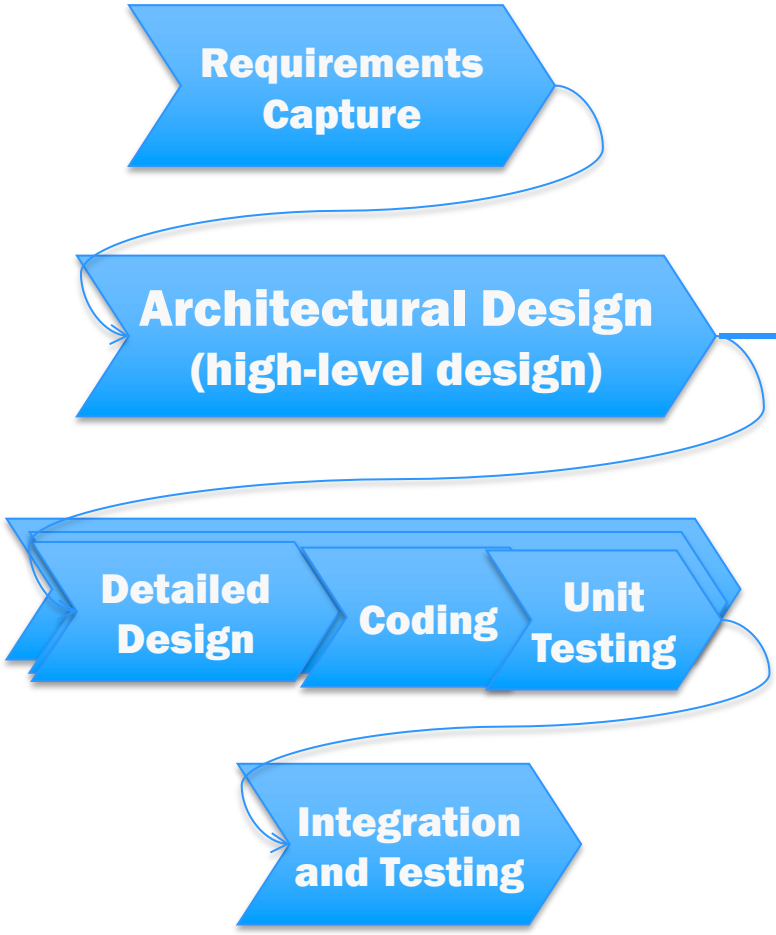
Repository Architecture

- 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.



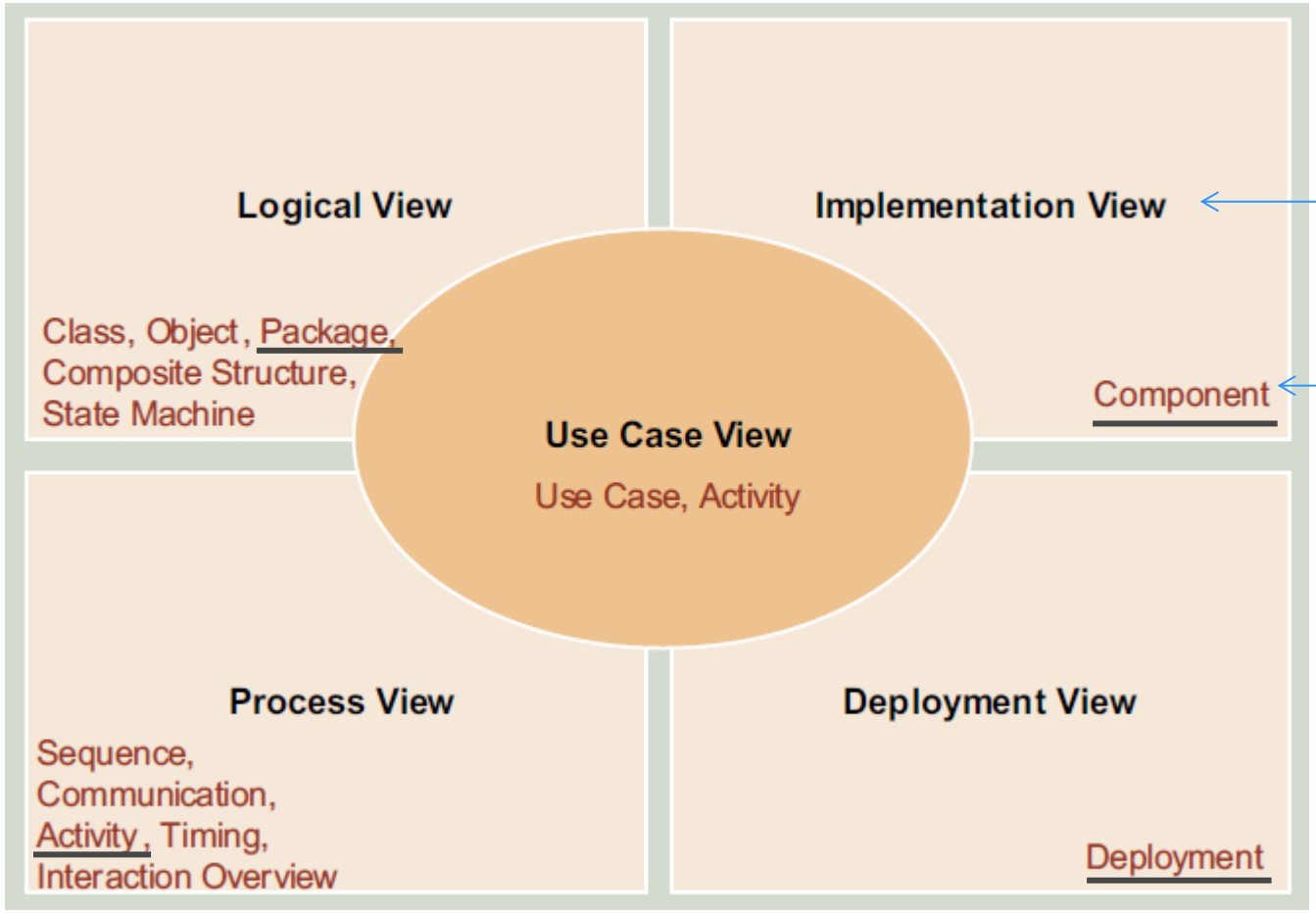
- Variants:
  - **Passive** repository: passively accepts requests from componentes
  - **Active** repository: changes to the repository trigger component execution

Typical outputs of architectural design



- Architecture description document
  - Architecturally significant requirements
  - Architectural views
  - Components' responsibilities & interfaces
  - Common mechanisms
  - Technologies used & rational
  - Other significant decisions
- Architectural models
  - UML diagrams
- Architectural prototypes
  - For validating architectural design decisions
- Integration test scenarios
  - Clarify interactions among components

4+1 view model of software architecture



Source: Architectural Blueprints — The “4+1” View Model of Software Architecture, Philippe Kruchten, Philippe, IEEE Software 12 (6), pp. 42-50, 1995

4+1 view model of software architecture

- **Logical view**: shows the key abstractions in the system as objects, object classes, or their packages

  → UML package diagrams

- **Implementation view**: shows how the software is decomposed (into software components) for development

  → UML component diagrams

- **Deployment view**: shows the system hardware and how software components are distributed by hardware nodes

  → UML deployment diagrams

- **Process view**: shows how, at run-time, the system is composed of interacting processes

  → UML activity diagrams (or other diagrams, depending on the focus)

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

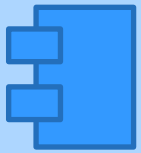
Architectural views and UML diagrams

Logical View:
Package diagrams (and others)

Shows logical packages* and their relationships

*division of responsibilities, independently of allocation to sw components or hw nodes

Implementation View:
Component diagrams



Shows software components and dependencies among them

Process View:
Activity diagrams (and others)

Shows processing steps, data/object stores, data/object-flows, and opportunities for parallelization

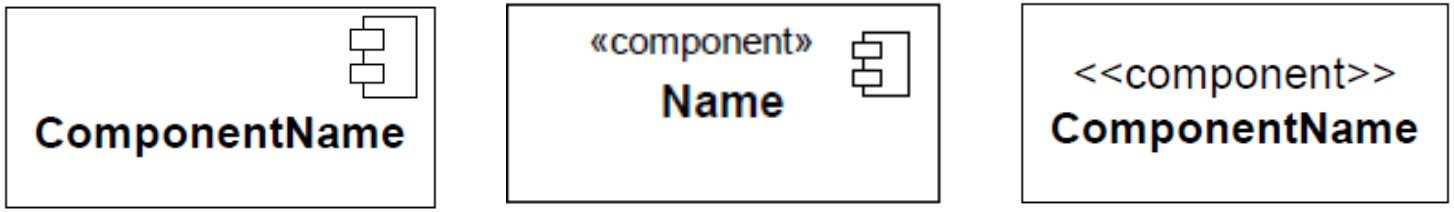
Deployment View:
Deployment diagrams

Shows hardware nodes, communication relationships and software artifacts deployed on them

Component Diagrams

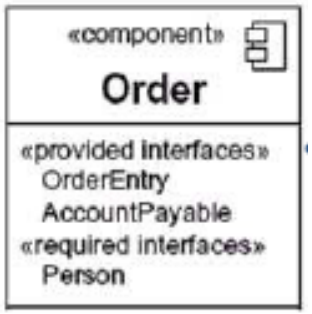
Components

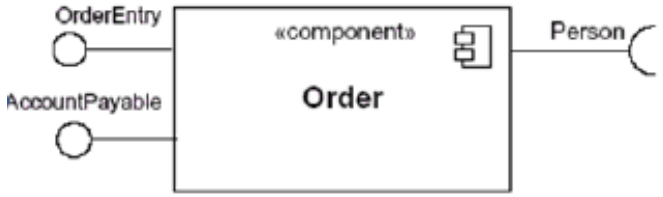
- A **component** represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment (source: UML spec)
  - examples: libraries, executables, etc.
  - components can be implemented and replaced independenlty
- Components are the central concept in **Component-based Software Engineering** (CBSE) as units of **reuse**
- Equivalent notations:

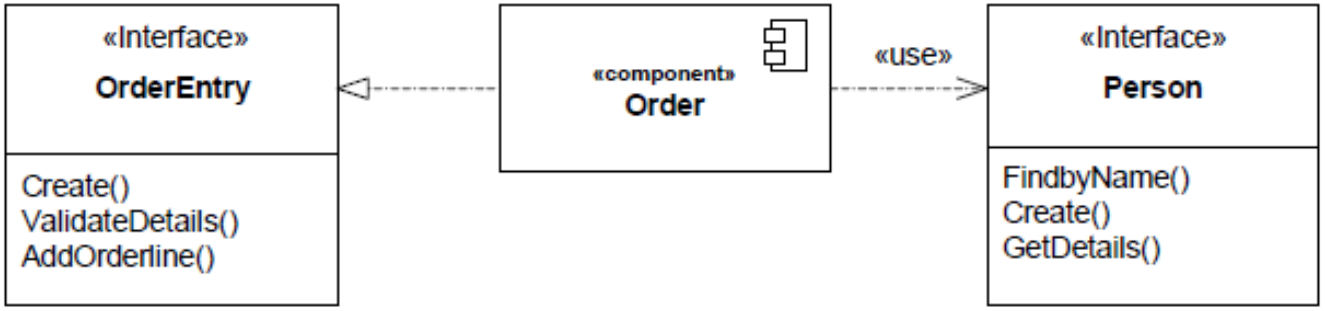


Interfaces

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

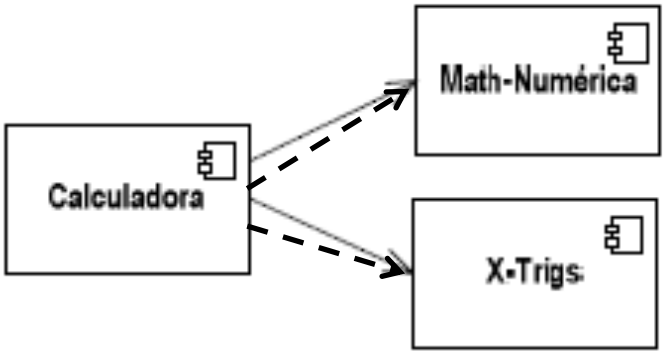




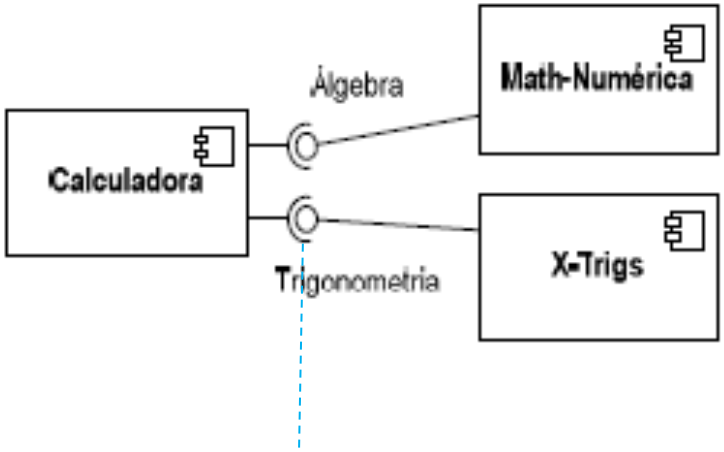


Dependencies

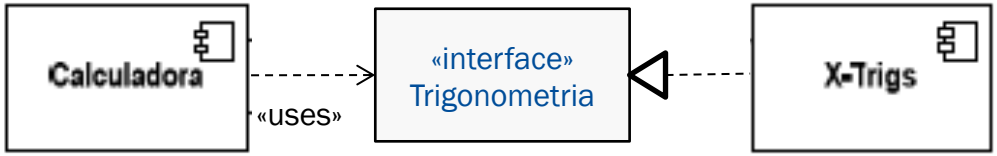
- To promote intercangeability, components shouldn't depend directly on other components but rather on interfaces (that are implemented by other components)



(simplified view, omitting interfaces)

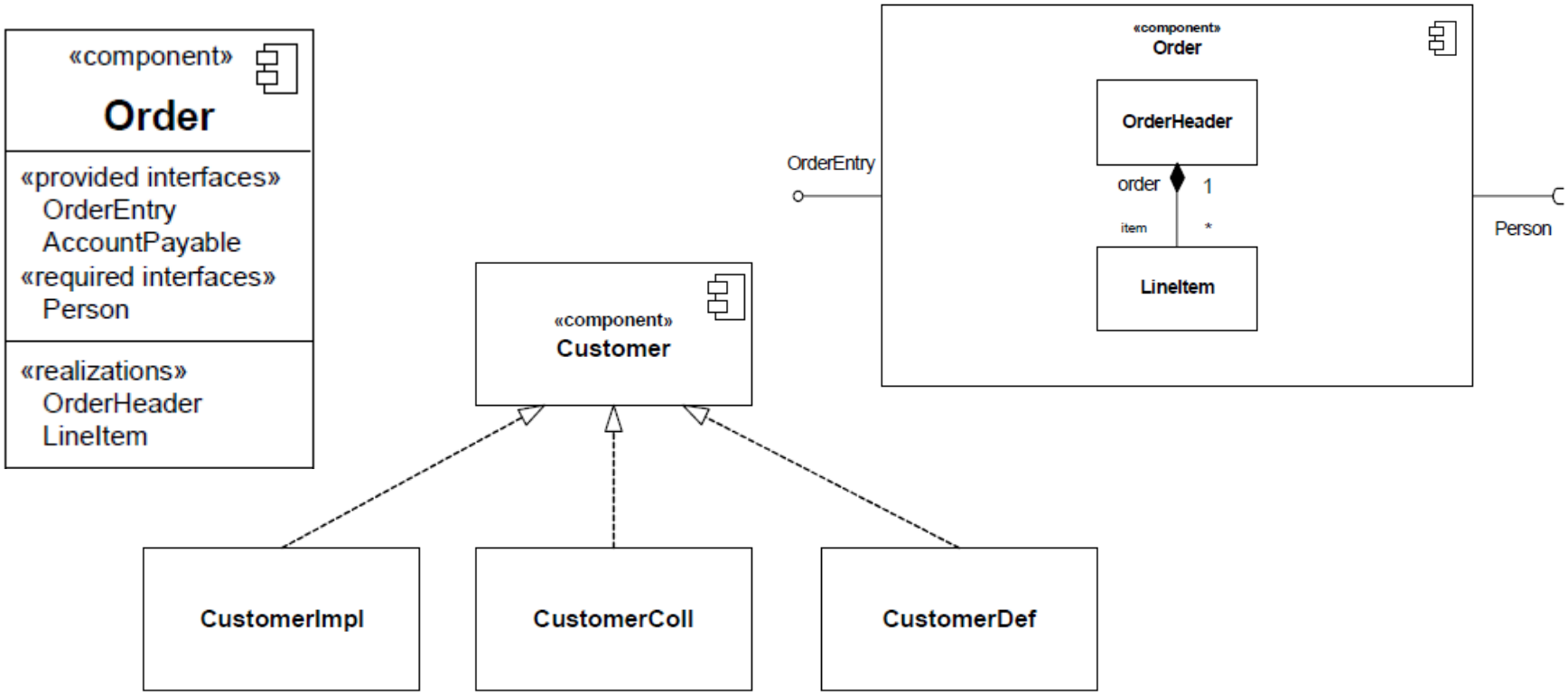


assembly connector, shorthand for:



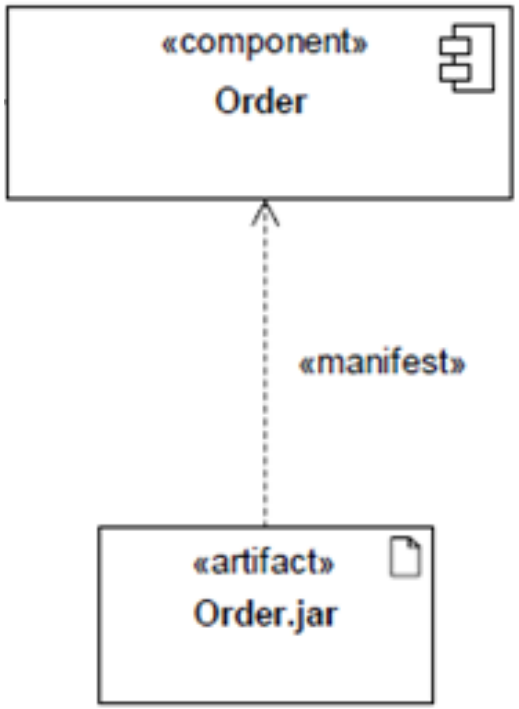
Components and classes

- The behavior of a component is usually **realized** (implemented) by its internal classes
- Equivalent notations:

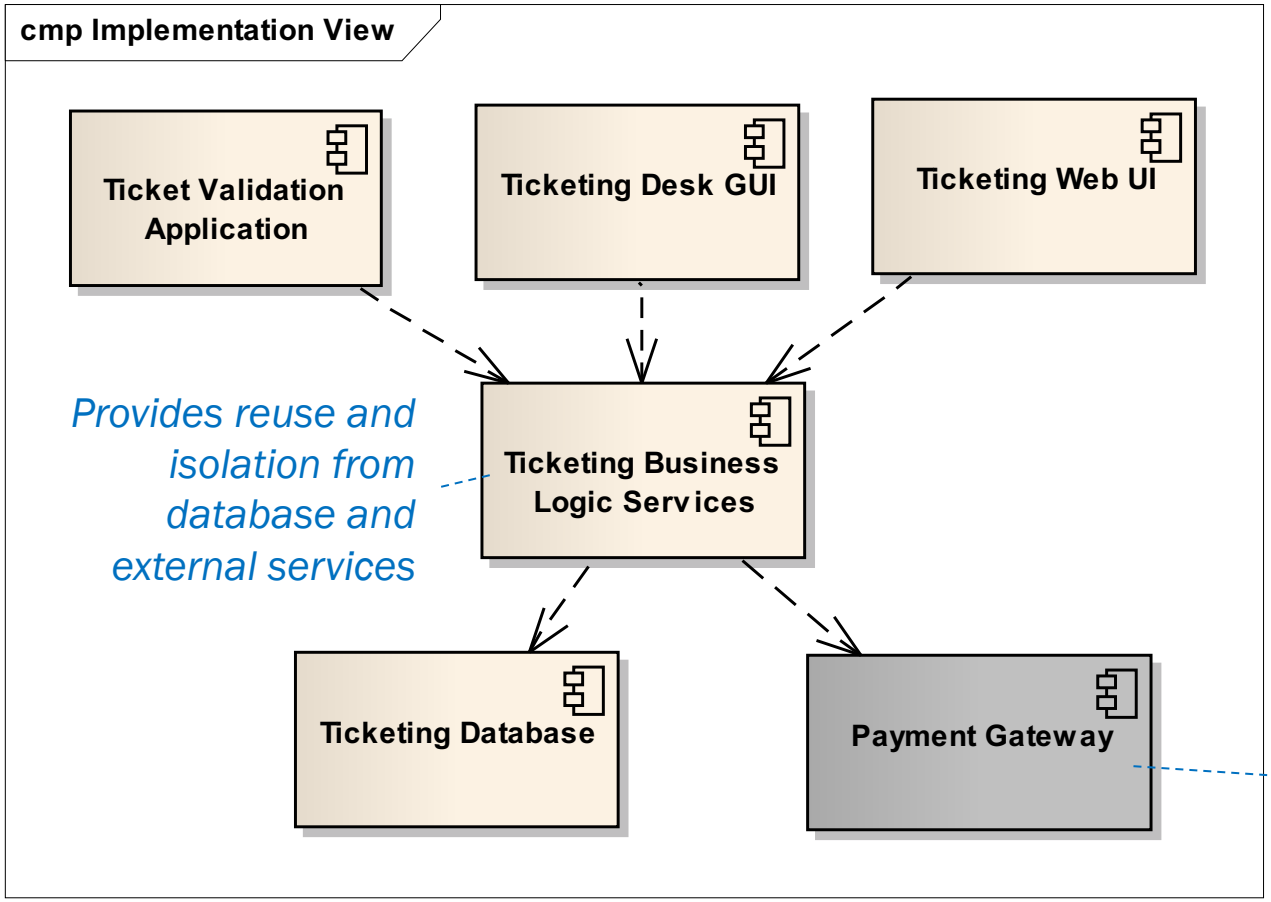


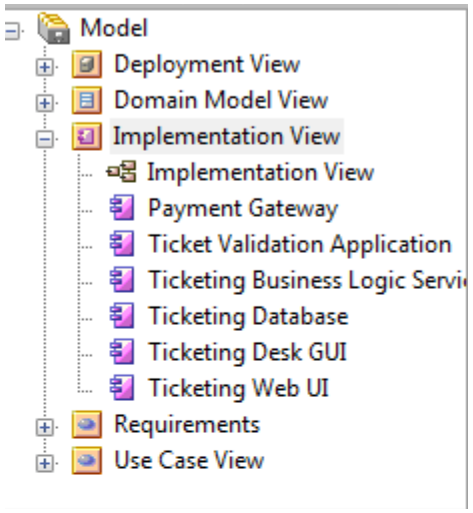
Components and artifacts

- Components manifest physically as artifacts (that may be deployed in hardware nodes)
- Notation:



Example: Electronic Ticketing System



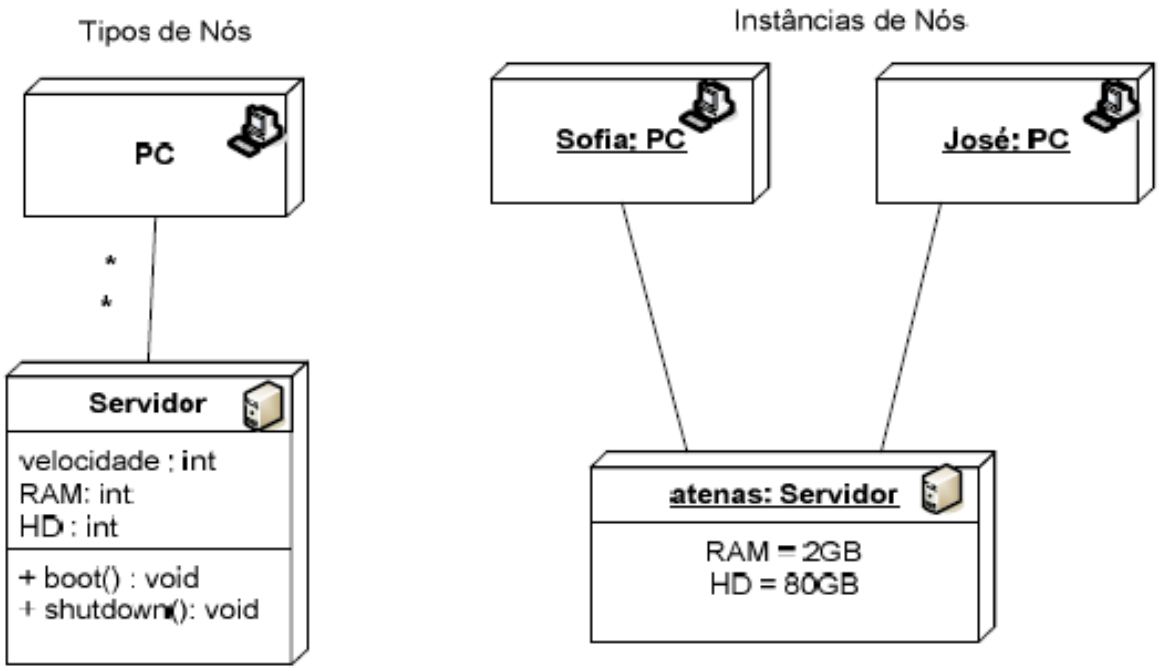


This is an example of a **layered architecture** (why?)

Deployment Diagrams

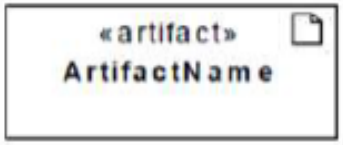
Nodes

- Nodes are computational resources where artifacts may be deployed
  - Are connected by communication associations
  - My be represented as types or instances
  - May have attributes, methods and stereotypes (as text or icon)



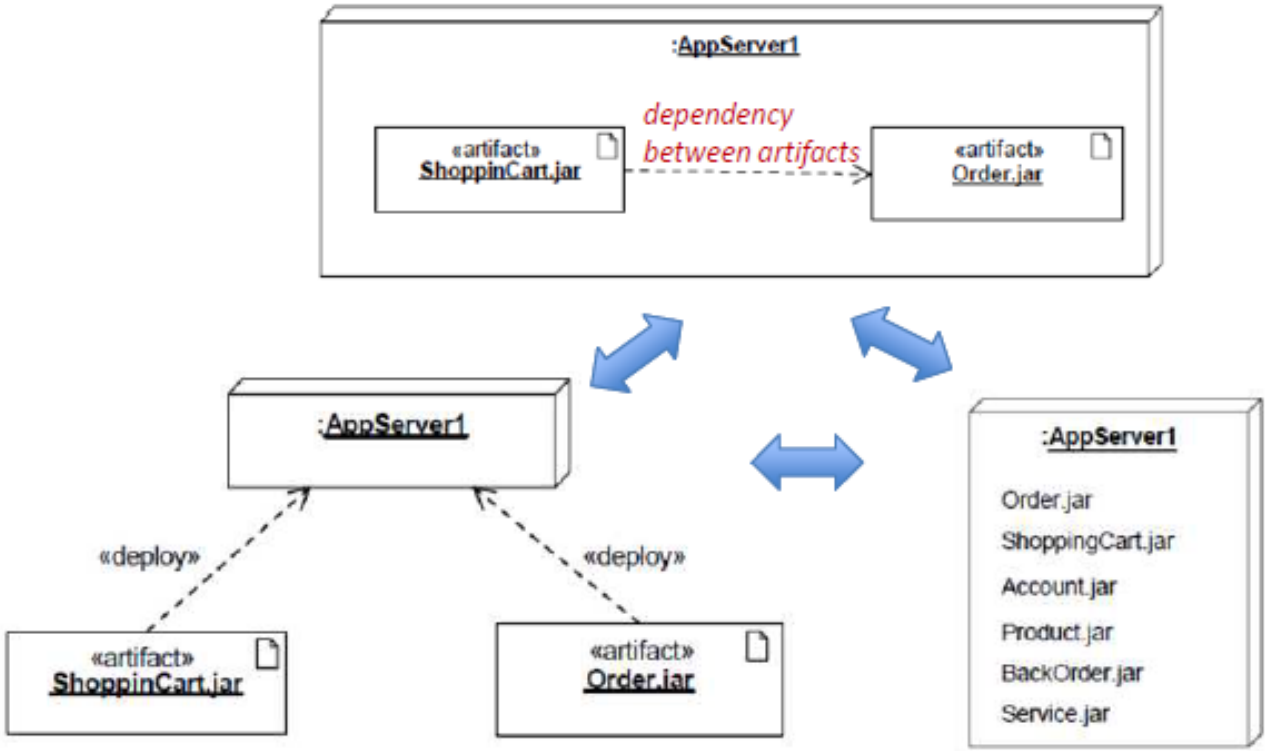
Artifacts

Artifacts are physical information elements used or produced by a software development process or by the installation or operation of a system, like for example model files, source code files, executable files, scripts, etc.

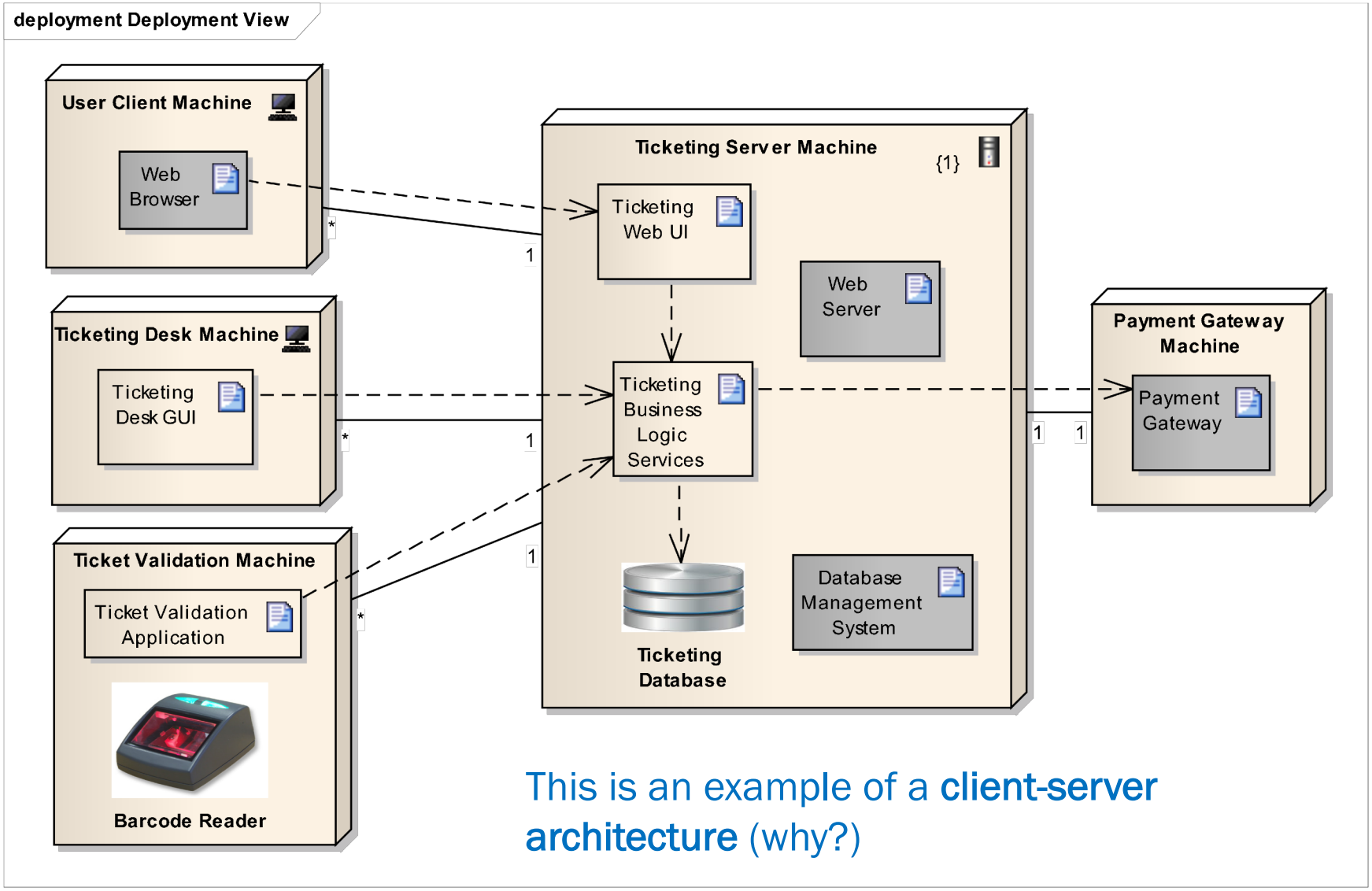


Artifacts are deployed on hardware nodes (as types or instances)

It is possible to indicate dependencies between artifacts



Example: Electronic Ticketing System

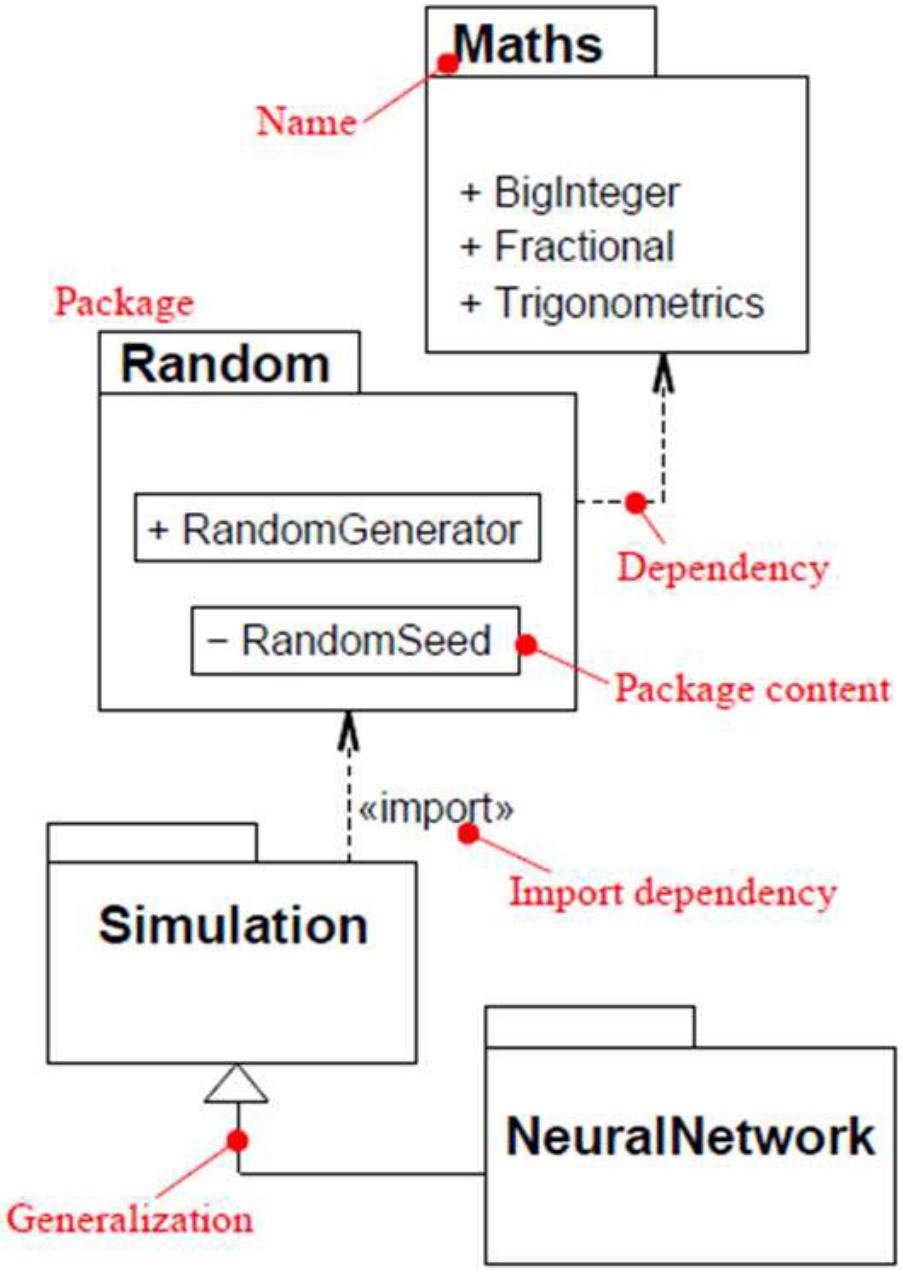


This is an example of a **client-server architecture** (why?)

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.



References and further reading

- Software Engineering, 10th edition, Ian Sommerville, chapter 6
- An Introduction to Software Architecture, David Garlan and Mary Shaw, January 1994, CMU-CS-94-166, Carnegie Mellon University
- Architectural Blueprints — The “4+1” View Model of Software Architecture, Philippe Kruchten, IEEE Software 12 (6), pp. 42-50, 1995
- Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, F. Buschmann et al, Willey, 1996
- IEEE Recommended Practice for Architectural Description of Software Intensive Systems (IEEE 1471-2000)
- OMG Unified Modeling LanguageTM (OMG UML), Superstructure: http://www.omg.org/technology/documents/modeling_spec_catalog.htm
- UML tutorials by Sparx Systems: http://sparxsystems.com/resources/uml2_tutorial/