

Software Architecture

L.EIC-ES-2021-22

Ademar Aguiar, J. Pascoal Faria

Waterfall

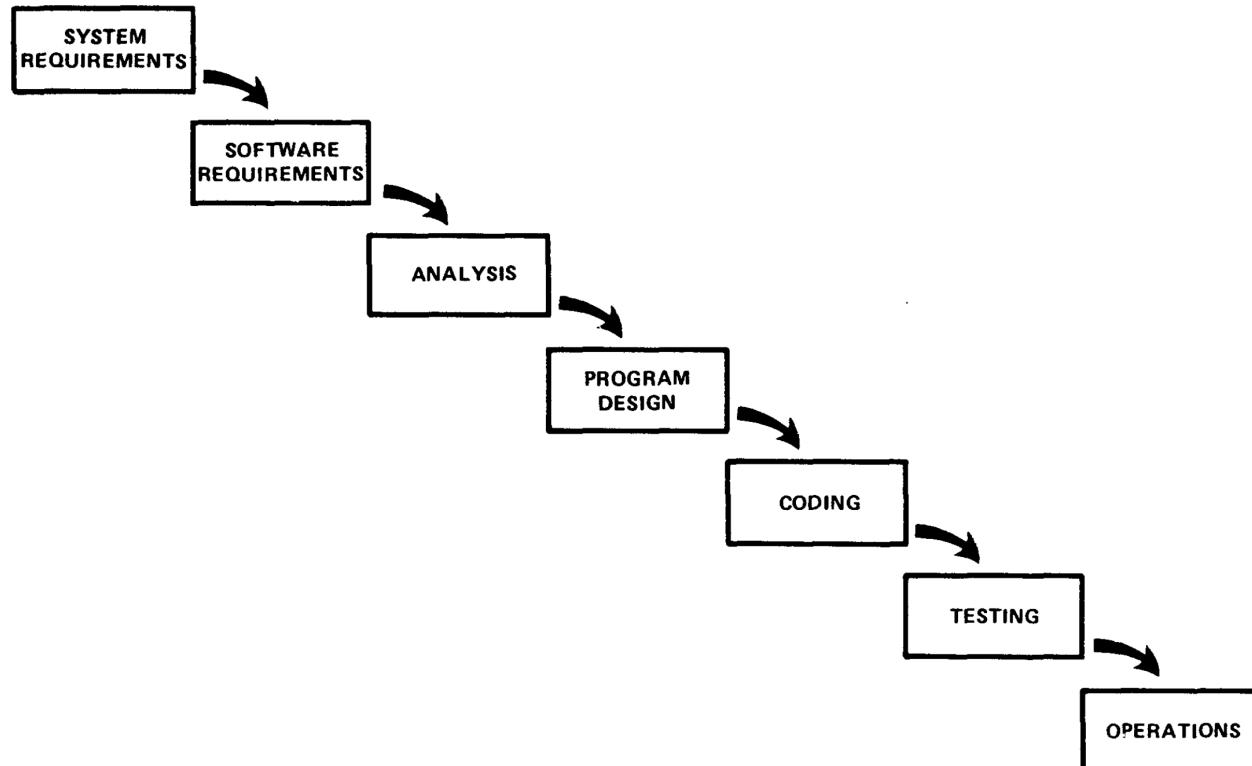
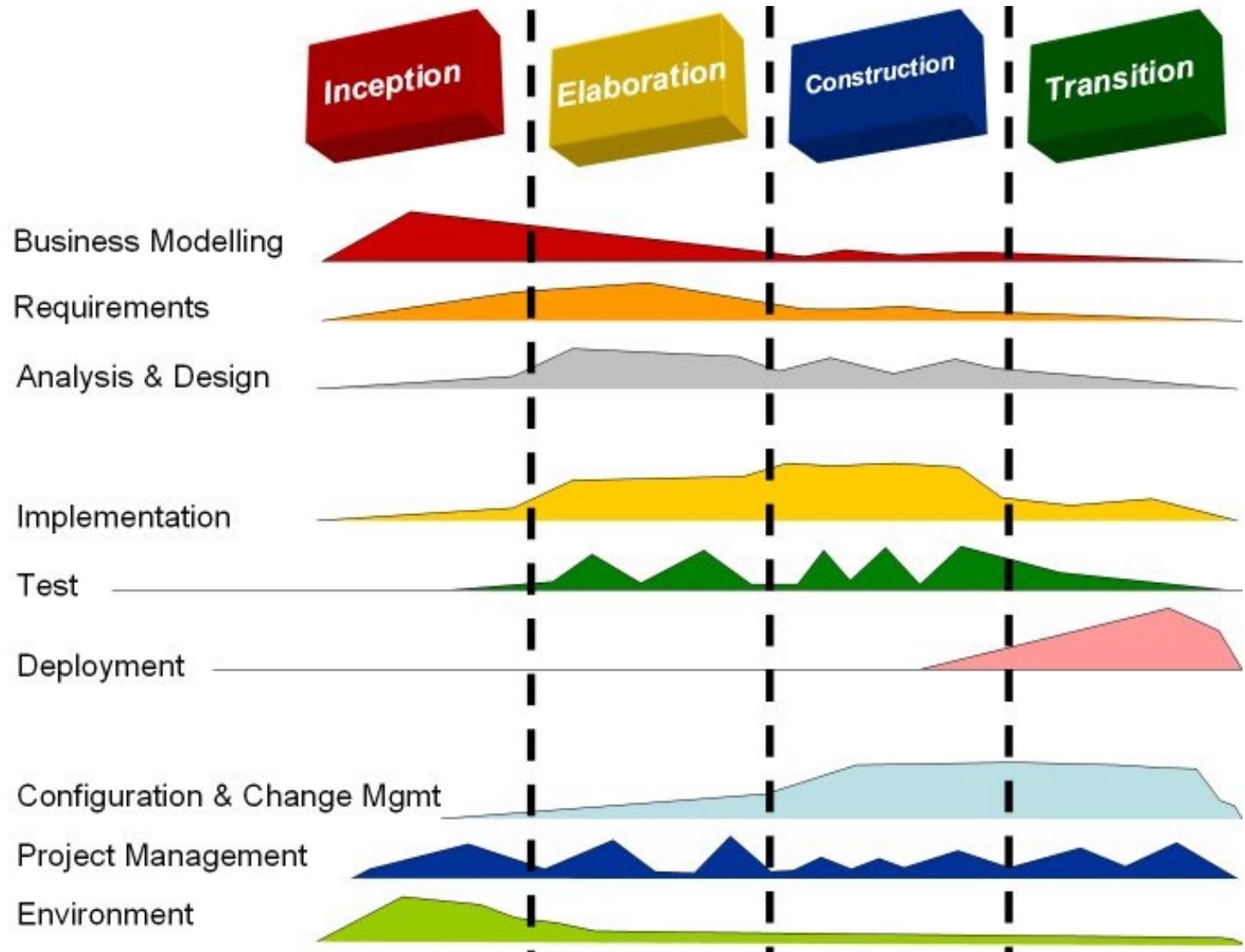


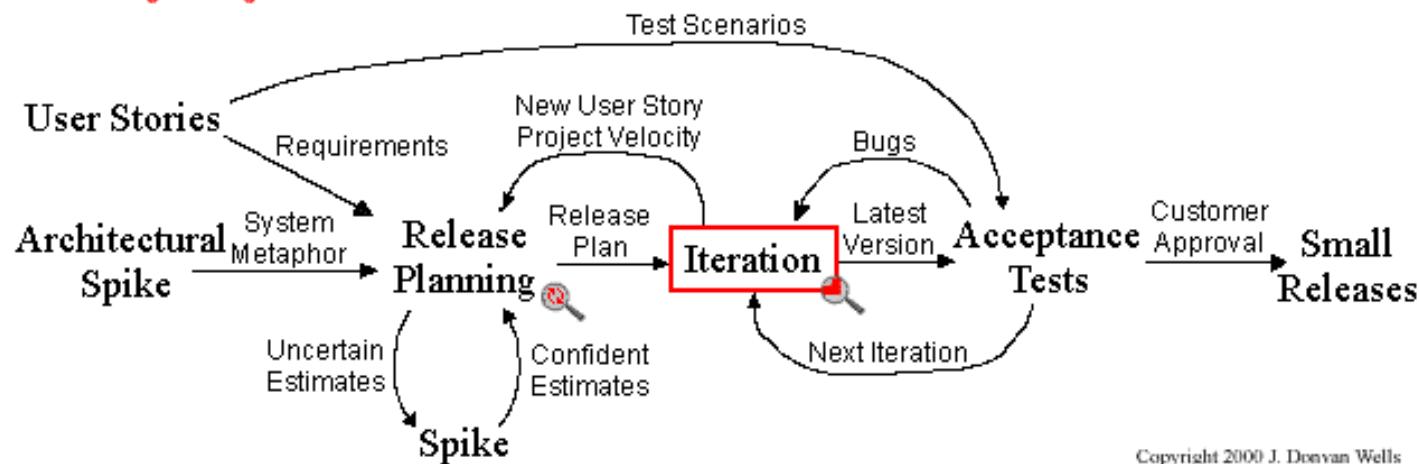
Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

Rational Unified Process



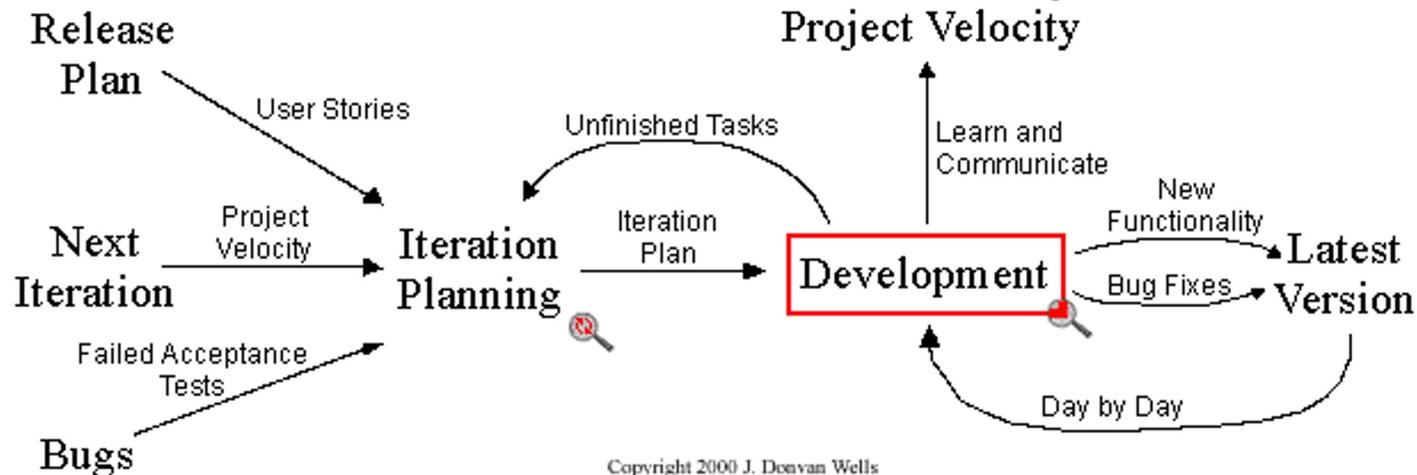


Extreme Programming Project

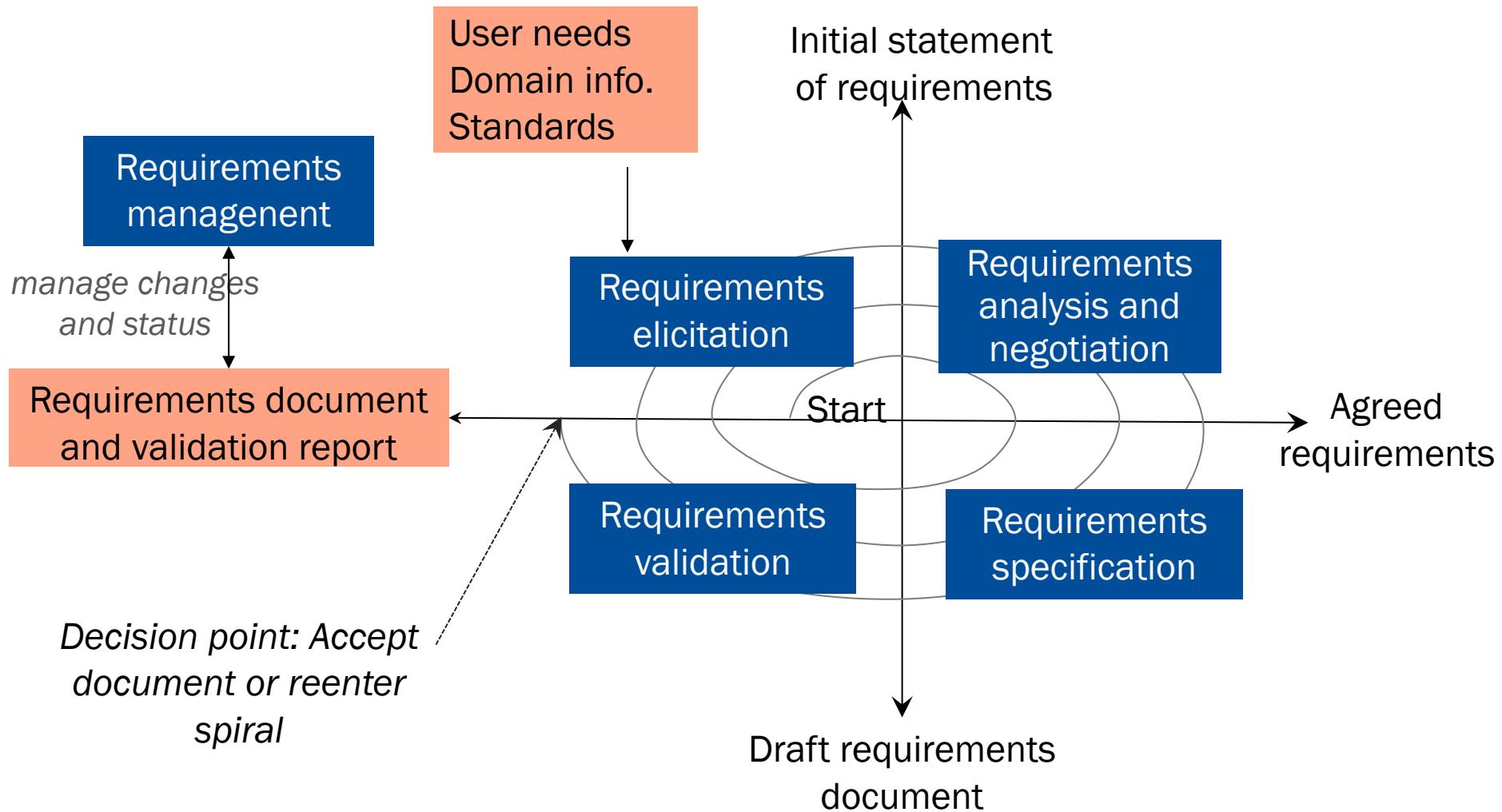


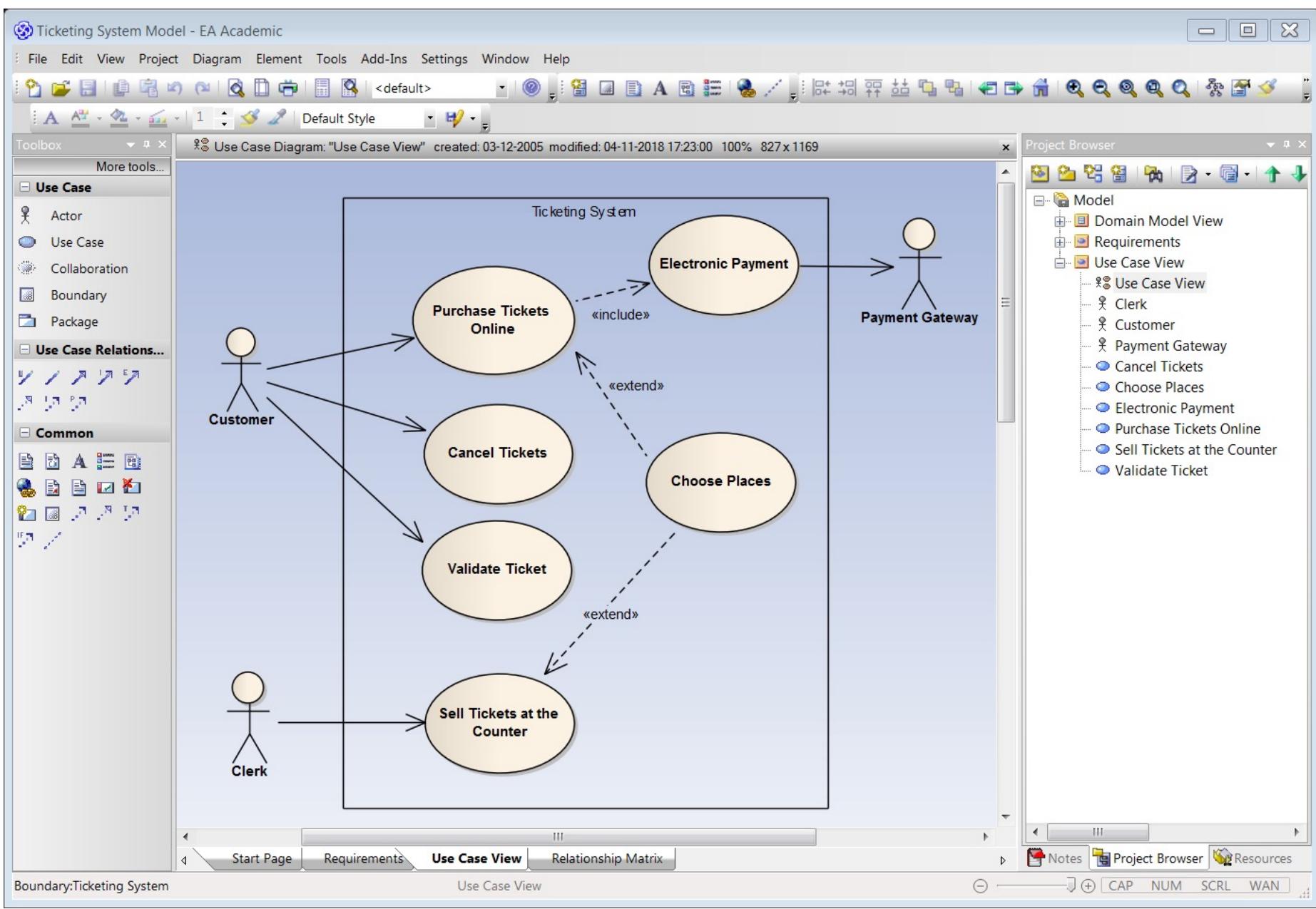
Iteration

Zoom Out



Requirements engineering process





User Stories – Example

(FRONT)

As an automobile driver, I want
to be able to remotely start my
car so that it will be warmed
up by the time I get to it.

Prototyping



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

Software Architecture

L.EIC-ES-2021-22

Ademar Aguiar, J. Pascoal Faria

Architecting...

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



wall
door



how to build them on
water?



how to layout a set of them preserving
privacy, accessibility and security?

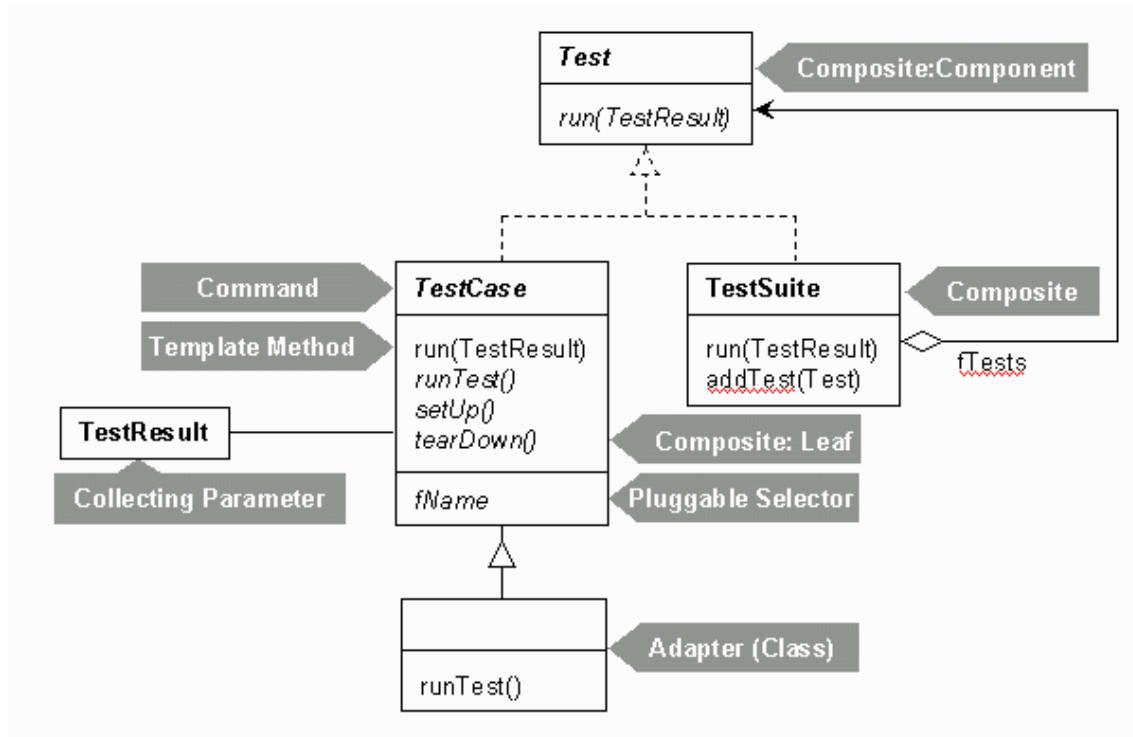


how to make them look beautiful from sky?



Architecting...

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



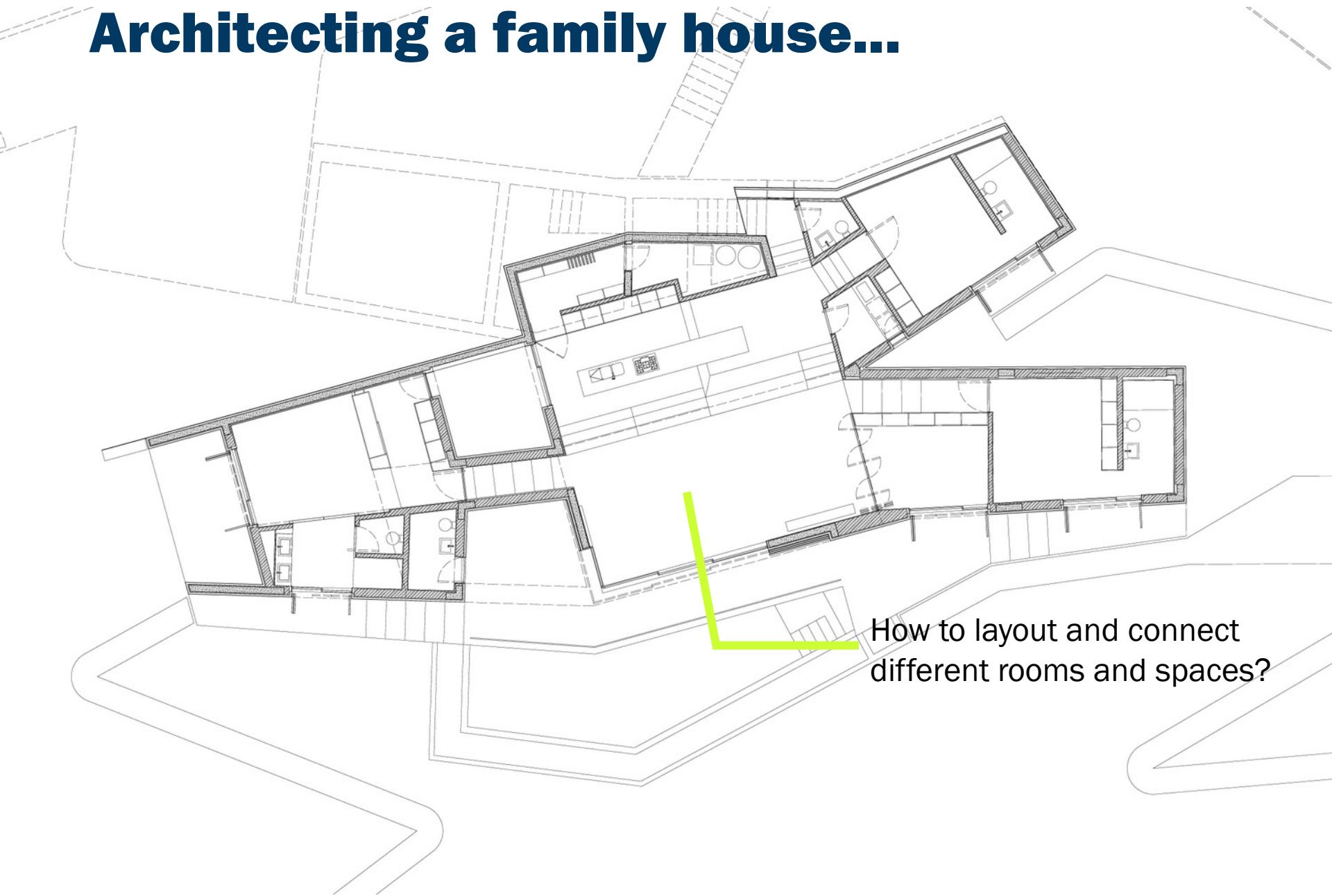
- Which building elements can we identify here? Hmm...

Architecting a family house...



How to layout and connect
different rooms and spaces?

Architecting a family house...



Architecting a multi-story building...

How to layout and
connect several houses?



Architecting a multi-story building...

How to layout and connect several houses?



...



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



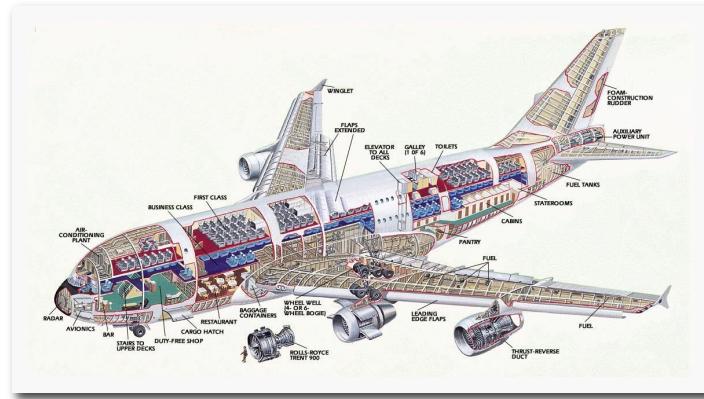
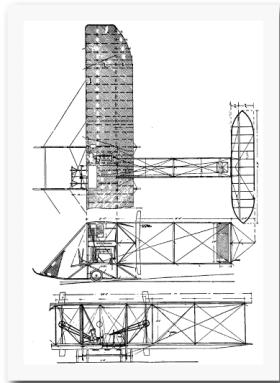
Architecting...

Main differences between all these buildings:

- Scale
- Cost, Time, Teams
- Process, skills
- 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



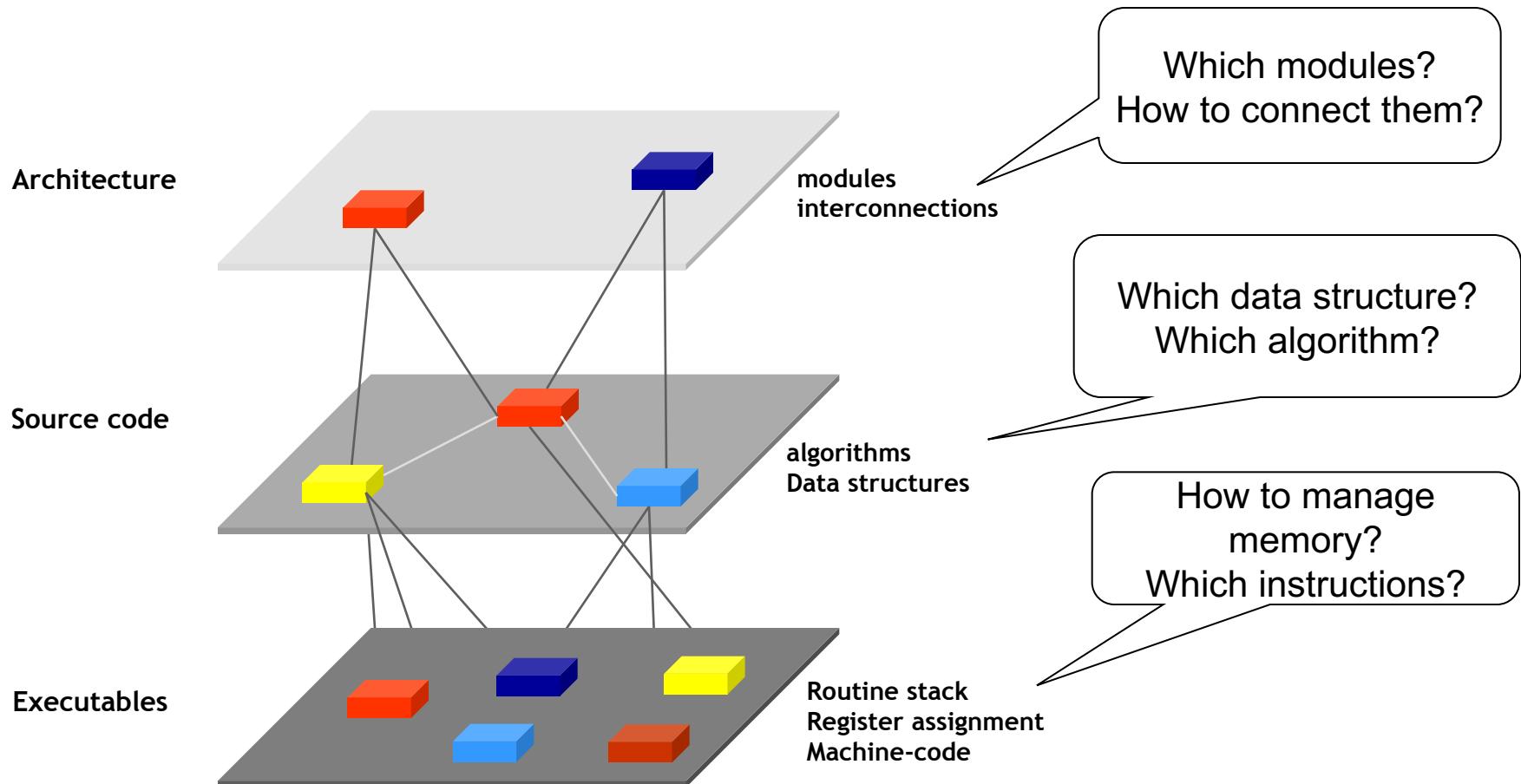
Software Construction: is it harder? Yes!

- 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 “physical 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

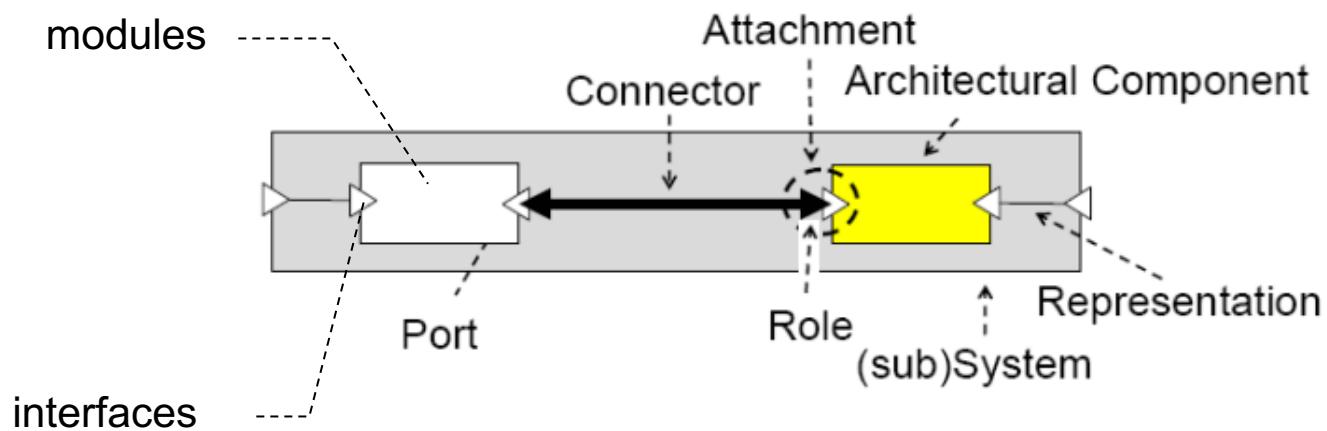


Software Architecture

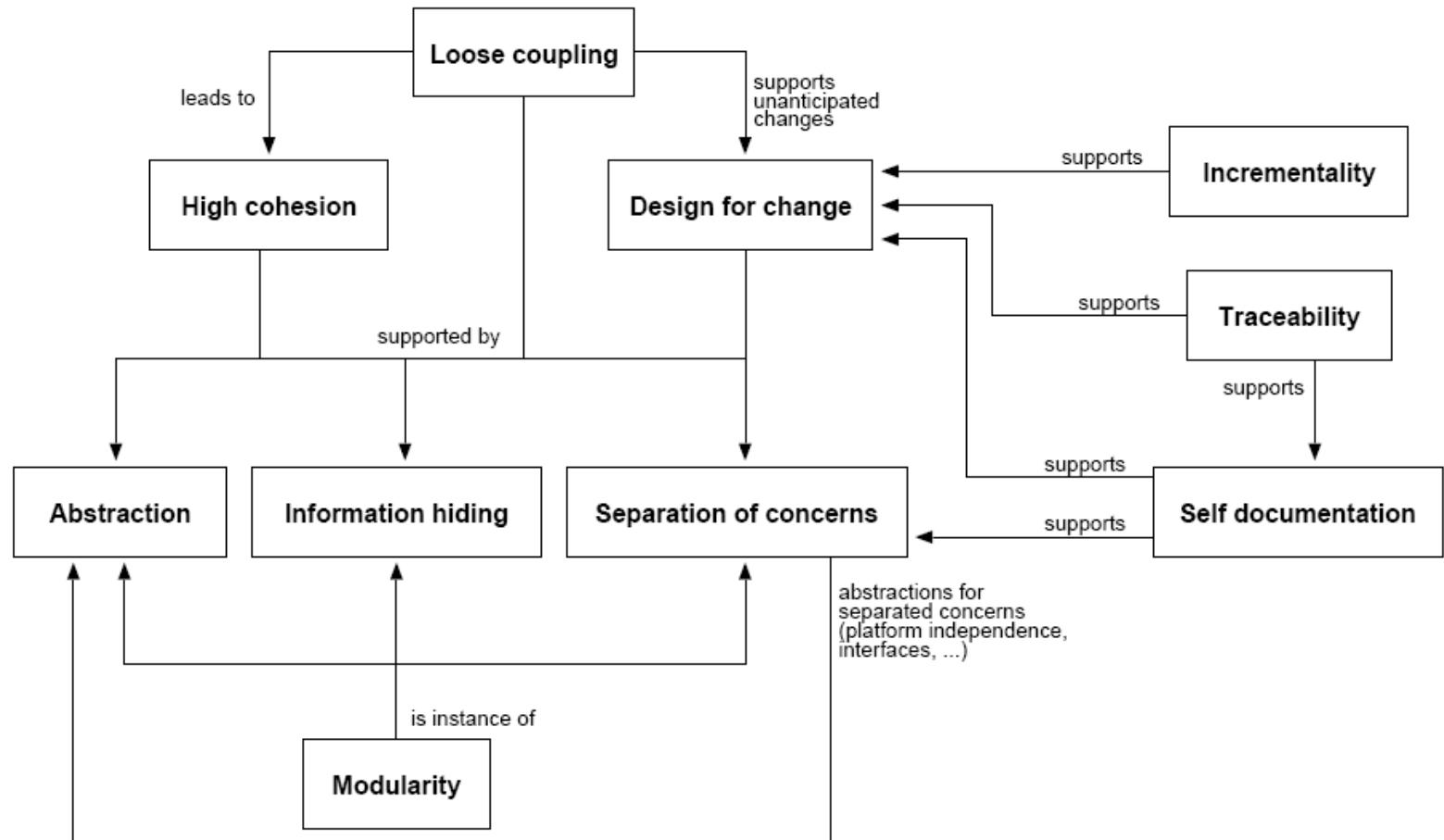
- Software Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

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 collaboration behaviors between elements?
- Which main architectural style informs the system?



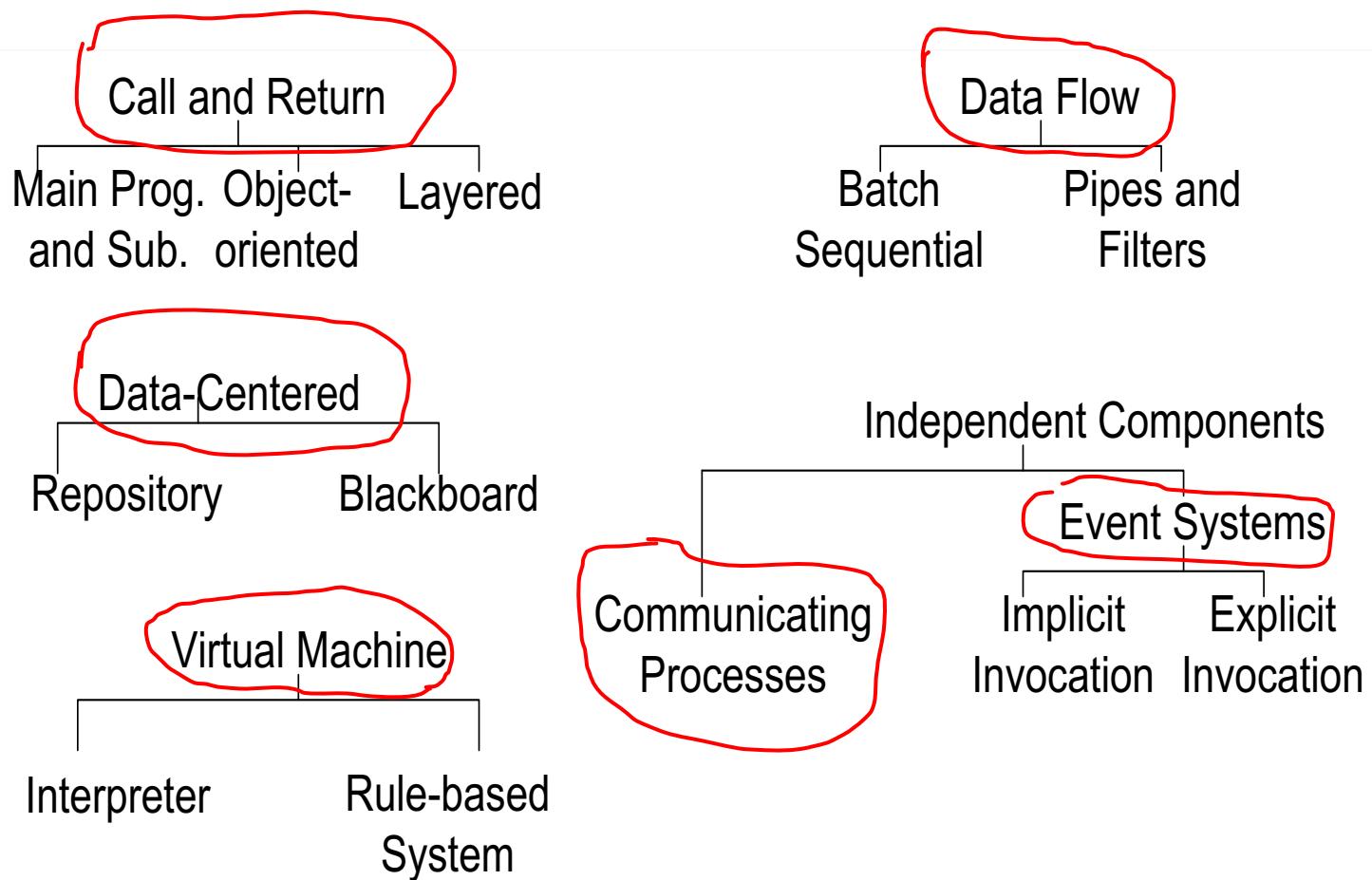
Software Architecture: key principles



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



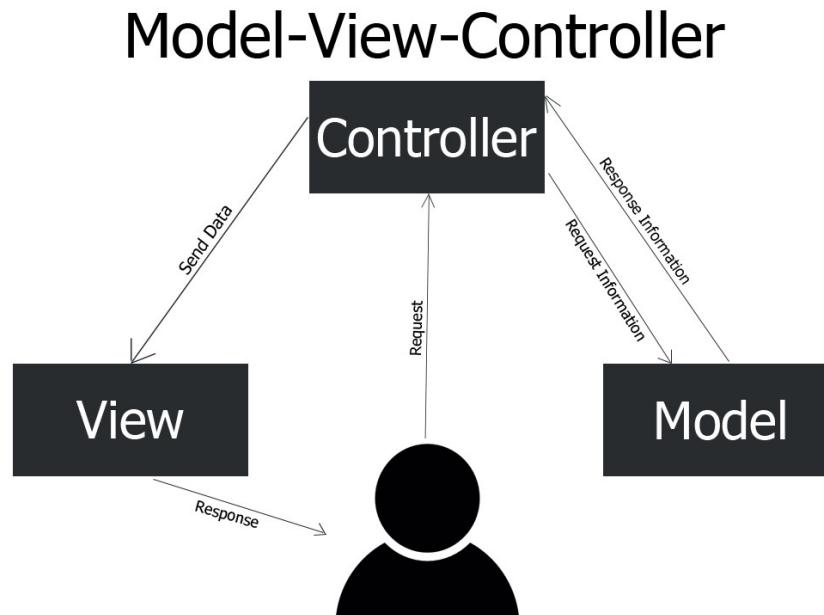
From Chapter 5, Software Architecture in Practice, p. 95

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)

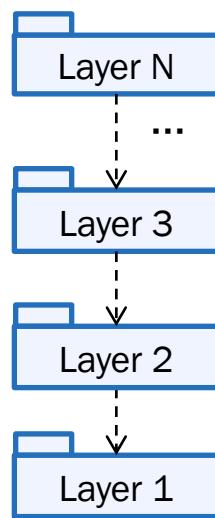
- Organizes the system as a set of data processing components (filters), connected so that data flows between components for processing (as in a pipe).
- The name comes from Unix, e.g. (shell command):

<i>filter (component)</i>	<i>pipe (connector)</i>	<i>filter</i>	<i>pipe</i>	<i>filter</i>
grep gets *.[ch]		cut -f1 -d:		sort -u

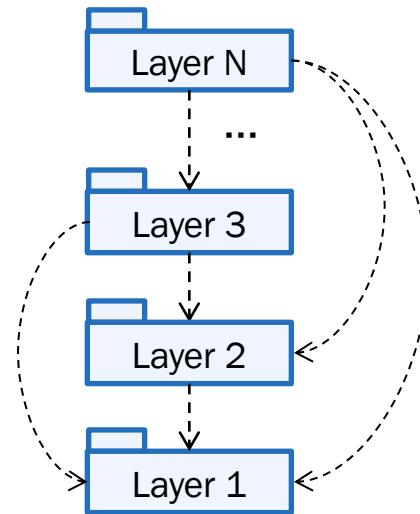
Layered Architecture

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

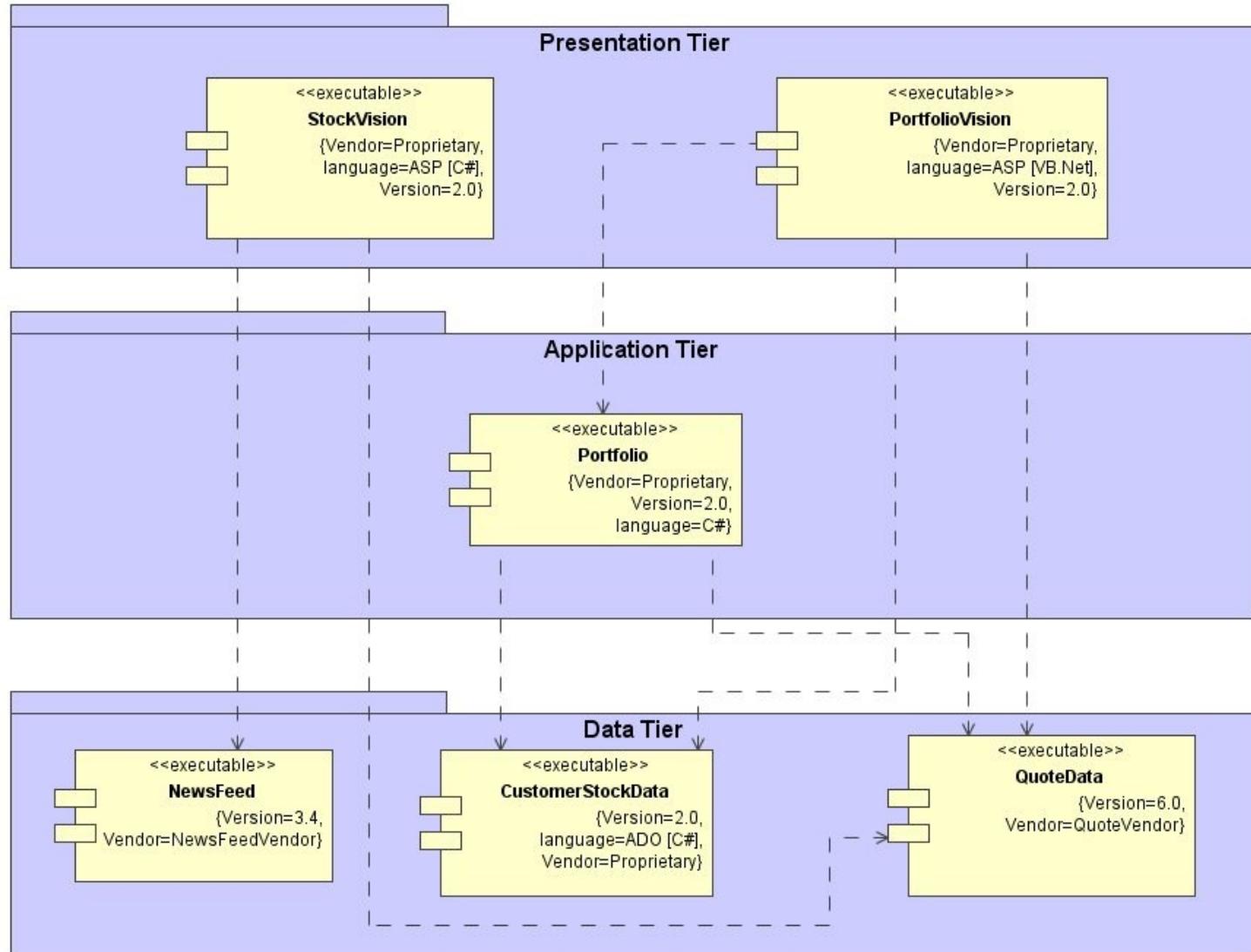
Strict: each layer can only interact with the layer directly below (default).



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

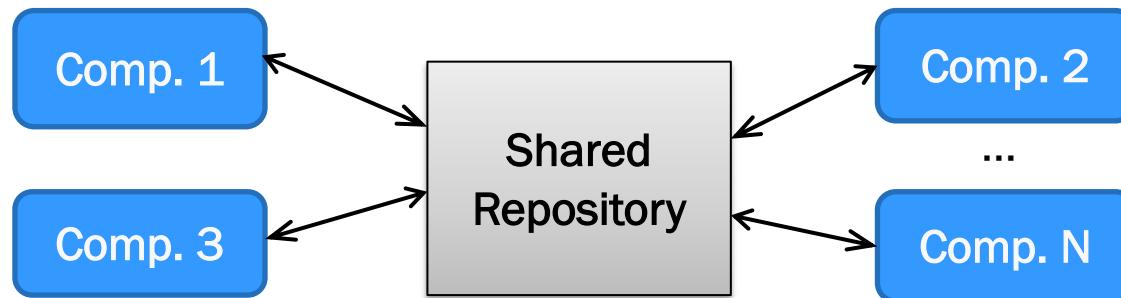


Example: Three-Layered Services Application



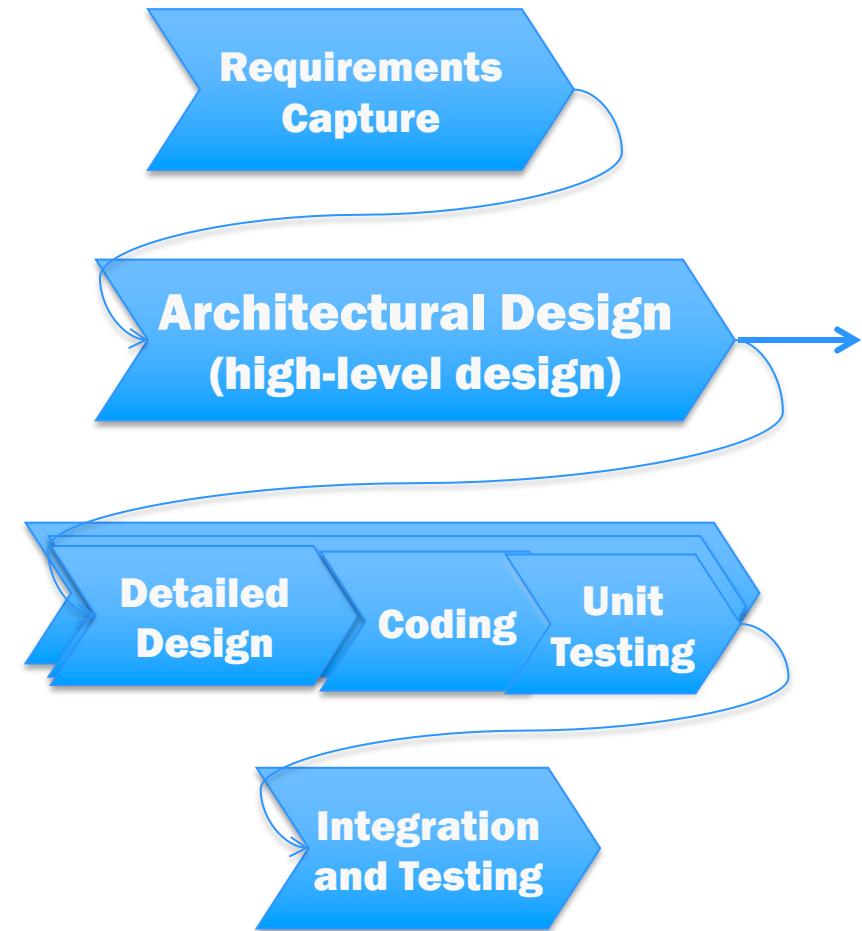
Repository Architecture

- 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.



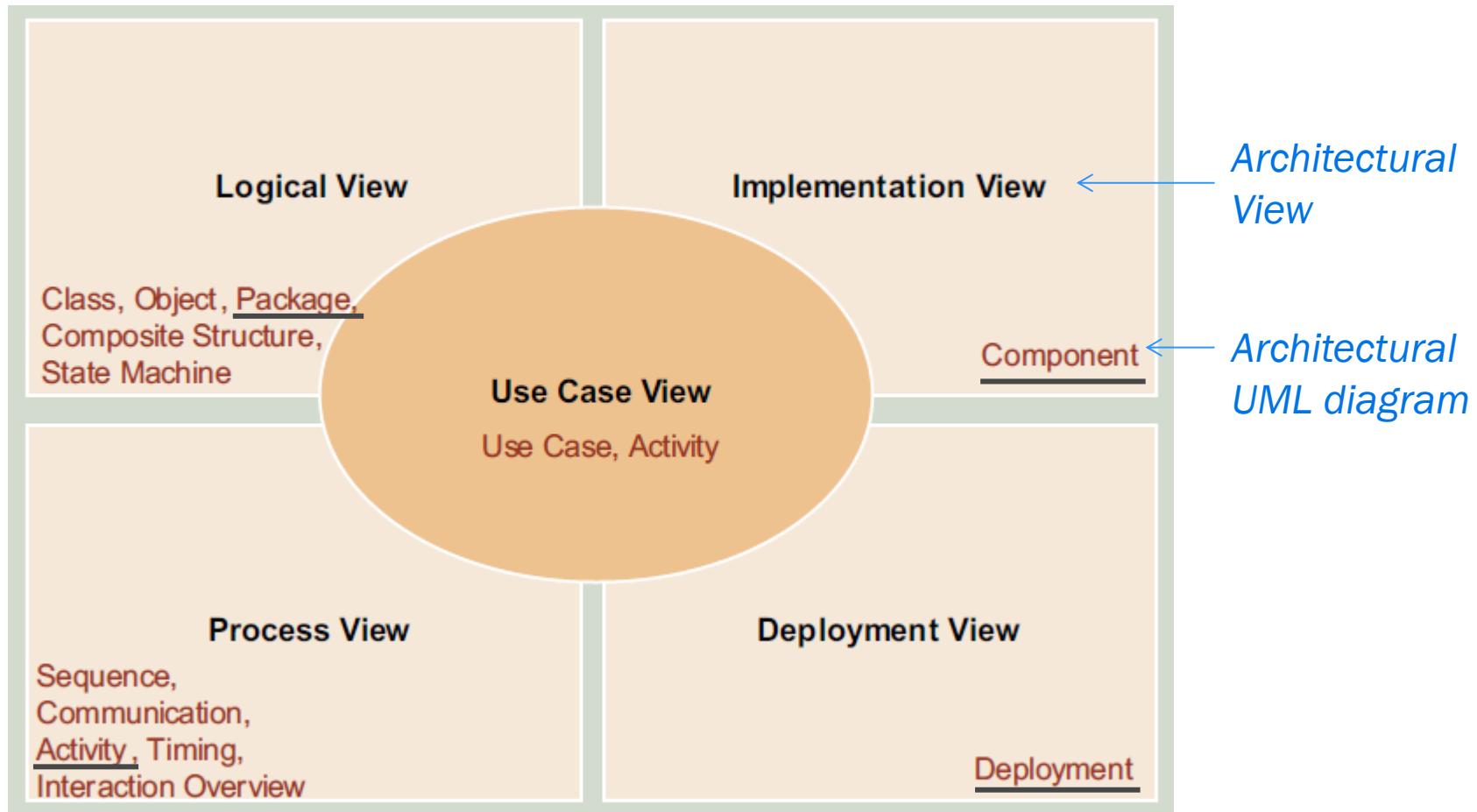
- Variants:
 - **Passive** repository: passively accepts requests from components
 - **Active** repository: changes to the repository trigger component execution

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

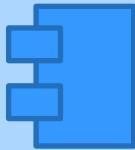
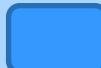


Source: Architectural Blueprints – The “4+1” View Model of Software Architecture,
Philippe Kruchten, Philippe, IEEE Software 12 (6), pp. 42-50, 1995

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.

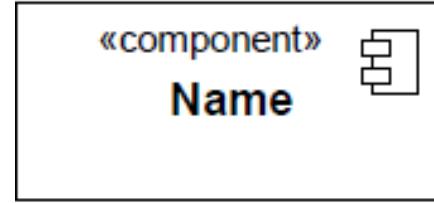
Architectural views and UML diagrams

<p>Logical View: 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>Implementation View: Component diagrams</p> <p>Shows software components and dependencies among them</p>	
<p>Process View: Activity diagrams (and others)</p> <p>Shows processing steps, data/object stores, data/object-flows, and opportunities for parallelization</p>		<p>Deployment View: Deployment diagrams</p> <p>Shows hardware nodes, communication relationships and software artifacts deployed on them</p>	

Component Diagrams

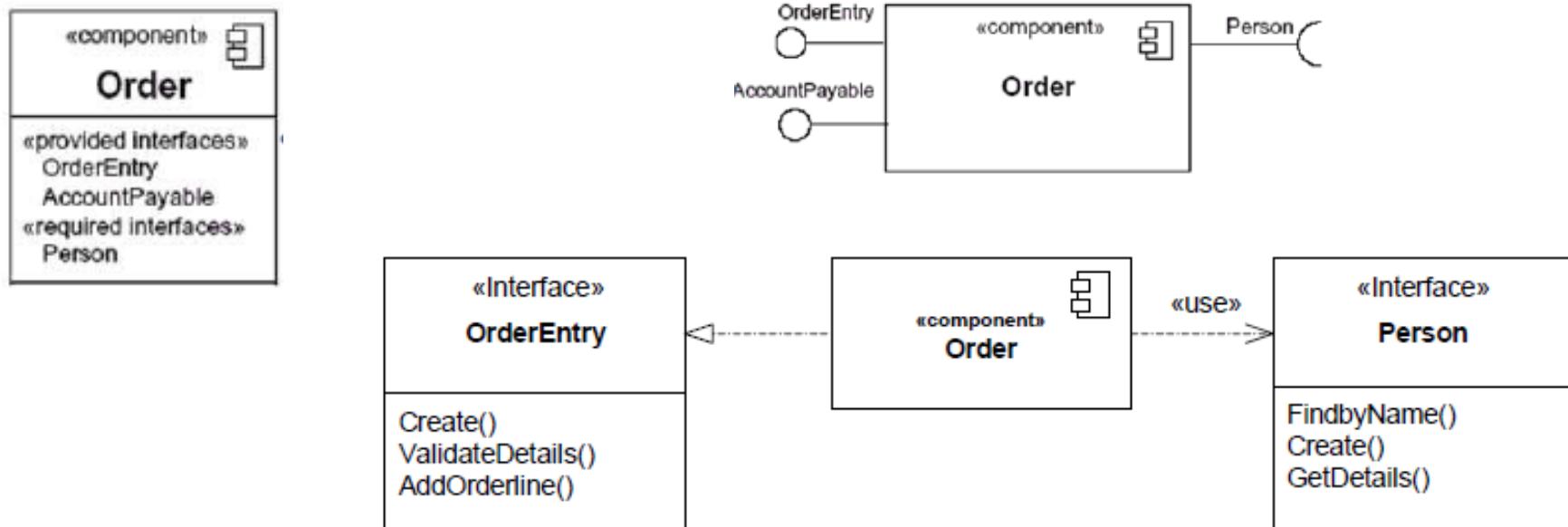
Components

- 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 independently
- Components are the central concept in **Component-based Software Engineering** (CBSE) as units of **reuse**
- Equivalent notations:



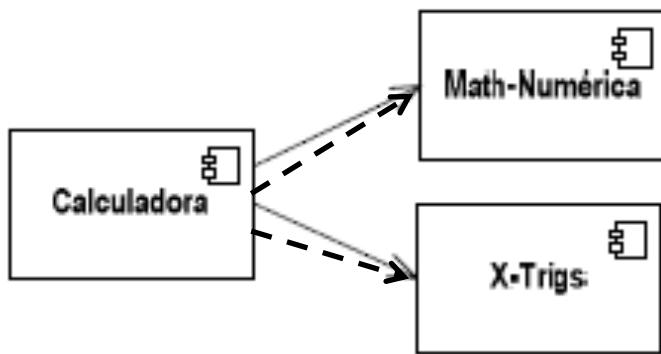
Interfaces

- 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:

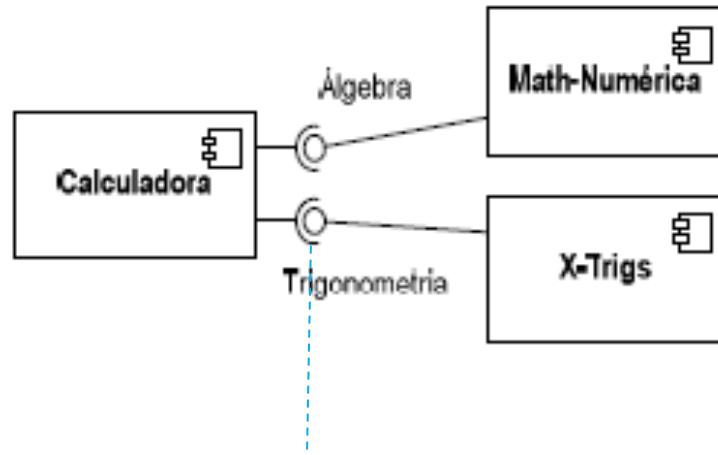


Dependencies

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



(simplified view,
omitting interfaces)

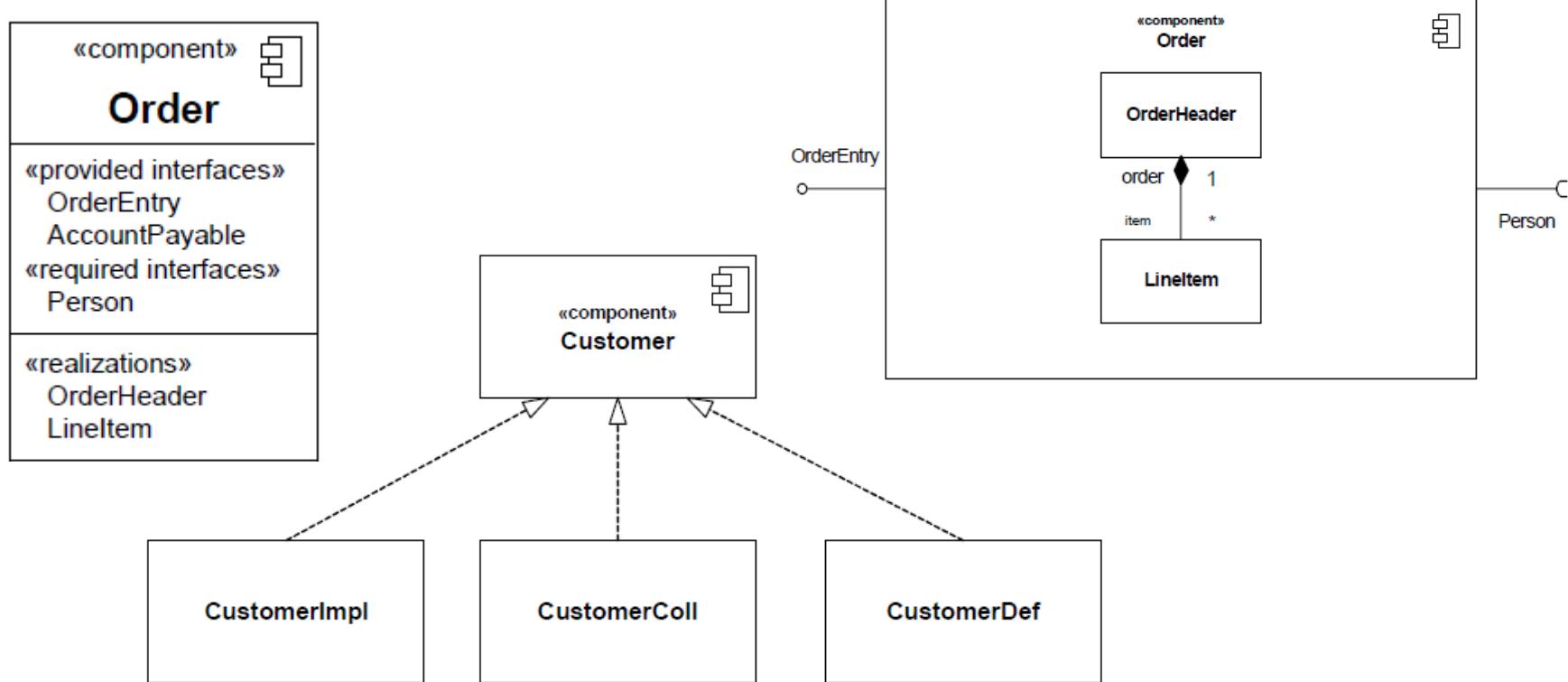


assembly connector, shorthand for:



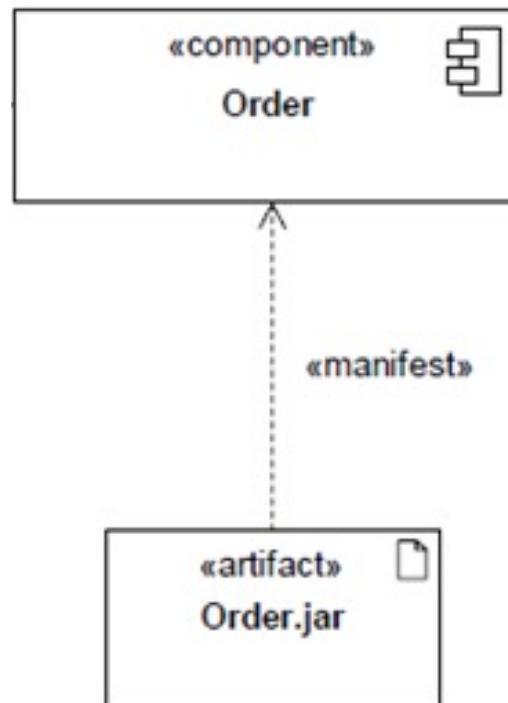
Components and classes

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

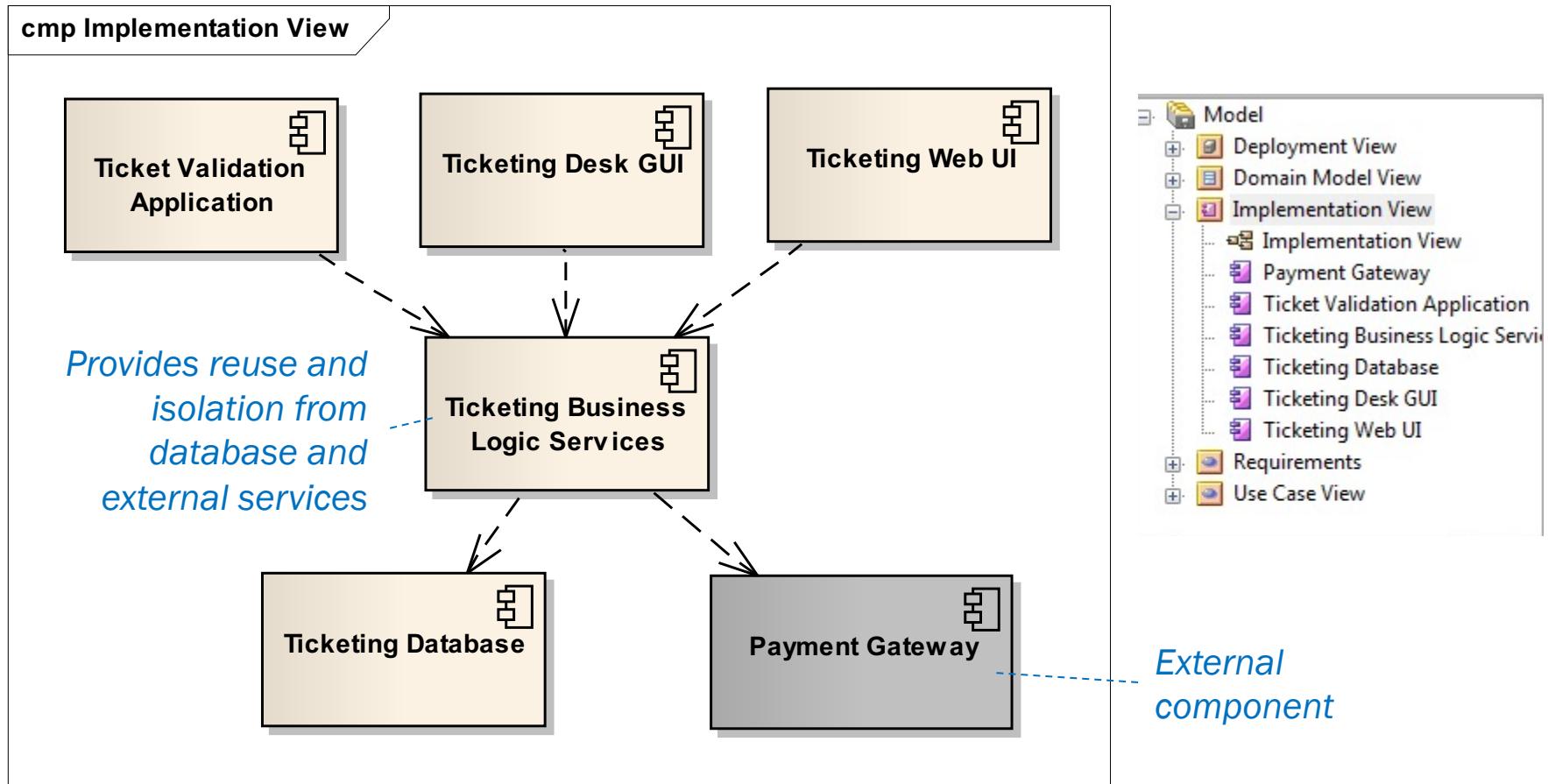


Components and artifacts

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



Example: Electronic Ticketing System

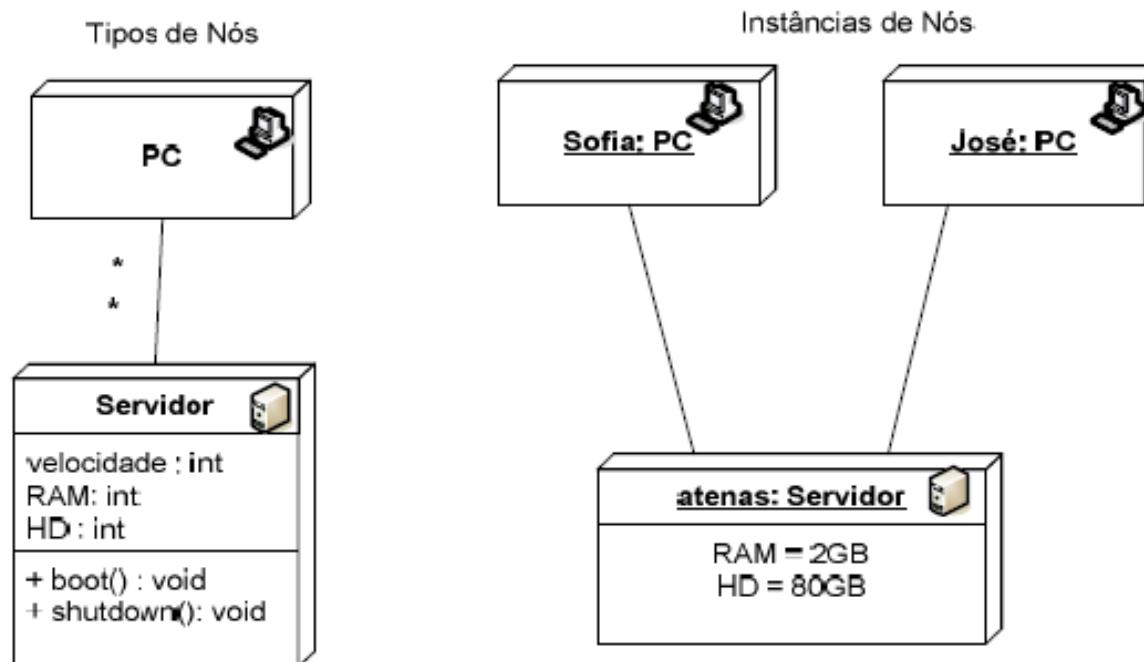


This is an example of a layered architecture (why?)

Deployment Diagrams

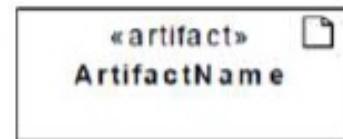
Nodes

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



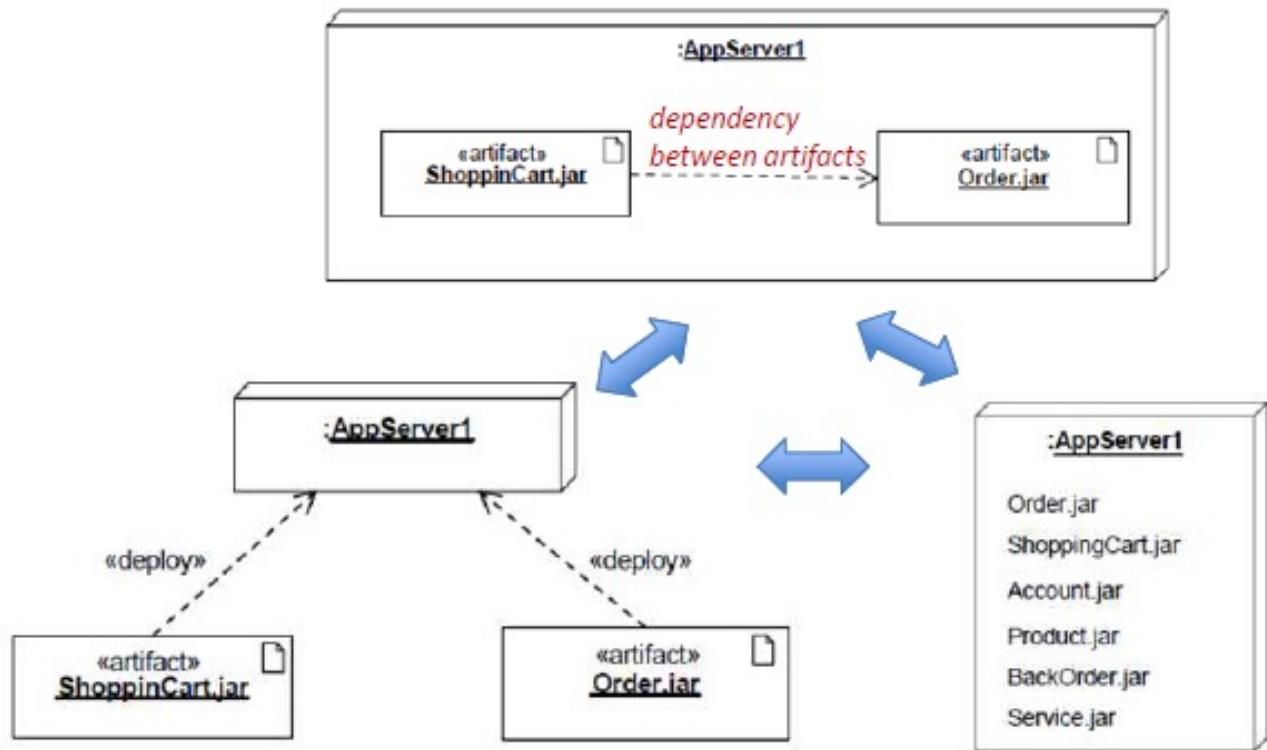
Artifacts

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.

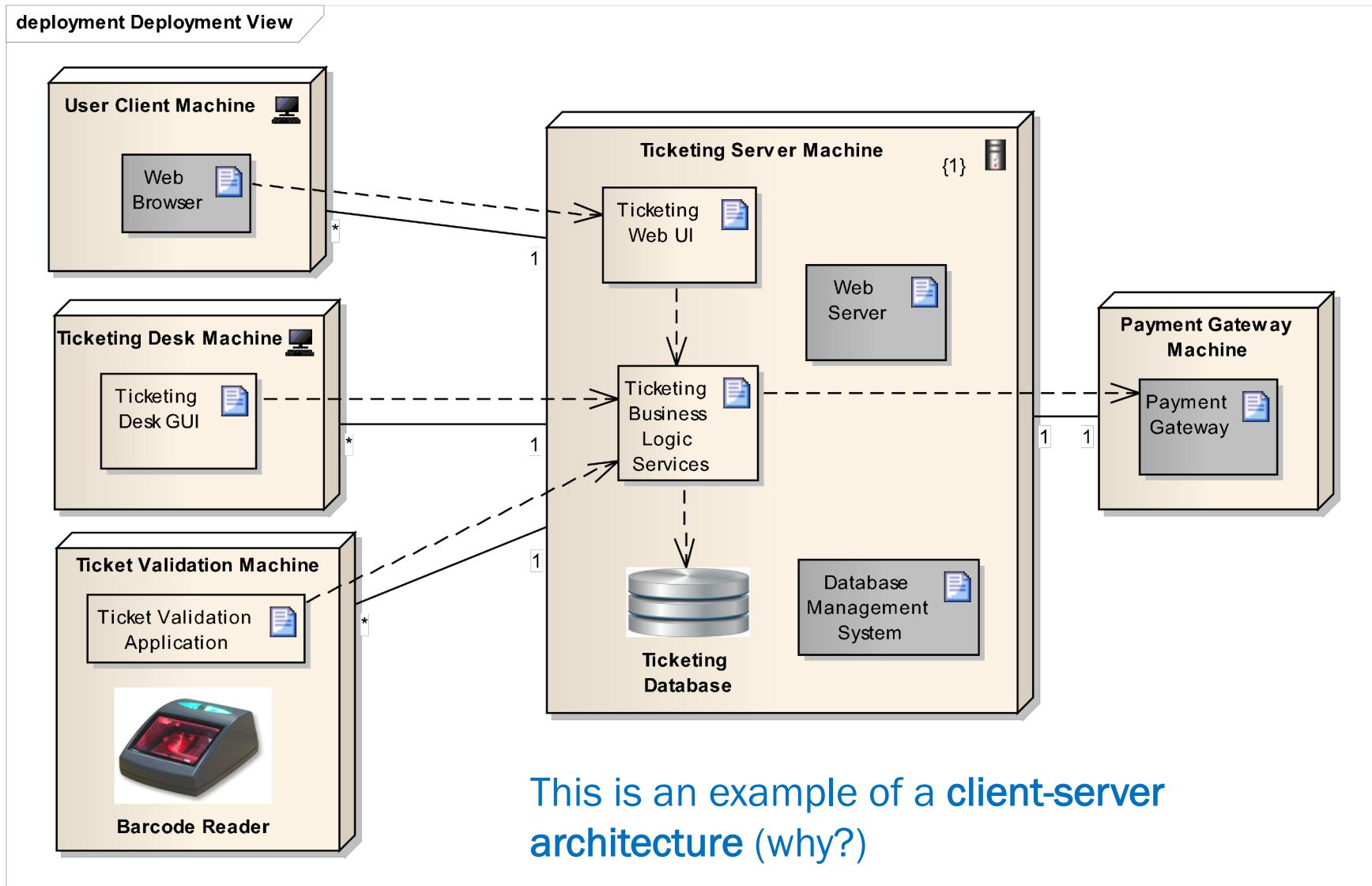


Artifacts are deployed on hardware nodes (as types or instances)

It is possible to indicate dependencies between artifacts



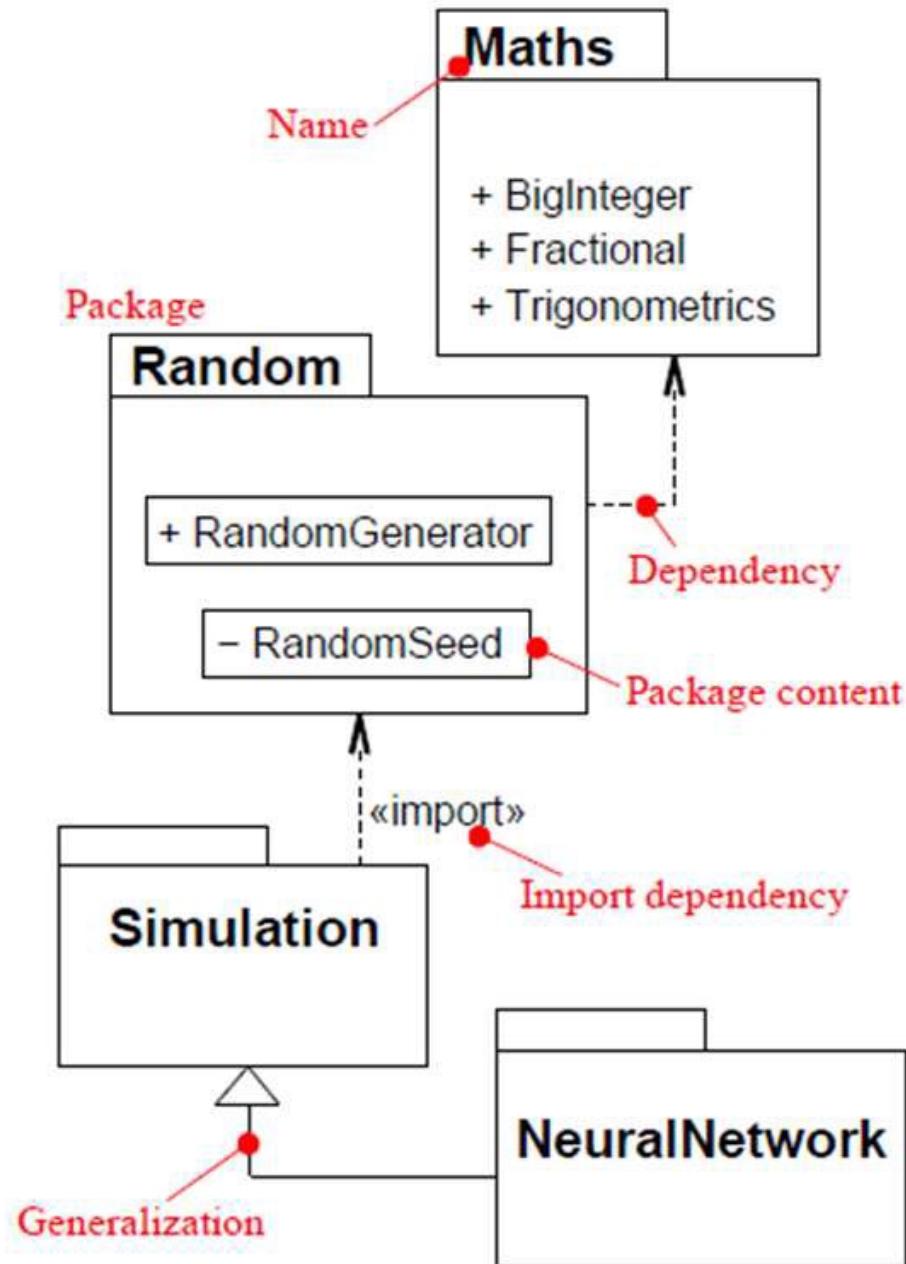
Example: Electronic Ticketing System



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, Wiley, 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/

UNI4ALL

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

