ENPM 613: SOFTWARE DESIGN AND IMPLEMENTATION

GENERATE ARCHITECTURE: METHODS, STYLES, TACTICS

Dr. Tony D. Barber / Prof. Sahana Kini

Fall, 2022

TODAY'S CLASS OBJECTIVES AND OUTCOME

- Understand how to represent and document software architecture
- Explain different architecture representations
- Define what are architecture views and what is the benefit of using them for documenting architecture
- Compare and contrast different notations for architecture documentation
- Read and develop architecture views using appropriate UML diagrams

OUTLINE

- Lecture
  - Documenting architecture
    - Architecture Views
    - Notations for documenting architecture
    - Documenting components and interfaces
    - Design decisions and their documentation
- Assignments

RESOURCES

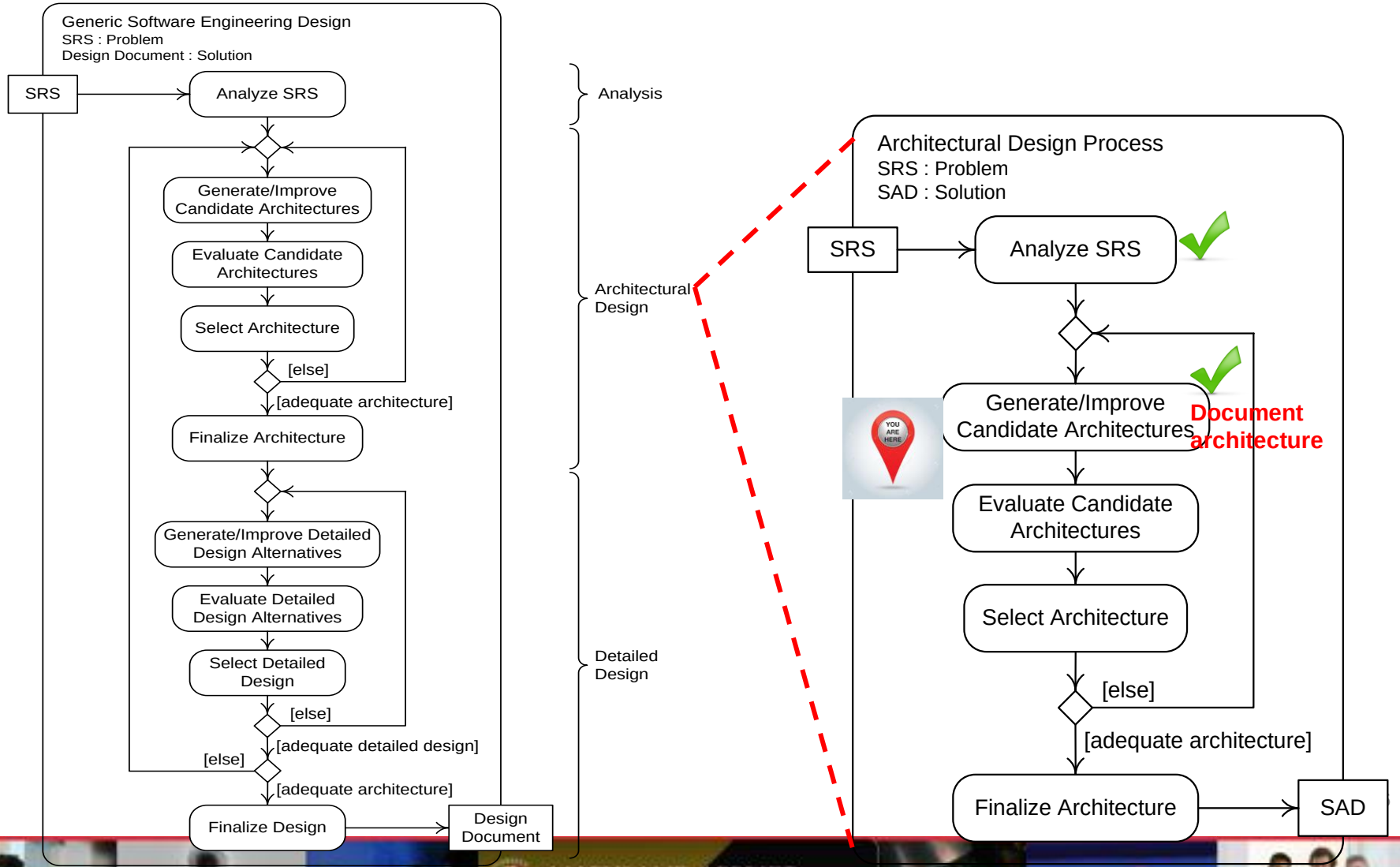
- **Books**
  - Textbook Chapters 9 and 10 (and 12)
  - *Documenting Software Architectures: Views and Beyond*, by Paul Clements et all
  - *Software Architecture in Practice, (SEI Series in Software Engineering),* by Len Bass and Paul Clements
  - *Software Modeling and Design - UML, Use Cases, Patterns, and Software Architectures*, By Hassan Gomaa
    - **Good examples/case studies!**
- **Paper**
  - *Architectural Blueprints—The "4+1" View Model of Software Architecture*, by Philippe Kruchten, IEEE Software 12 (6) November 1995
- **Standard -** *ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description* http://www.iso-architecture.org/ieee-1471/
- SEI web site: http://www.sei.cmu.edu/architecture/

OUTLINE

- Lecture
  - Documenting architecture
    - Architecture Views
    - Notations for documenting architecture
    - Documenting components and interfaces
    - Design decisions and their documentation
- Assignments

We are here

DESIGN PROCESS – WHERE ARE WE?



ARCHITECTURAL DESIGN sPECIFICATION CONTAINS:

- **Decomposition -** Program *parts* or modules (also called components or elements) and the *decomposition* relation between them
- **Responsibilities -** *Data* and *behavior*
- **Interfaces -** An interface is a *boundary across which entities communicate*
- **Collaborations -** Who does what when (behavior)
- **Other relationships -** Uses, dependencies, generalization, etc.
- **Properties -** Performance, reliability, etc.
- **States and Transitions**

[From: https://users.cs.jmu.edu/foxcj/Public/ISED/index.htm Ch8]

SOFTWARE ARCHITECTURE DOCUMENTATION

- How do we represent the architecture?
- What representations and notations do we use?

OUTLINE

- Lecture
  - Documenting architecture
    - Architecture Views
      - Notations for documenting architecture
      - Documenting components and interfaces
      - Design decisions and their documentation
- Assignments

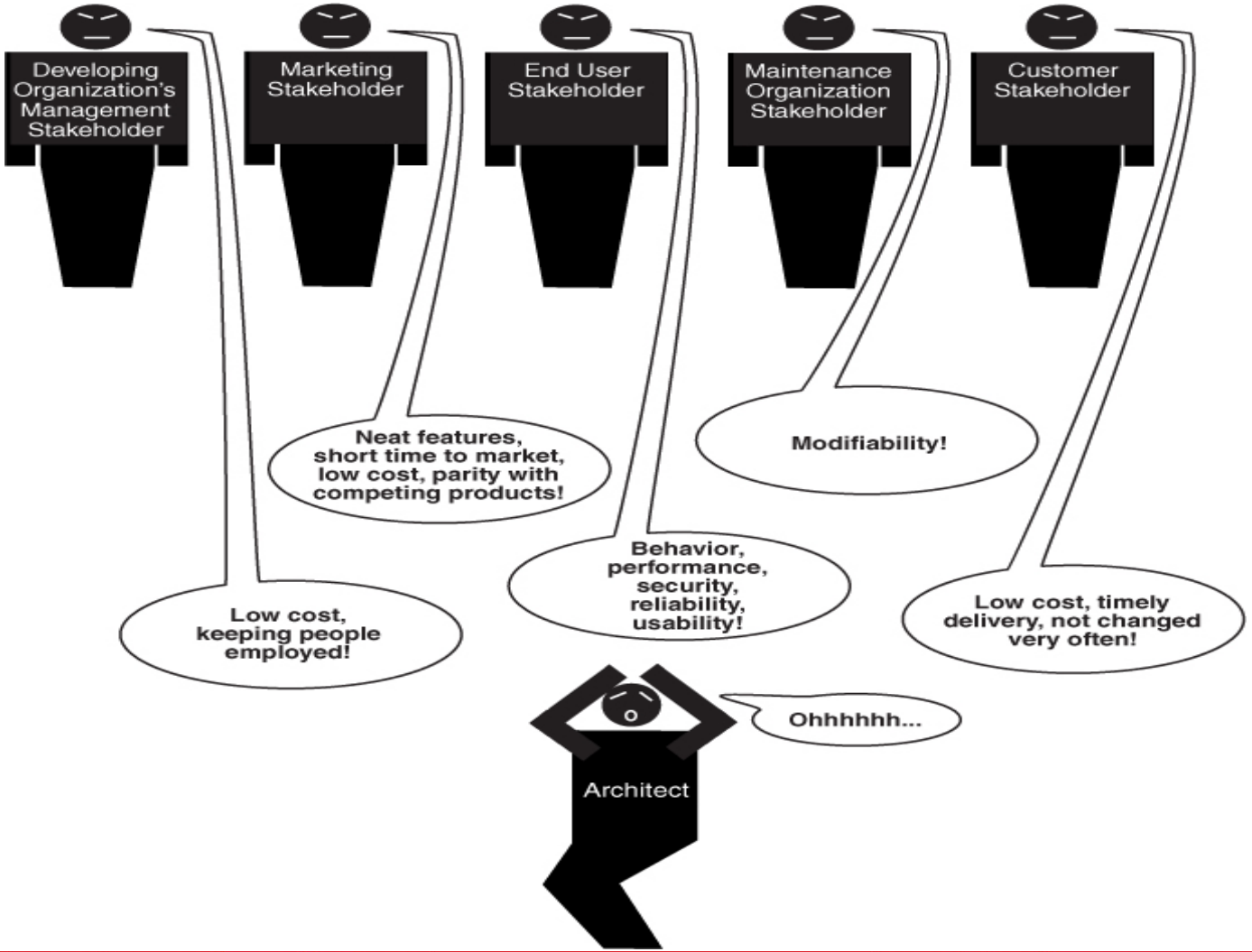
We are here

DESCRIBING SOFTWARE ARCHITECTURE USING VIEWS

- Views are used to describe Software Architecture
- Each view address one *concern*, for example:
  - Structural view shows the decomposition of system
  - Behavioral view shows how components interact at run-time
  - Deployment view shows how components are assigned to hardware elements

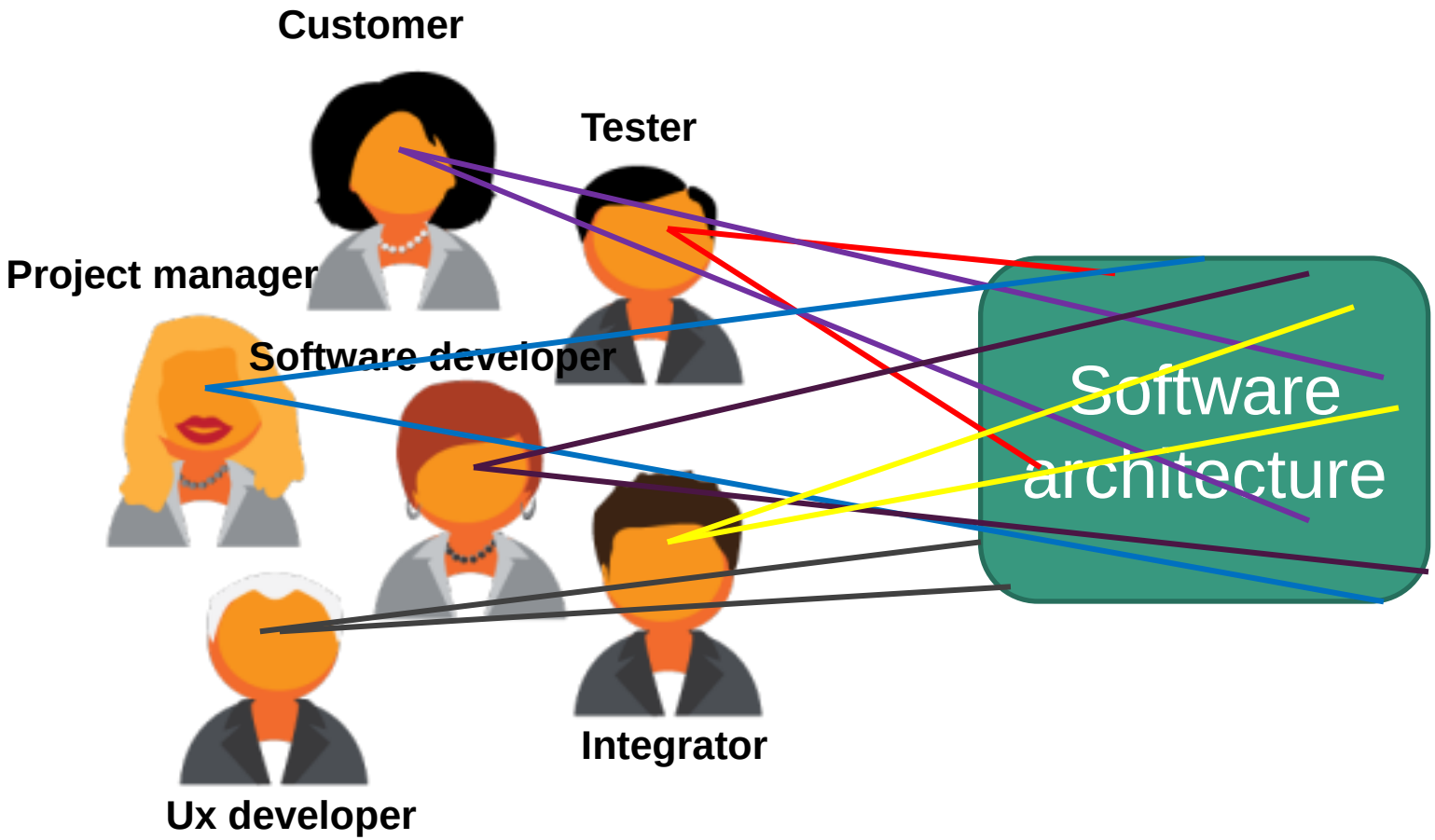
- One or many *concerns* belong to one or more *stakeholders*

SOFTWARE STAKEHOLDERS



[From: *Software Architecture in Practice*, by Len Bass and Paul Clements]

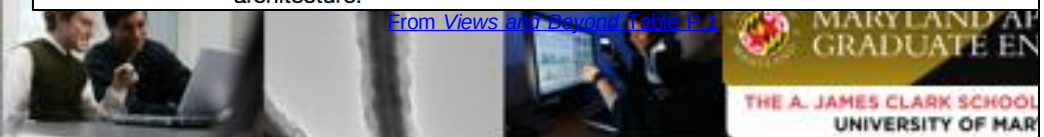
SOFTWARE ARCHITECTURE STAKEHOLDERS AND VIEWS



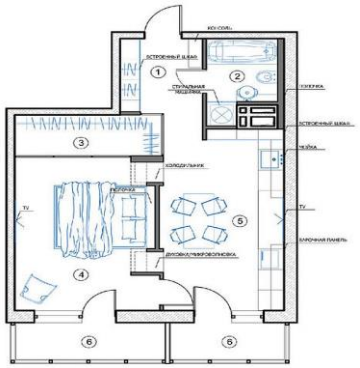
ARCHITECTURE STAKEHOLDERS' CONCERNS

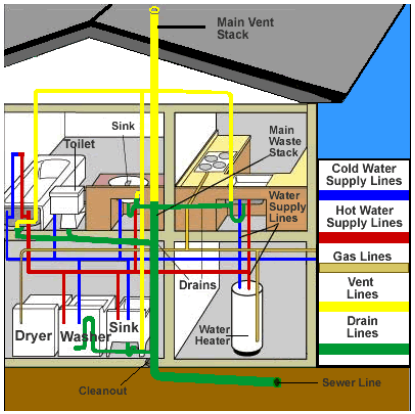
Name	Description	Use for Architecture Documentation
Architect	Responsible for the development of the architecture and its documentation. Focus and responsibility is on the system.	Negotiating and making trade-offs among competing requirements and design approaches. A vessel for recording design decisions. Providing evidence that the architecture satisfies its requirements.
Business manager	Responsible for the functioning of the business/organizational entity that owns the system. Includes managerial/executive responsibility, responsibility for defining business processes, and more.	Understanding the ability of the architecture to meet business goals.
Conformance checker	Responsible for assuring conformance to standards and processes to provide confidence in a product's suitability.	Basis for conformance checking, for assurance that implementations have been faithful to the architectural prescriptions.
Customer	Pays for the system and ensures its delivery. The customer often speaks for or represents the end user, especially in a government acquisition context.	Assuring required functionality and quality will be delivered, gauging progress, estimating cost, and setting expectations for what will be delivered, when, and for how much.
Database administrator	Involved in many aspects of the data stores, including database design, data analysis, data modeling and optimization, installation of database software, and monitoring and administration of database security.	Understanding how data is created, used, and updated by other architectural elements, and what properties the data and database must have for the overall system to meet its quality goals.
Deployer	Responsible for accepting the completed system from the development effort and deploying it, making it operational, and fulfilling its allocated business function.	Understanding the architectural elements that are delivered and to be installed at the customer's or end user's site, and their overall responsibility toward system function.
Designer	Responsible for systems and/or software design downstream of the architecture, applying the architecture to meet specific requirements of the parts for which they are responsible.	Resolving resource contention and establishing performance and other kinds of runtime resource consumption budgets. Understanding how their part will communicate and interact with other parts of the system.
Evaluator	Responsible for conducting a formal evaluation of the architecture (and its documentation) against some clearly defined criteria.	Evaluating the architecture's ability to deliver required behavior and quality attributes.
Implementer	Responsible for the development of specific elements according to designs, requirements, and the architecture.	Understanding inviolable constraints and exploitable freedoms on development activities.
Integrator	Responsible for taking individual components and integrating them, according to the architecture and system designs.	Producing integration plans and procedures, and locating the source of integration failures.
Maintainer	Responsible for fixing bugs and providing enhancements to the system throughout its life (including adaptation of the system for uses not originally envisioned).	Understanding the ramifications of a change.
Network administrator	Responsible for the maintenance and oversight of computer hardware and software in a computer network. This may include the deployment, configuration, maintenance, and monitoring of network components.	Determining network loads during various use profiles and understanding uses of the network.
Product line manager	Responsible for development of an entire family of products, all built using the same core assets (including the architecture).	Determining whether a potential new member of a product family is in or out of scope and, if out, by how much.
Project manager	Responsible for planning, sequencing, scheduling, and allocating resources to develop software components and deliver components to integration and test activities.	Helping to set budget and schedule, gauging progress against established budget and schedule, and identifying and resolving development-time resource contention.
Representative of external systems	Responsible for managing a system with which this one must interoperate, and its interface with our system.	Defining the set of agreement between the systems.
System engineer	Responsible for design and development of systems or system components in which software plays a role.	Assuring that the system environment provided for the software is sufficient.
Tester	Responsible for the (independent) test and verification of the system or its elements against the formal requirements and the architecture.	Creating tests based on the behavior and interaction of the software elements.
User	The actual end users of the system. There may be distinct kinds of users, such as administrators, superusers, and so on.	Users, in the role of reviewers, might rely on architecture documentation to check whether desired functionality is being delivered. Users might also refer to the documentation to understand what the major system elements are, which can aid them in emergency field maintenance.

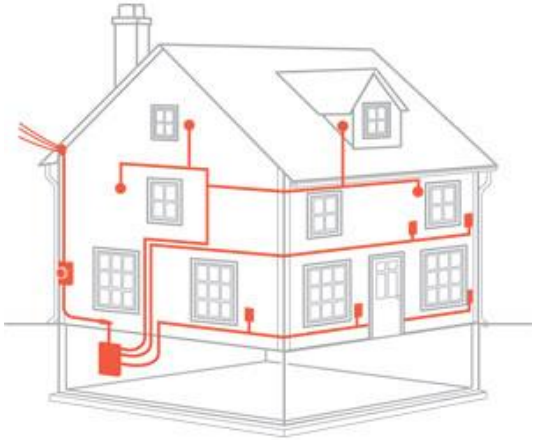
From *Views and Beyond* Table P.1

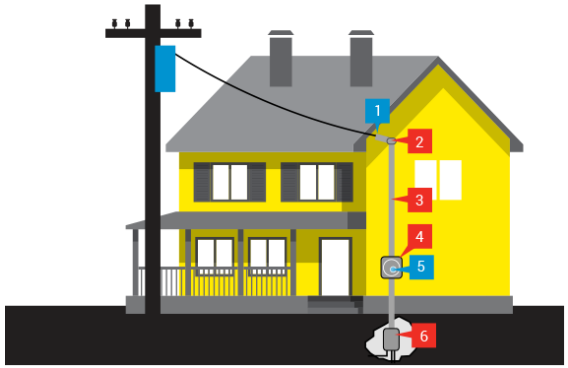


EXAMPLE VIEWS FOR A BUILDING ARCHITECTURE









Same house, different views that provide a complete representation of the house architecture

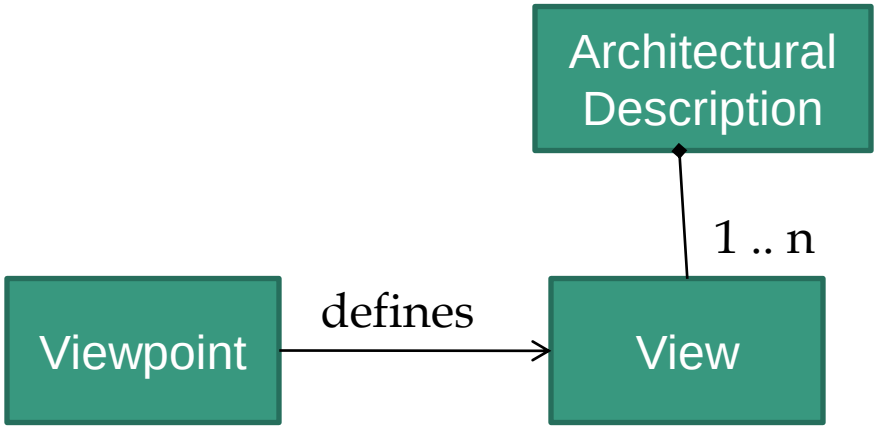
Who are the stakeholders for each of these views?

Note the specific notations (conventions) for each view

VIEWPOINTS AND VIEWS

- A **view** is a representation of all or part of an architecture, from the perspective of one or more concerns which are held by one or more of its stakeholders
- A **viewpoint** is a collection of patterns, templates and conventions for constructing one type of view

VIEWPOINTS, VIEWS, AND ARCHITECTURAL DESCRIPTION



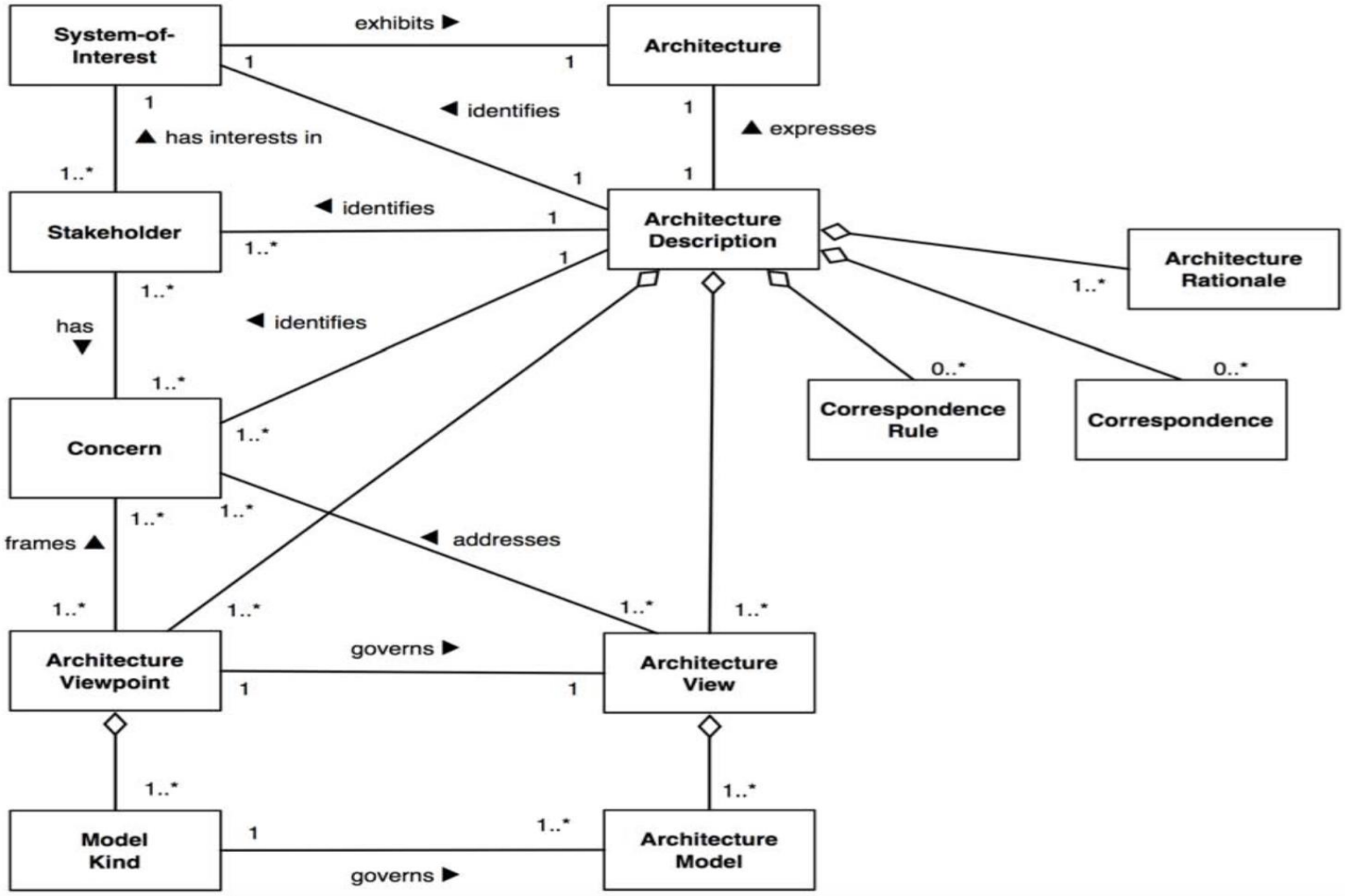
- An architectural description is a collection of one or more views
- ***A viewpoint is used to define (the structure and content of) a view***
- The structure and content of a particular view is defined by one viewpoint

IEEE STANDARD 42010

- IEEE Standard 42010 *Systems and software engineering — Architecture description*
  - Defines architecture description and documentation best practices
    - Based on viewpoints and views
    - **Does not prescribe any <u>specific</u> set of views**

IEEE STANDARD 42010

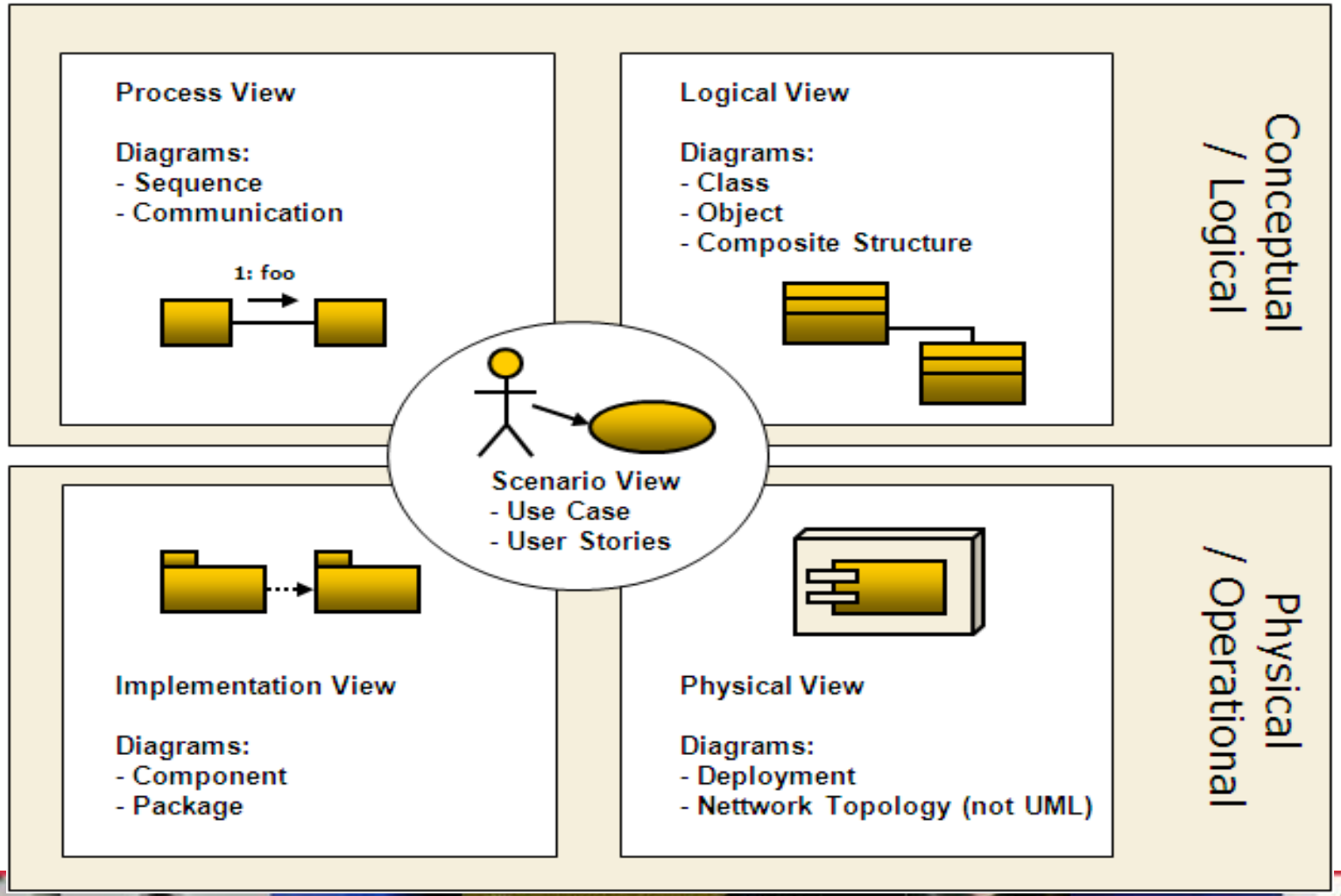
ISO/IEC/IEEE 42010:2011(E)



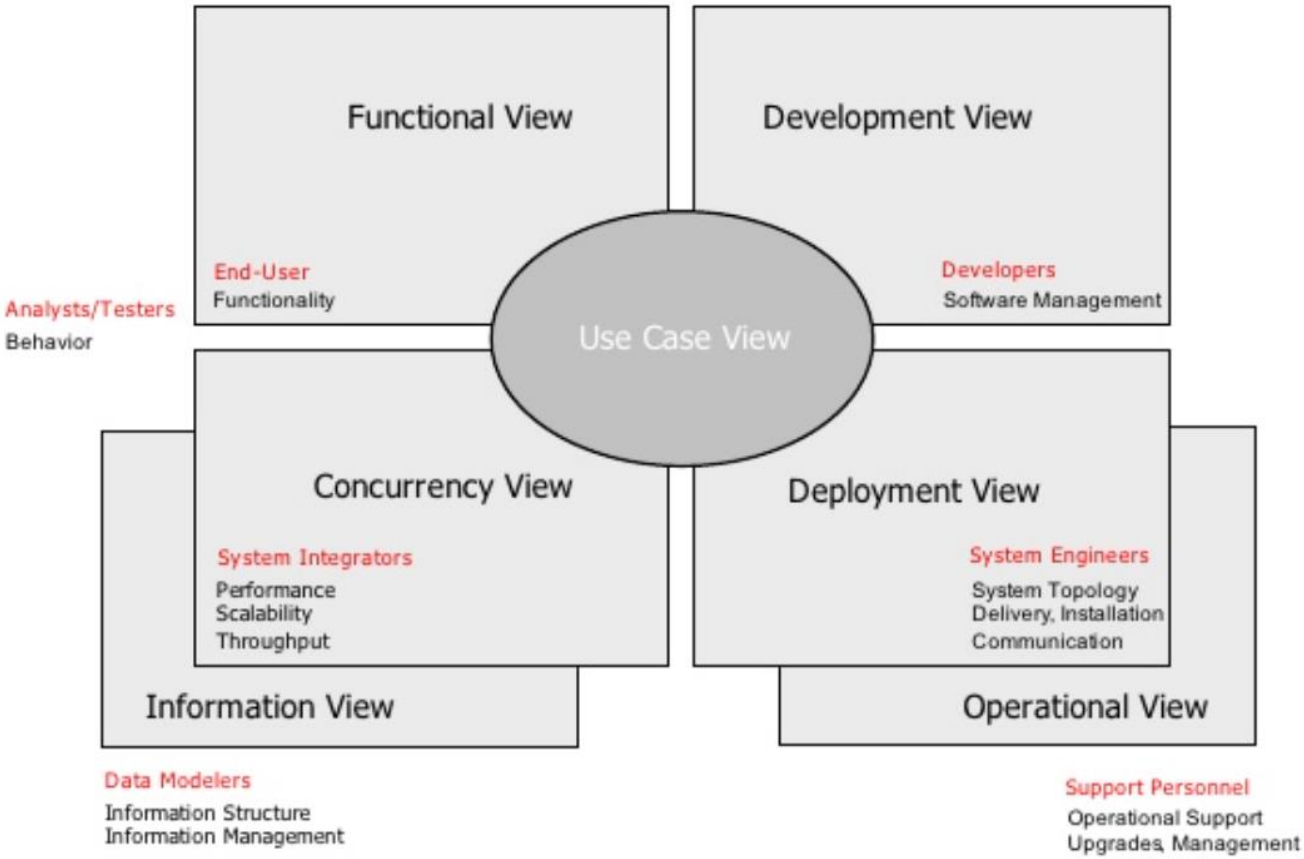
SOFTWARE ARCHITECTURE VIEWS SETS

- There have been various sets of views proposed and used:
  - IBM/Kruchten “4 + 1” Views
  - Siemens Views
  - SEI Views
  - Philips Views
  - Rozanski and Woods Views
  - …

“4 + 1” VIEWS

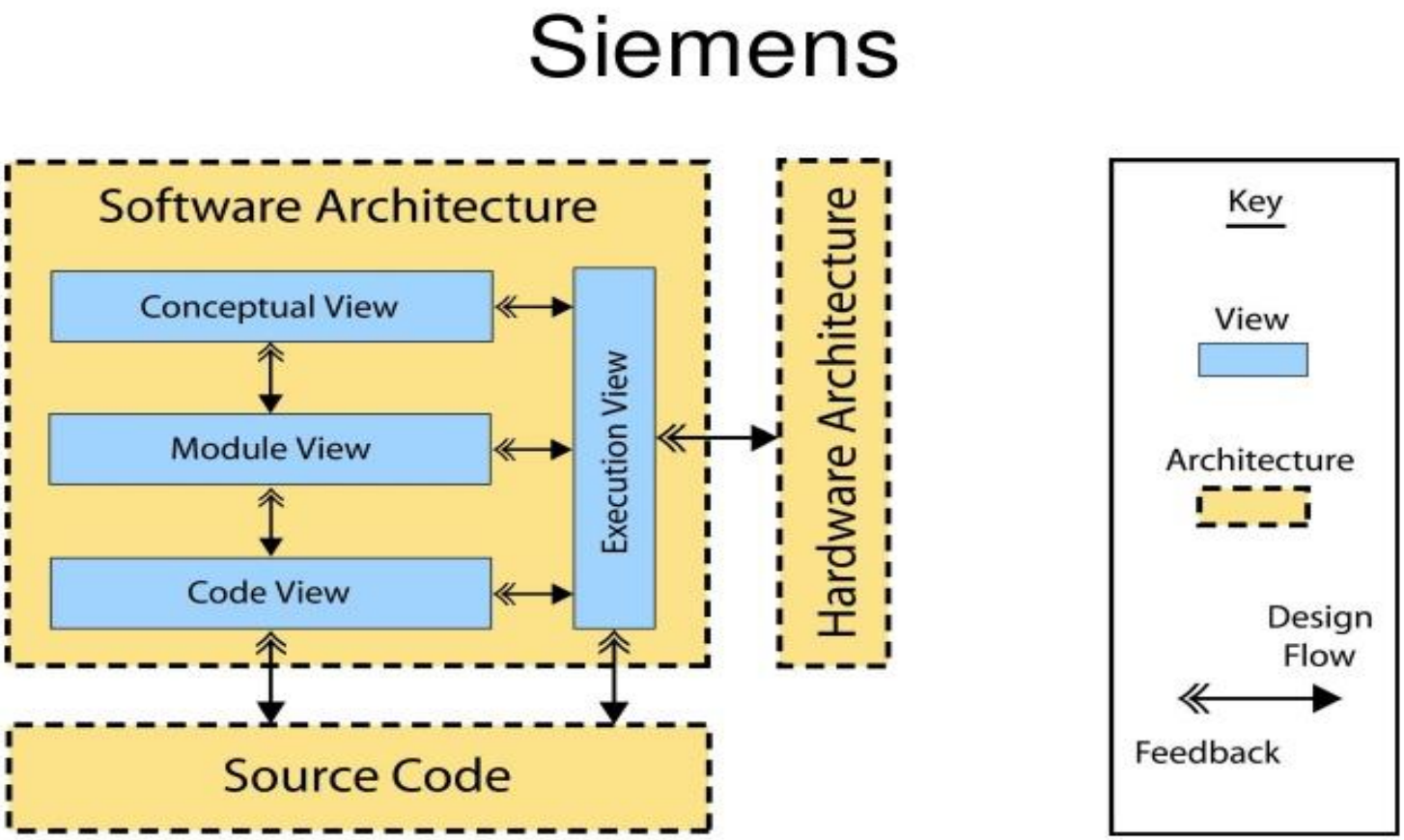


RUP + SEI VIEWS SET



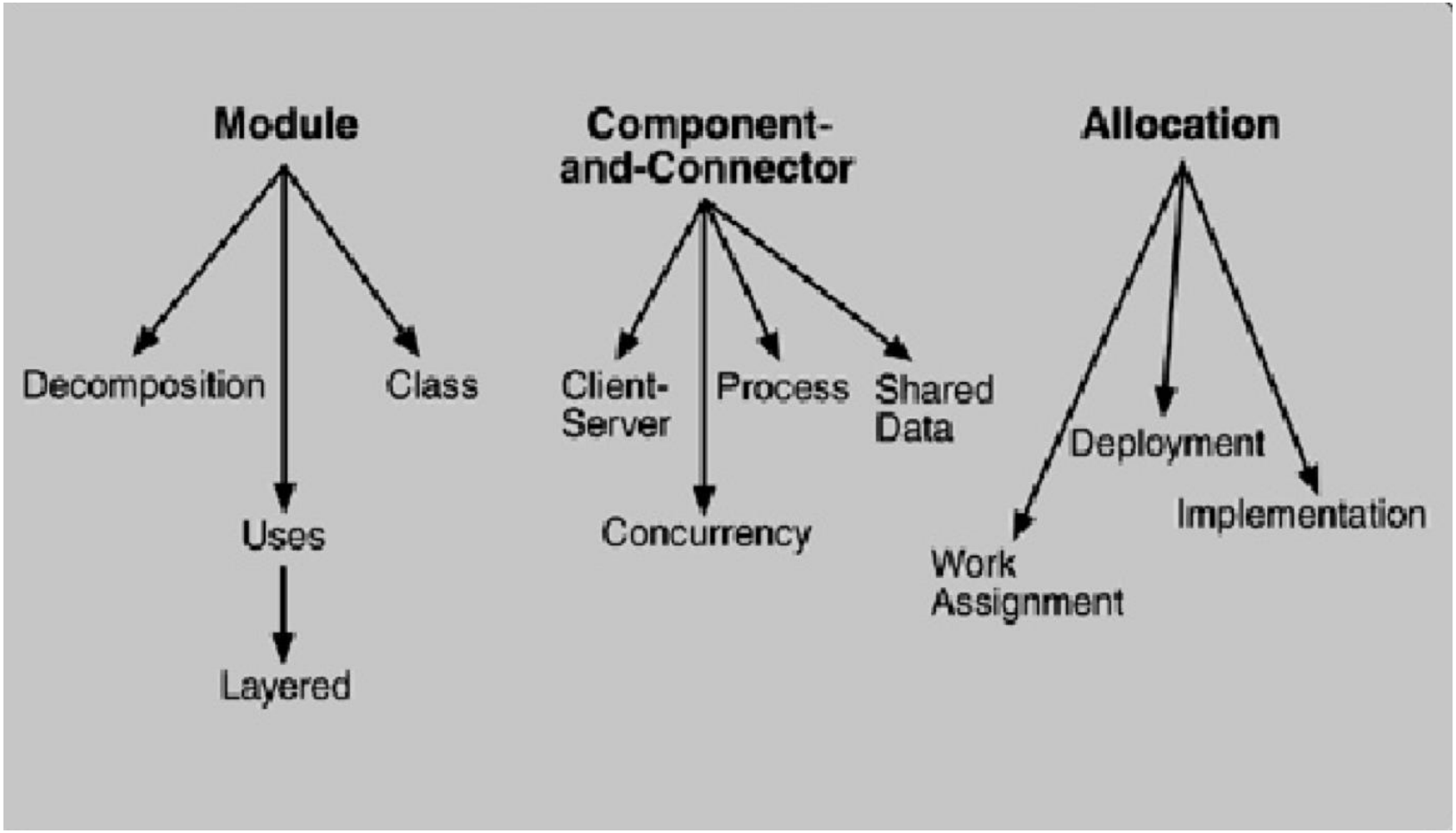
Rozanski, Nick, and Eion Woods. Software Systems Architecture. New Jersey: Pearson Education, 2005.

SIEMENS VIEWS



Adapted from "Applied Software Architecture", Hofmeister, C. et al. (2000).

SEI VIEW SETS AND VIEWS



From: *Documenting Software Architectures: Views and Beyond*, by Paul Clements et all

SOFTWARE ARCHITECTURE ASPECTS

- Structure (static) – components, relations, interfaces
- Behavior (dynamic) – how do components work together, to perform the software requirements?
  - Behavior – expressed as scenarios (the “+1” view of the “4+1” view set)
    - Use case scenarios
    - Quality scenarios
  - Behavior scenarios are used both for the architecture *construction* **and** for its *analysis/evaluation*
- Allocation – mapping to architecture elements of other considerations
  - Deployment
  - Requirements
  - Work assignment
  - Code base

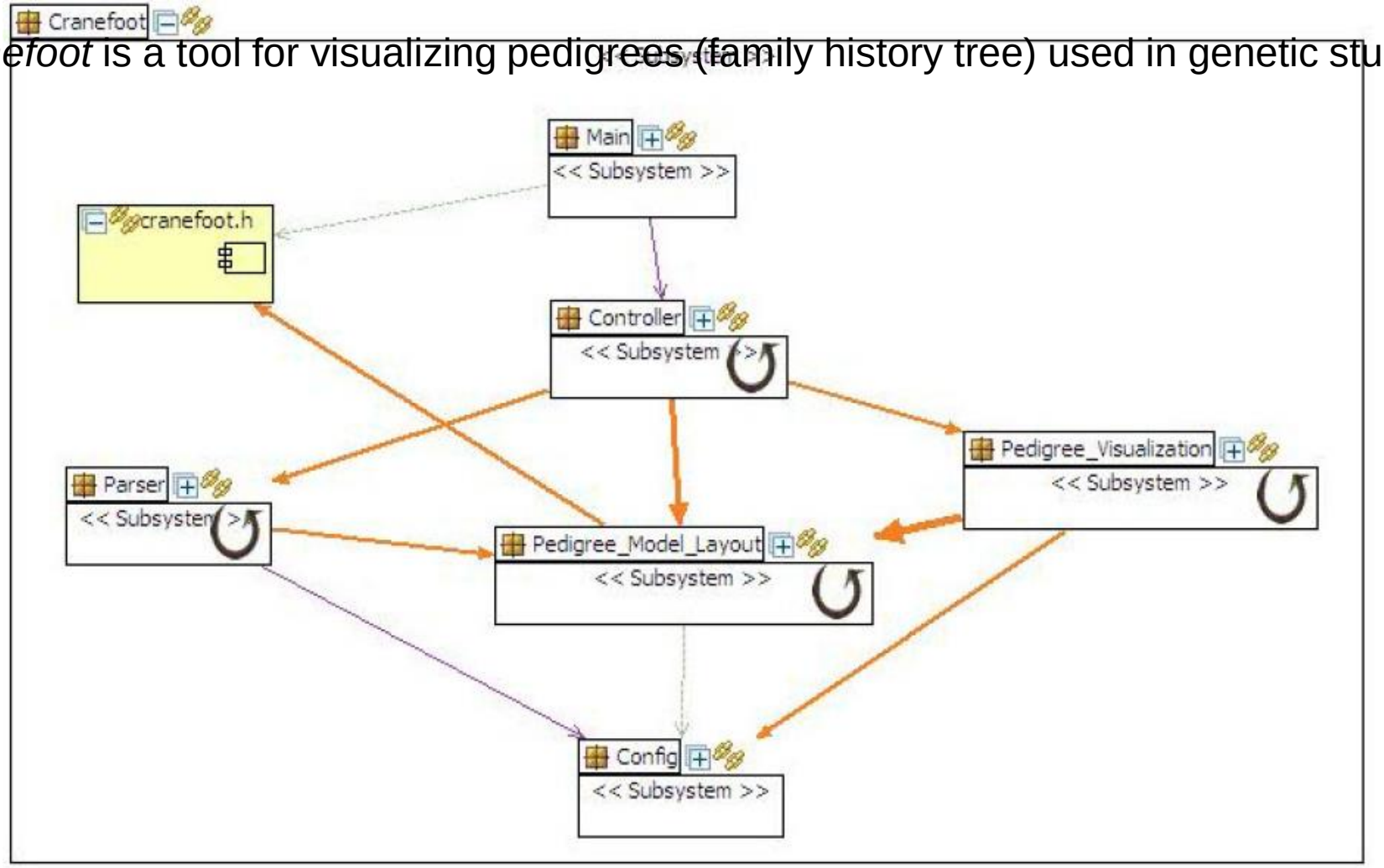
COMPARISON OF DIFFERENT VIEW SETS

Functional	
Structures	Decomposition, Uses, Layered, Abstraction.
"4+1" model SEI model Siemens model	Logical view. Module viewtype. Module view.
Behavioral	
Structures	Process, Concurrency, Shared Data, Client-Server.
"4+1" model SEI model Siemens model	Process view. C&C viewtype. Execution view.
External	
Structures	Implementation, Work Assignment.
"4+1" model SEI model Siemens model Rational ADS	Development view. Allocation viewtype. Code view. Realization viewpoint.

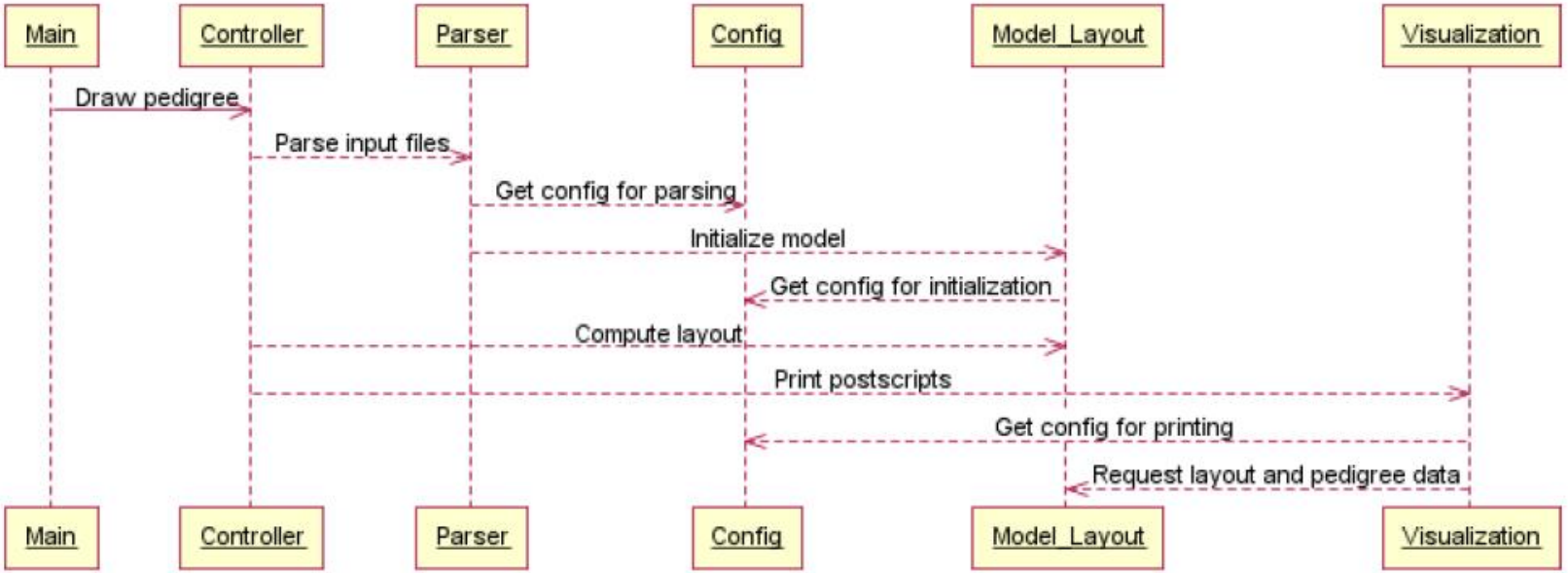
From A Survey of Software Architecture Viewpoint Models

EXAMPLE *MODULE/LOGIC* VIEW

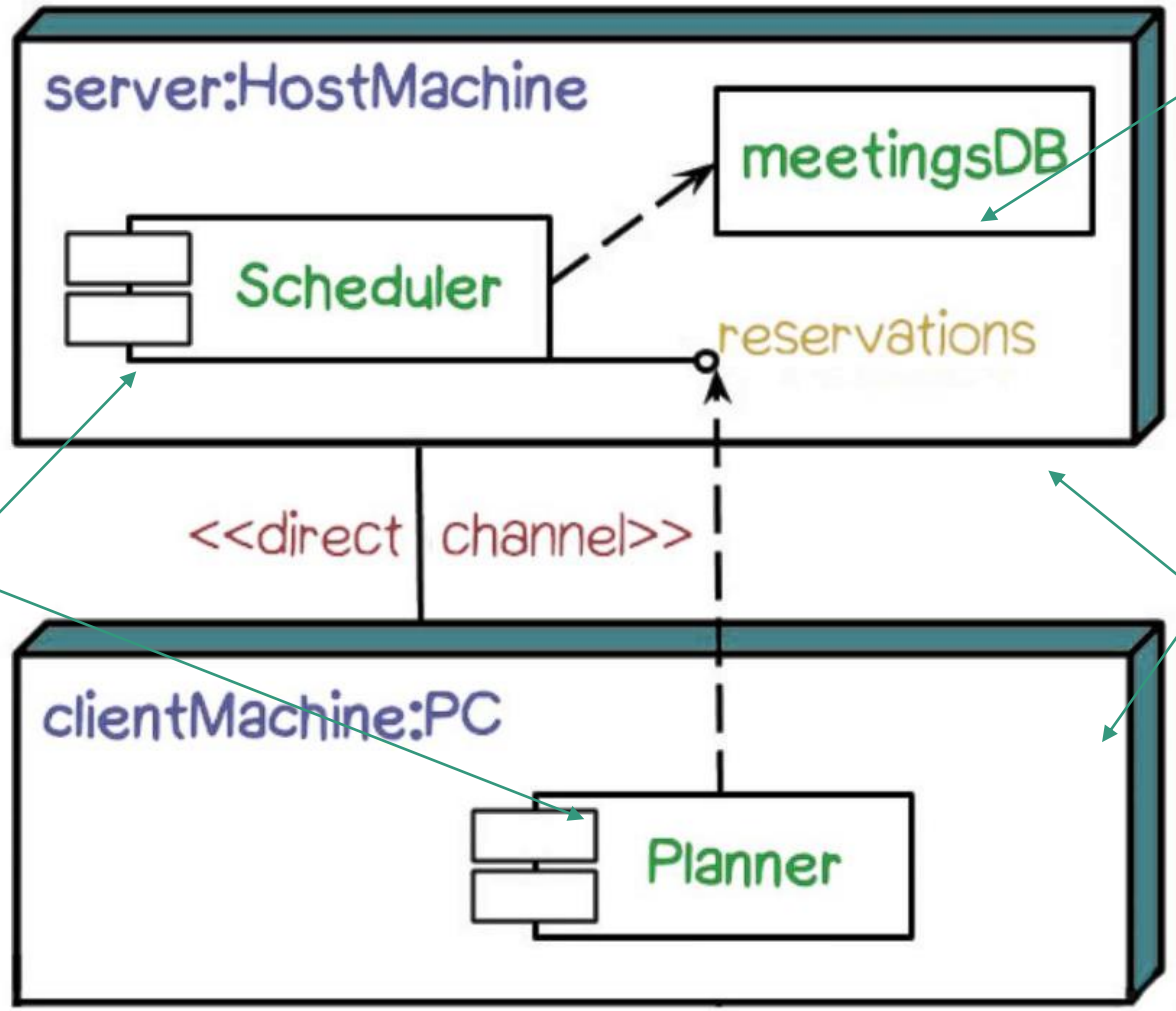
Cranefoot is a tool for visualizing pedigrees (family history tree) used in genetic studies



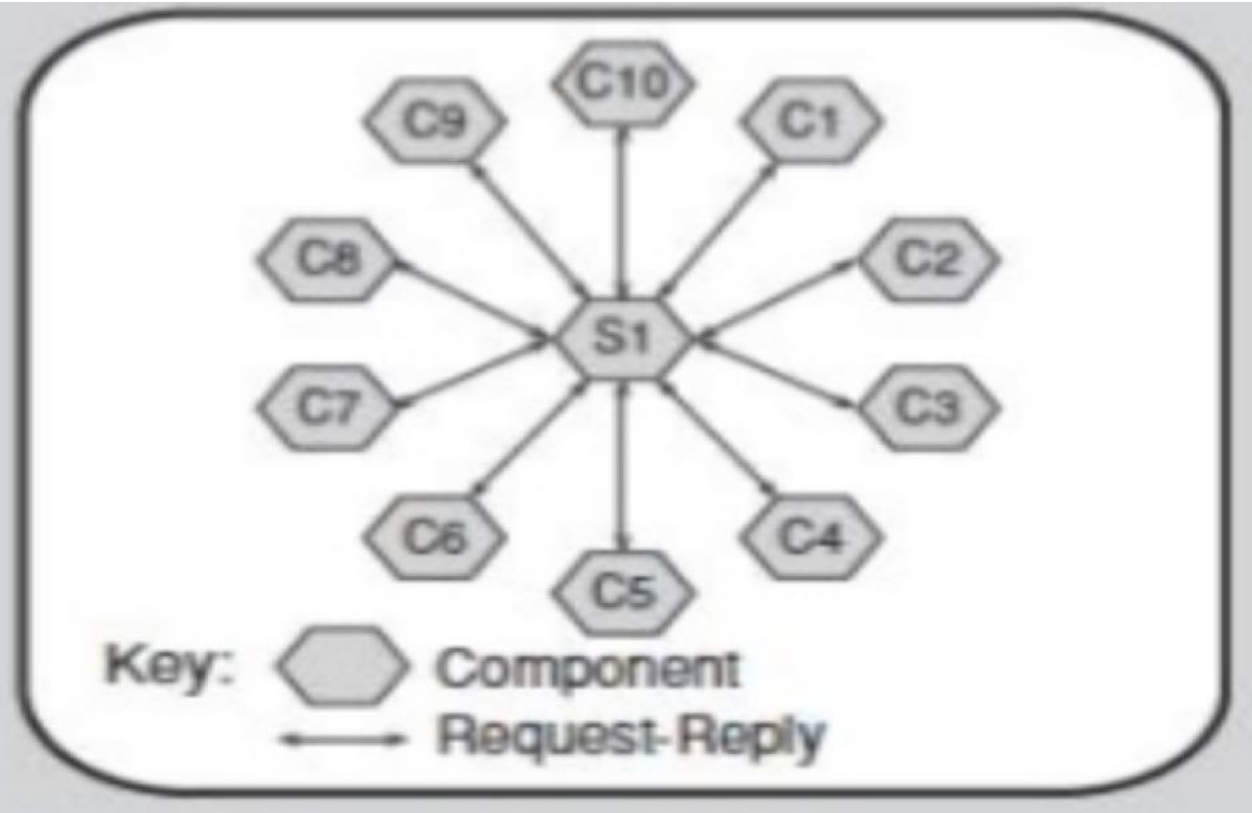
EXAMPLE BEHAVIORAL VIEW



EXAMPLE *DEPLOYMENT* VIEW



EXAMPLE *RUNTIME* VIEW



Runtime view (execution) – one instance of *Server* (S1) and 10 instances of *Client* (C1…C10)

WORK ALLOCATION VIEW

- **Manager**'s concern - allocation of architecture elements to teams/team members for further work (detailed design, implementation, testing)
  - Identify the expertise required on each team/developer
  - Determine communication mechanisms (meetings, online, etc)

	Allocated to team/team member for…		
Component	**Detailed design**	**Implementation**	**Testing**
C1	Team1	Team1	Team5
C2	Bob	Joe	Sarah

REQUIREMENTS ALLOCATION VIEW

Addresses the concern that:

- requirements are realized and
- there are no architectural elements that are not required

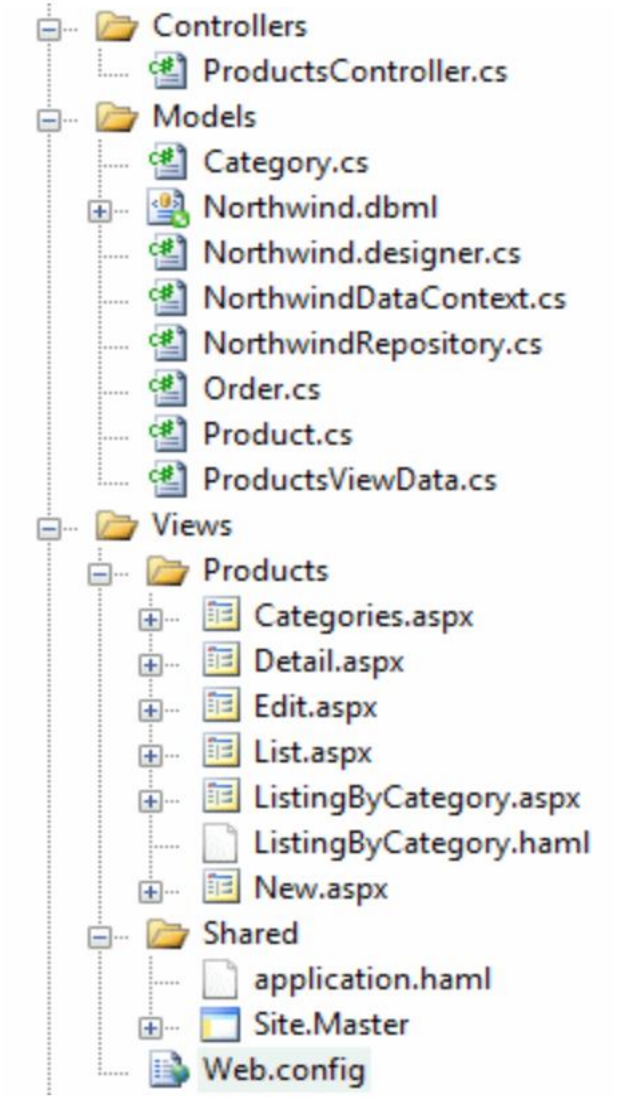
Example of Requirements allocation (trace) to architectural elements

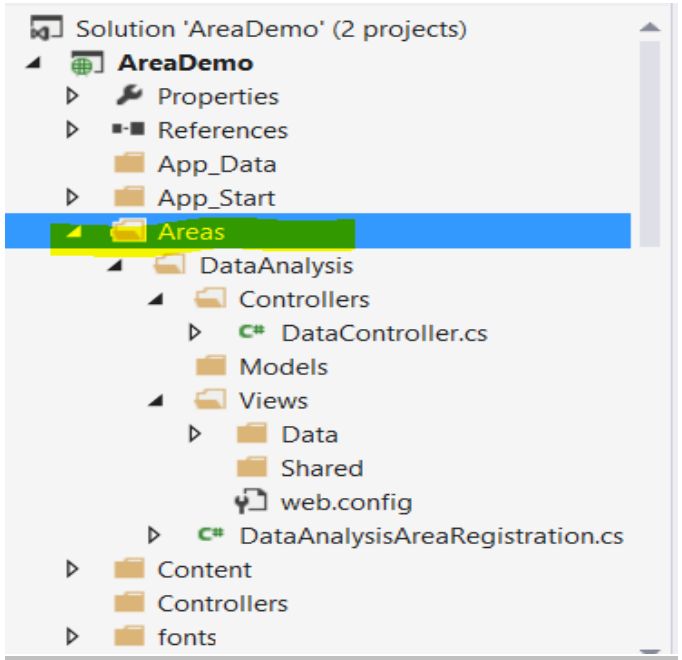
	Component1 Name	Component2 Name	Component3 Name	Component4 Name	Component5 Name
Use Case1/Feature1 Name	X			X	
Use Case2/Feature2 Name		X			X
Scenario1 Name	X	X			X
Abuse Case1 Name			X	X	

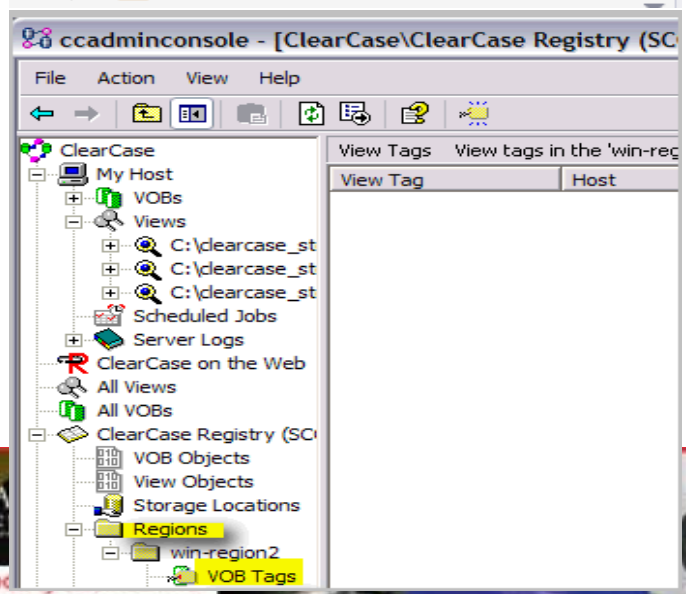
IMPLEMENTATION VIEW ("CODE ARCHITECTURE")

- Code organization
  - Allocation of architectural components to code elements
- Addresses stakeholders' concerns:
  - **Developer**:
    - Organize source code into object code, libraries, and directories
    - Map architectural elements to source code files and data repositories, for implementation and maintenance & evolution
  - **Tester**: integrate and test (I&T) code
    - Dependencies, I&T strategy
    - Incorporate acquired components
  - **Contractor:**
    - Where (in the code repository) to upload the files corresponding to the architectural components they develop
  - **Builder**: how to put together a build/configuration

EXAMPLE IMPLEMENTATION VIEWS

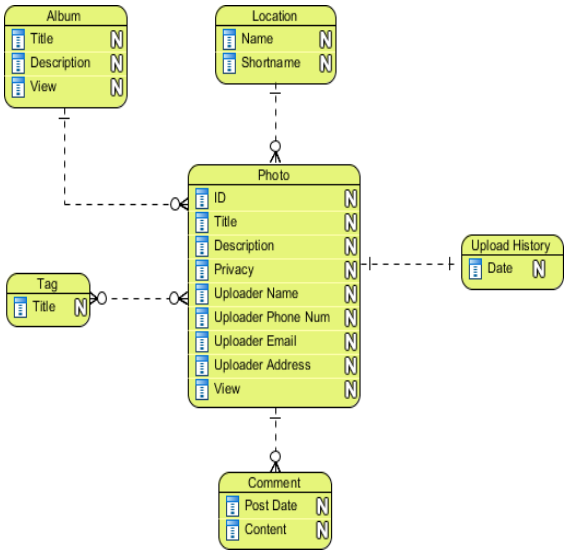




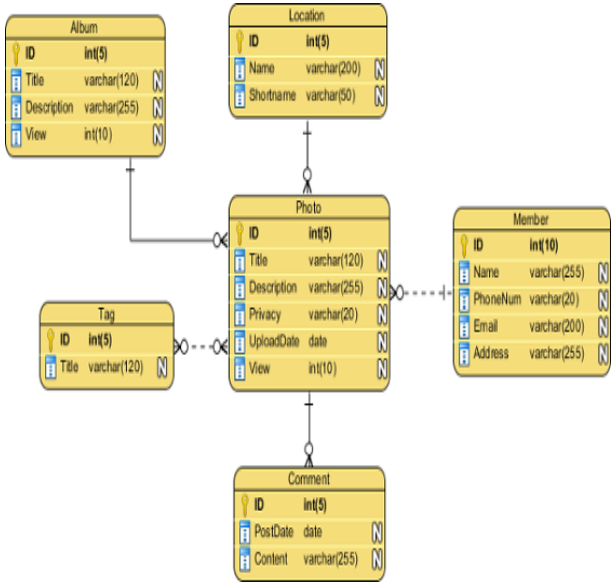


OTHER VIEWS …

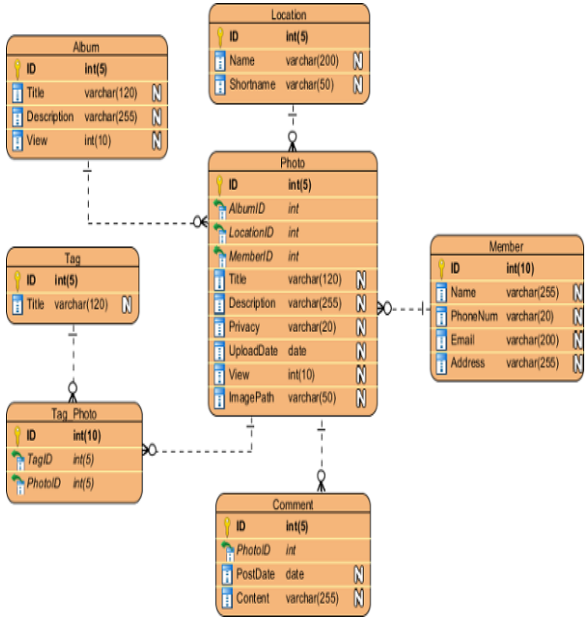
- Depending on the application domain, other views might be needed
- E.g., **Information/Data View** – Conceptual, Logical and Physical Data Models



Conceptual ERD example



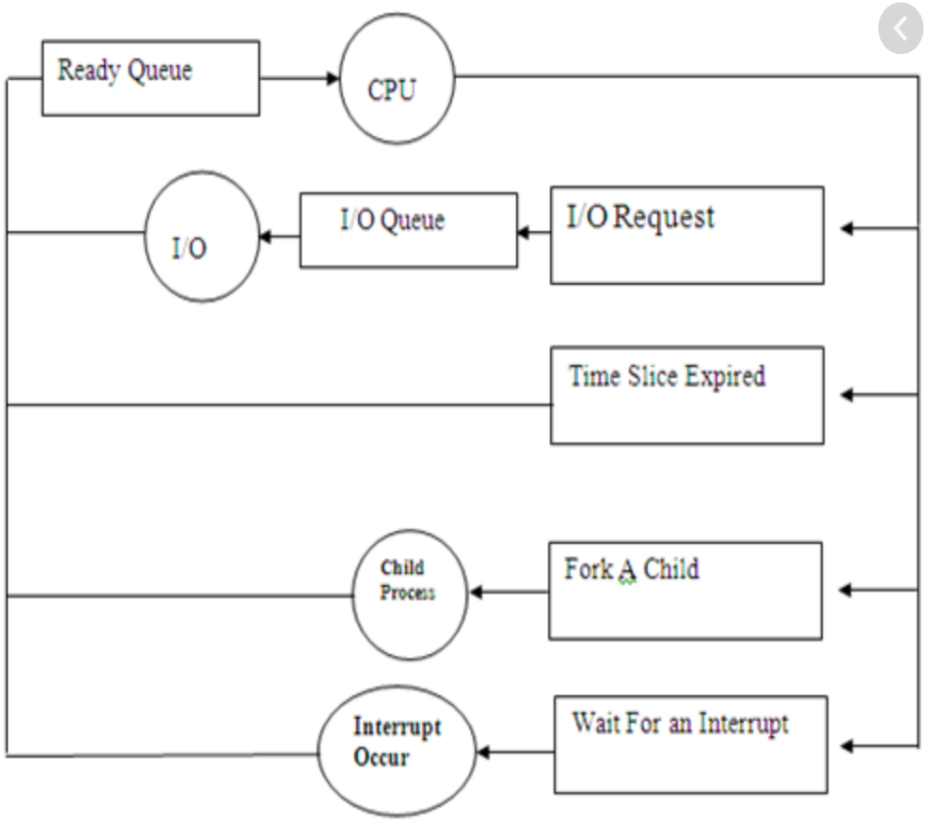
Logical ERD example

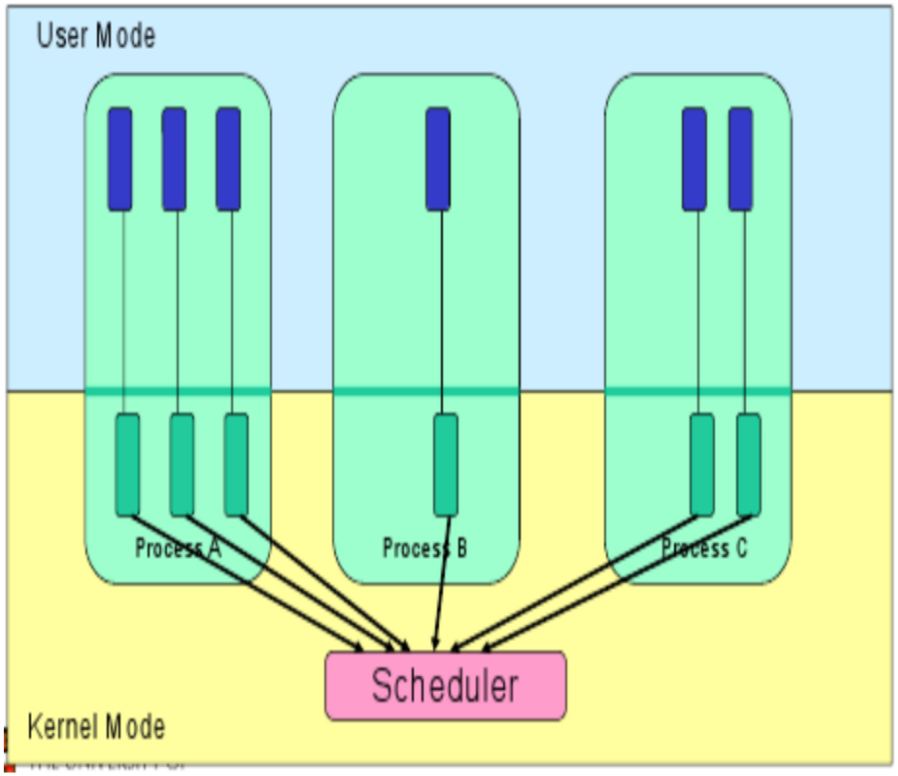


Physical ERD example

From: Visual Paradigm User's Guide

PROCESS VIEW

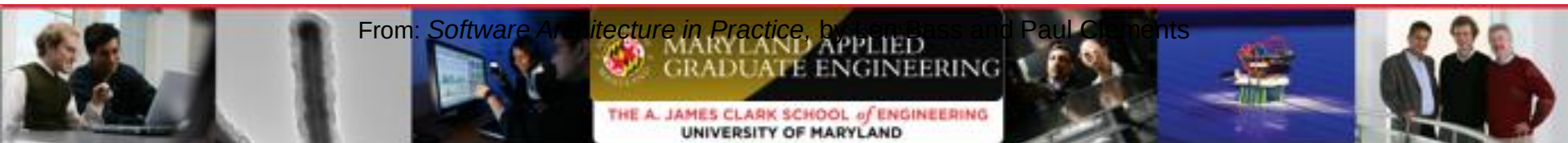




Operating system examples – from How Operating Systems Work: 10 Concepts you Should Know as a Developer

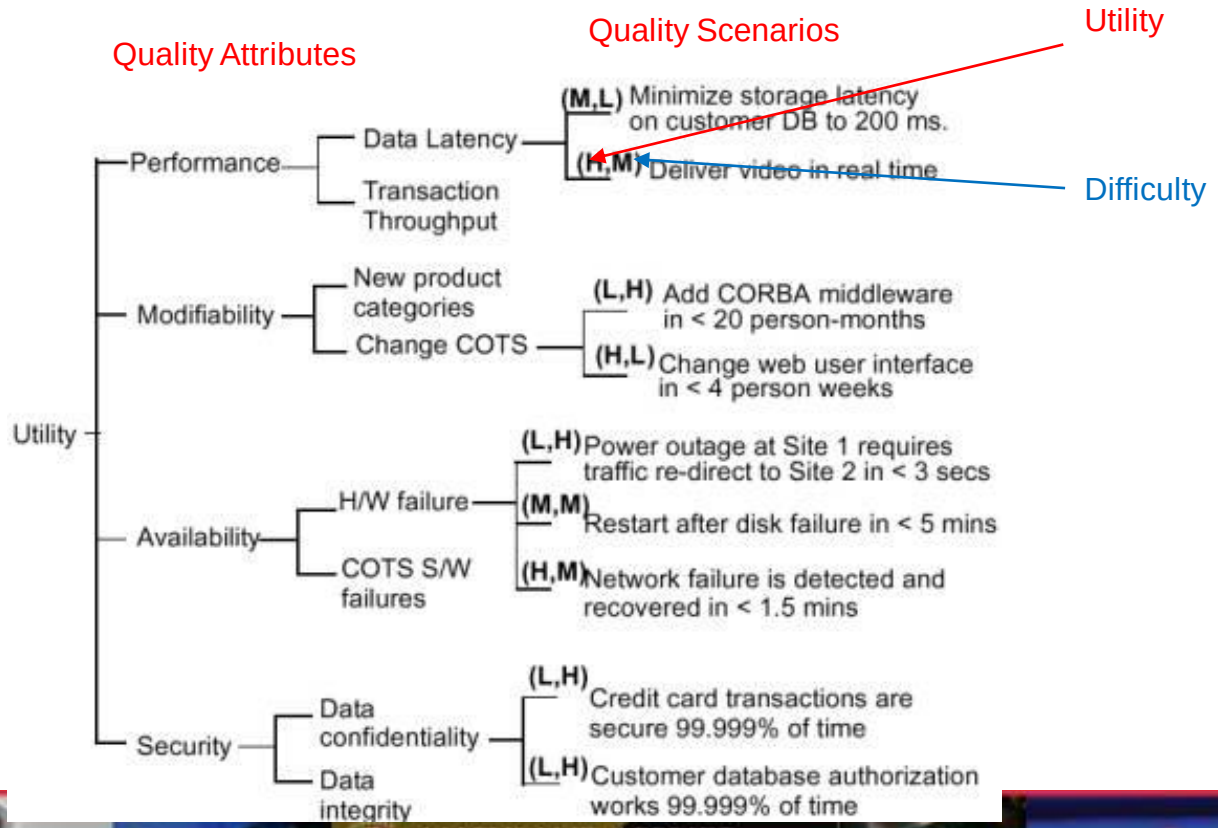
VIEWS SUPPORT REASONING ABOUT QUALITY ATTRIBUTES

- For example:
  - The **module** view (what modules use what other modules) is strongly tied to the ease with which a system can be *extended* or *modified*
  - The **concurrency** view (parallelism within the system) is strongly tied to the ease with which a system can be made *free of deadlock* and *performance* bottlenecks
  - The **deployment** view is strongly tied to the achievement of *performance*, *availability*, and *security* goals



QUALITY SCENARIOS DIFFICULTY/RISKS EVALUATION

- As architecture is developed, we may need to refine assessment of scenarios for their **difficulty** of implementation and **risk**
  - Reprioritize scenarios based on both ***importance*** to stakeholders **and *difficulty/risk***



WHAT VIEWS TO USE?

- **It depends** … on what is needed in **your** project! e.g.,
  - All architectures need structure and behavior representations
  - If only one HW node and environment (OS) -> no deployment view necessary
  - If only one process and thread -> no process view necessary
  - If only one team member -> no work allocation view necessary
  - No stored data -> no information/data view necessary

*A project **can** identify their own set of views; BUT the viewpoints for creating these views need to be **defined** (what they contain and how they are represented)*

CLASS EXERCISE

- Identify architecture concerns, aspects, elements, and views that you will need to document your LMS architecture

VIEWPOINTS FOR THE CLASS PROJECT

- **Logical**
  - Structural/static view
  - Contains models that represent architectural components and their relations and interfaces – use UML component diagrams
  - Descriptions/specifications of components and interfaces – use provided templates
- **Behavior**
  - Shows how your architectural components collaborate to realize (top priority) functional scenario or quality scenarios – use UML activity or sequence diagrams
- **Allocation**
  - **Deployment**
    - Allocation of software artifacts (e.g., executables and data) to hardware nodes – use UML deployment diagrams

VIEWPOINTS FOR THE CLASS PROJECT (CONTINUED)

- **Allocation** (continued)
  - **Requirements allocation**/trace
    - Allocation of main requirements to architecture components – use provided template (table)
  - **Architecture work allocation**
    - Allocation of architecture components to team members - use provided template (table)
- **Information**
  - Data models (e.g., conceptual, logical, physical data base models) – use ERD, UML class, diagrams or tables
- **Implementation** (code architecture)

- You will create ***views*** (models) that represent your LMS's architecture, based on these *viewpoints*
  - Views are instantiation of viewpoints

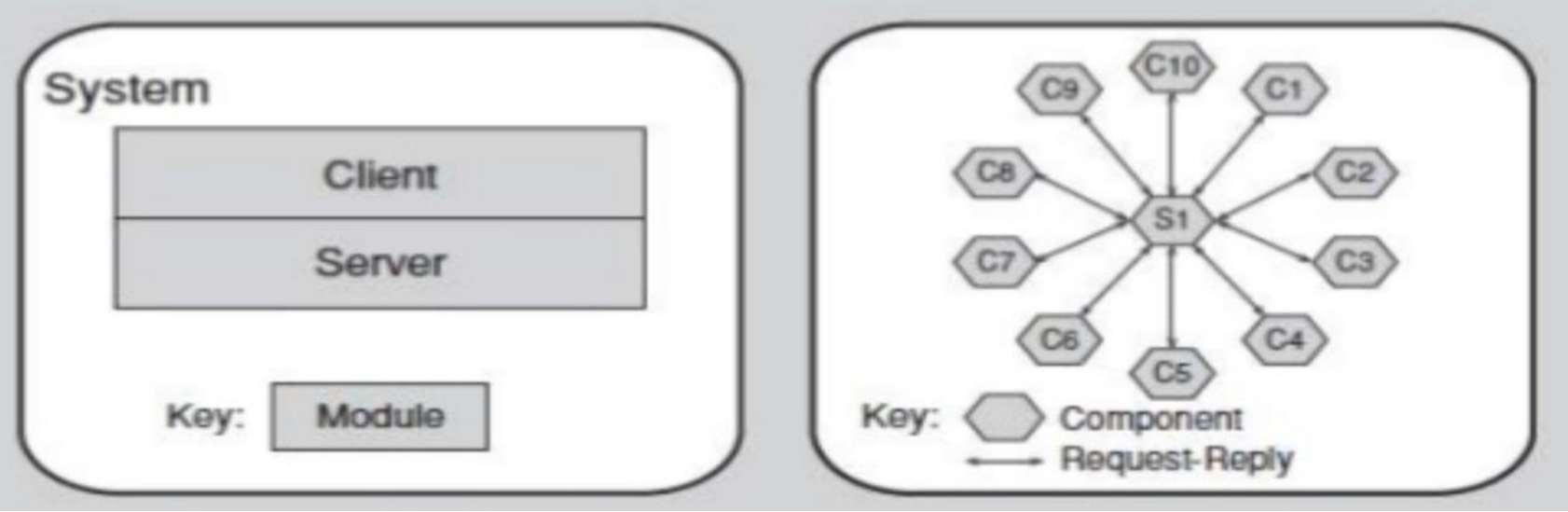
VIEWS RELATE TO EACH OTHER

- Elements of one view are related to elements of other views
  - We need to reason about these relations
- For example, a component in a decomposition (logical) view may be manifested as one, part of one, or several components in one of the runtime views
- In general, mappings between view elements are *many to many*

VIEWS MAPPING EXAMPLE LOGIC-RUNTIME

Logic view (static, structural)

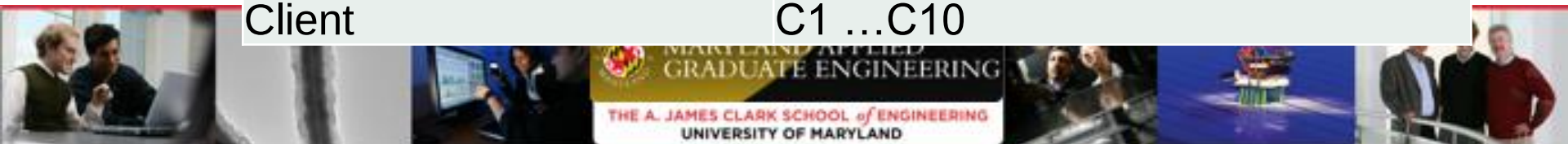
Runtime view (at execution) – one instance of *Server* and 10 instances of *Client*



From: Software Architecture in Practice, L. Bass, P. Clement, R Kazman

Views Mapping

Logic view element	Runtime view element
Server	S1
Client	C1 …C10



VIEWS MAPPING EXAMPLE LOGIC-IMPLEMENTATION

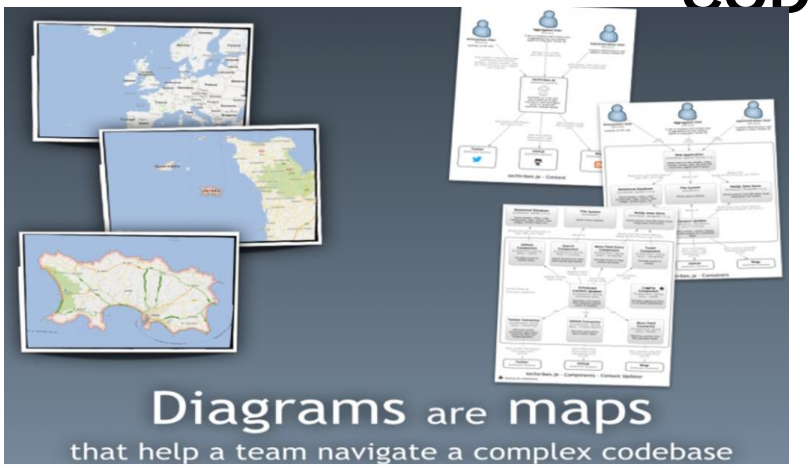
In what directories or files is each element stored during development, testing, and system building?

Logic view **Implementation view**

Structural Element	Implementation Element
Controller	pedigree.cpp, Pedigreeobject.cpp, Pedigreeobject.run.cpp
Parser	pedigreeobject.import.cpp, pedigreeobject.configure.cpp
Config	configTable.cpp, Tablet.h
Model	family.h, family.cpp, branch.connect.cpp, familyobject.cpp, familyobject.branch.cpp, familyobject.check.cpp, familyobject.link.cpp, branch.cpp, familyobject_methods.cpp, member.h, member.cpp, pedigreeobject.h, scriptum.h
Layout Algo\Inter-Branch	**Inter-Branch:** familyobject.simulate.cpp, branch.attract.cpp, branch.repel.cpp
Layout Algo\Intra-Branch	**Intra-Branch:** walker.cpp, branch.walk.cpp, branch.update.cpp, walker.h, walkertools.h, walkertools.cpp
Visualization\Print	pedigreeobject.print_toc.cpp, pedigreeobject.print.cpp, pedigreeobject.print_edges.cpp, pedigreeobject.print_nodes.cpp, pedigreeobject.print_links.cpp, pedigreeobject.print_hlegend.cpp, pedigreeobject.print_vlegend.cpp
Visualization\Postscript	postscript.cpp, psobject.cpp, psobject_methods.cpp, psobject.h, psobject.prolog.cpp

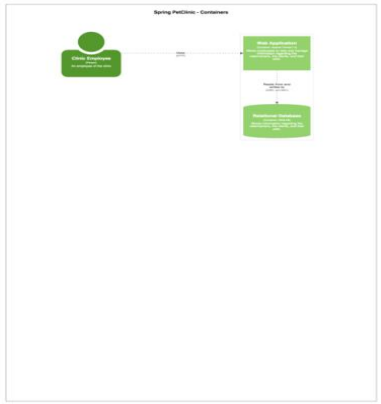
http://www.codingthearchitecture.com/2015/11/25/software_architecture_diagrams_should_be_maps_of_your_source_code.html

SOFTWARE ARCHITECTURE DIAGRAMS - MAPS OF SOURCE CODE

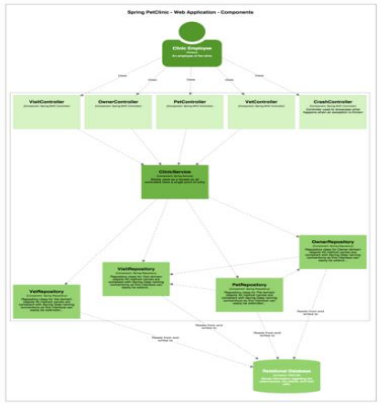


Simon Brown ***Coding the Architecture***
C4 software architecture model
Structurizr

System Context diagram



Container diagram



Component diagram

Source code

Double-click a software system

Double-click a container

Double-click a component

http://www.codingthearchitecture.com/2015/11/25/software_architecture_diagrams_should_be_maps_of_your_source_code.html

CONSISTENCY ACROSS VIEWS

- Views help us with partitioning the representation of our architecture
  - Separation of concerns
- But, the challenge is how to ensure consistency between them
- We need to ensure that the structures, features, and elements that appear in one view are compatible and in alignment with the content of other views

CONSISTENCY ACROSS VIEWS (CONTINUED)

- Consistency is a vital characteristics of the architecture description
- Without consistency, the system:
  - will not work properly
  - will not achieve its design goals
  - may even be impossible to build
- No tools exist yet to automate consistency between views
- Ensuring consistency comes down to the skill, thoroughness, and diligence of the architect
  - *Views consistency* checklists could help during reviews

EXAMPLE LOGICAL AND PROCESS VIEWS CONSISTENCY

- **Logical and Process View consistency/mapping**
  - **Goal:** Ensure that the *functional elements* are all mapped to a *task (process/thread)* that will allow them to execute, and the *inter-element interactions* are supported by the *inter process communication mechanisms*, if required
  - **Consistency check**: Is every *functional element* in the ***Logical view*** mapped to a *concurrent element* (a process or thread) responsible for its execution in the ***Process view***?
    - This can be a criterion/question in the design review checklist

OUTLINE

- Lecture
  - Documenting architecture
    - Architecture Views
    - Notations for documenting architecture
    - Documenting components and interfaces
    - Design decisions and their documentation
- Assignments

We are here

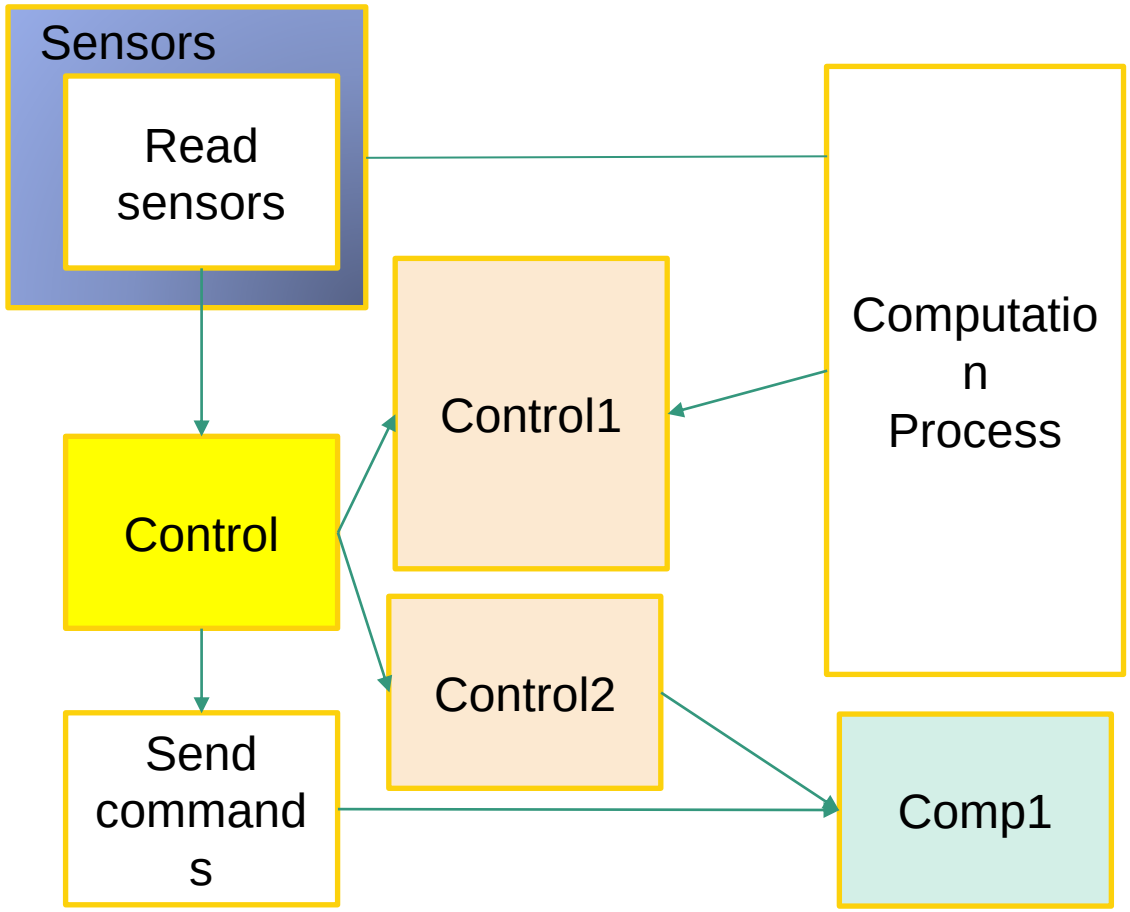
NOTATIONS FOR ARCHITECTURAL SPECIFICATION

Type of Information Specification	Useful Notations
Decomposition	Box-and-line diagrams UML class diagrams, package diagrams, component diagrams, deployment diagrams
Responsibilities	Text, box-and-line diagrams, class diagrams
Collaborations	Sequence and communication diagrams, activity diagrams, box-and-line diagrams
Interfaces	Text, class diagrams
Properties	Text
States and Transitions	State diagrams
Relationships	Box-and-line diagrams, component diagrams, class diagrams, deployment diagrams, text

“BOX-AND-LINE” DIAGRAMS

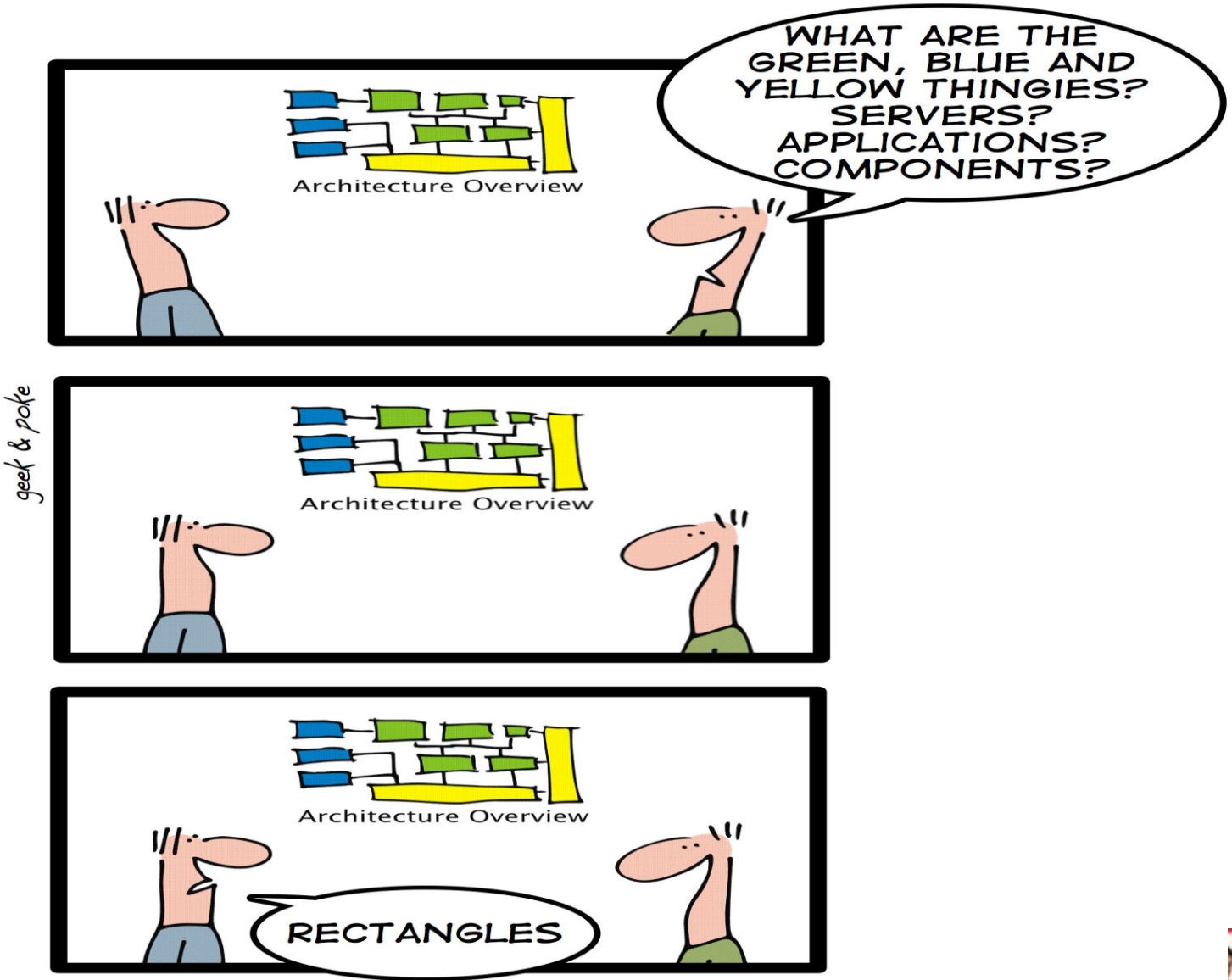
- Icons (boxes) connected with lines
- No rules governing formation
- Can be used for both static and dynamic modeling

BOX AND LINE DIAGRAM



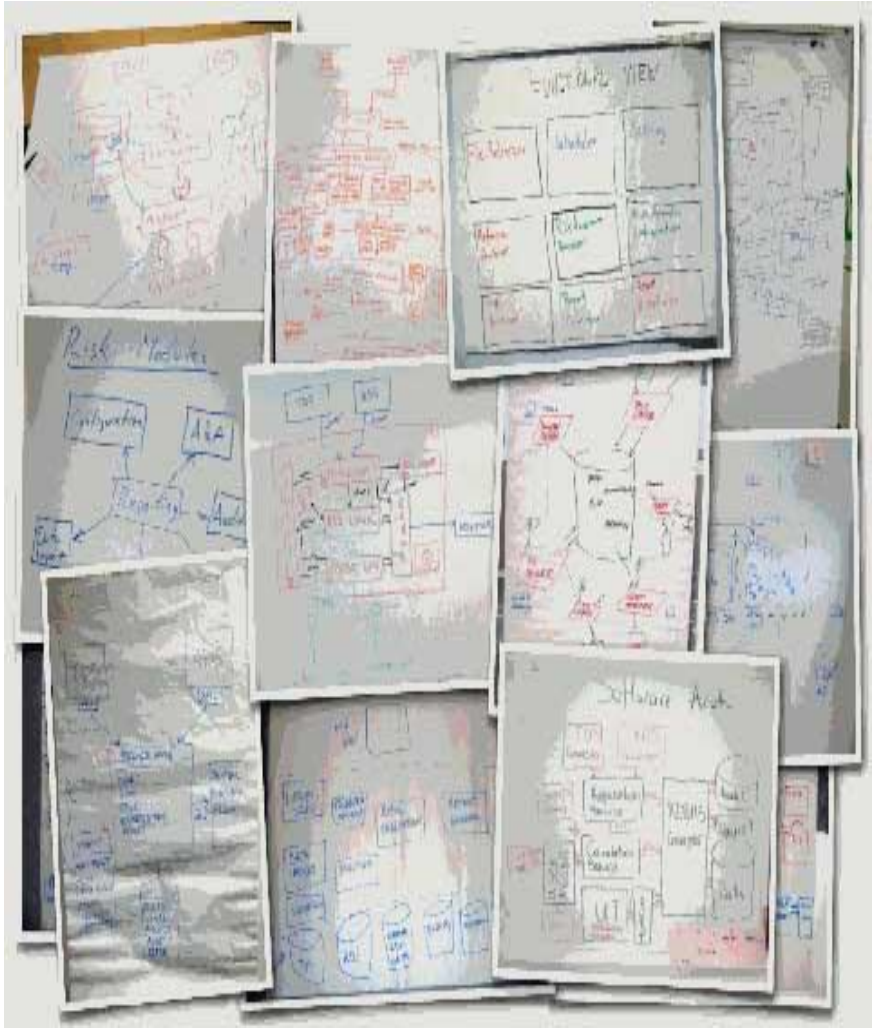
- How do you read this diagram?
- What does it tell you?
- What does it represent?
- Do you see any issues with this representation?

READING A BOX-AND-LINE DIAGRAM ☺



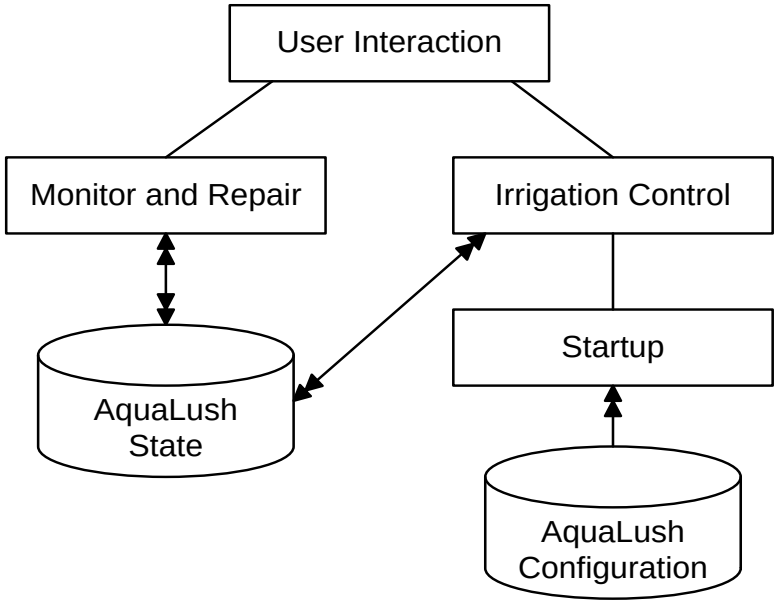
BOX AND LINE DIAGRAMS ISSUES

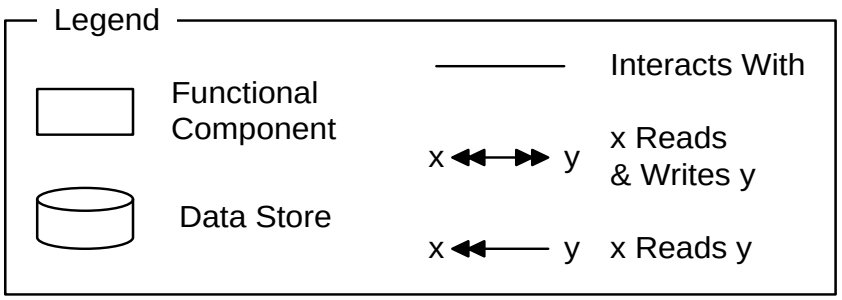
- Color-coding is usually not explained or is often inconsistent.
- The purpose of diagram elements (i.e. different styles of boxes and lines) is often not explained.
- Key relationships between diagram elements are sometimes missing or ambiguous.
- Generic terms such as "business logic" are often used.
- Technology choices (or options) are usually omitted.
- Levels of abstraction are often mixed.
- Diagrams often try to show too much detail.
- Diagrams often lack context or a logical starting point.



From: Simple Sketches for Diagramming your Software Architecture

BOX-AND-LINE DIAGRAM EXAMPLE - *AQUALUSH*





From: *Introduction to Software Engineering Design*, by C. Fox

BOX-AND-LINE DIAGRAMS HEURISTICS

- Make box-and-line diagrams only when no standard notation is adequate
- Keep the boxes and lines simple
- **MUST include a legend**
- **Make symbols for different things look different**
- **Use symbols consistently in different diagrams**
- Use grammatical conventions to name elements
- **Don’t mix static and dynamic elements.**

UML DIAGRAMS FOR ARCHITECTURE REPRESENTATION

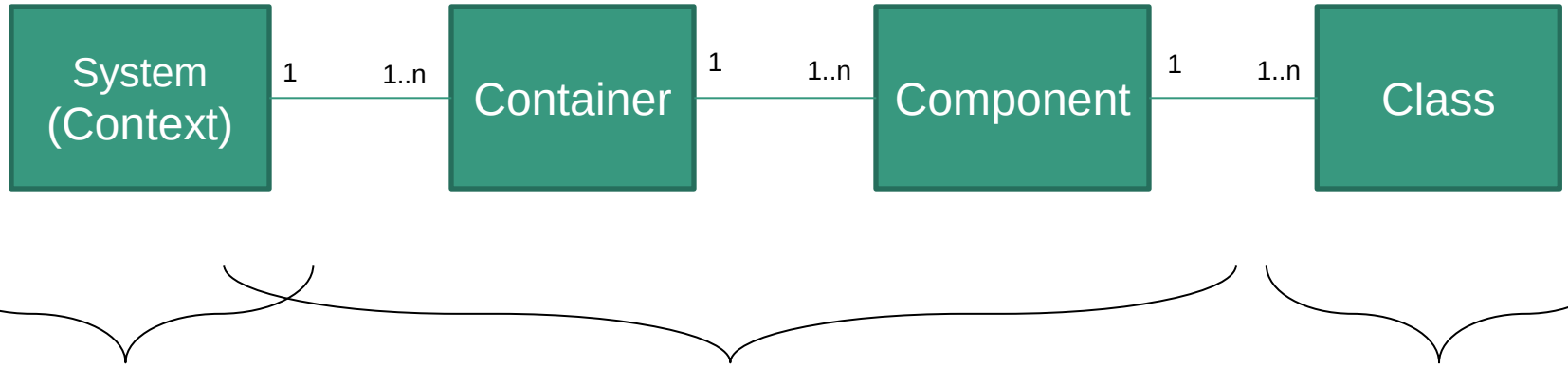
- UML Package diagrams are used to model sub-systems and their parts
- UML Component diagrams are used to model software components
- UML Activity, Sequence, Collaboration diagrams are used to model the behavior of the architecture elements
- UML Deployment diagrams are used to model physical architectures

ADVANTAGES & DISADVANTAGES OF DIFFERENT NOTATION

- "Box and line"
  - Advantages
    - Easier to learn, informal
    - Provide Flexibility
  - Disadvantage
    - Diagrams don't make much sense unless they are accompanied by a detailed narrative, which ultimately slows the team down
    - Potential lack of consistency - If heuristics are not defined and followed, use of this notation results in models that are difficult to read and understand
      - E.g., same symbols with different meanings, use undefined symbols, etc.
- Modeling language (UML)
  - Advantages
    - Well defined syntax
  - Disadvantage
    - Takes time to learn
    - Teams discuss their software systems with a common set of abstractions in mind rather than understanding what the various notational elements are trying to show (what they model) – notation over model content and purpose

ANOTHER NOTATION OPTION: ABANDONING UML…

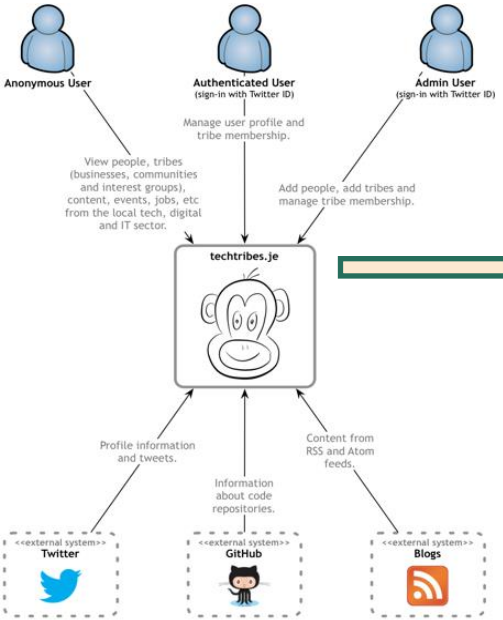
- … but not being totally informal
- Common abstractions over a common notation
- Use a small collection of simple diagrams that each shows a different part of the same overall story -> Simon Brown's "4C model" – **C**ontext, **C**ontainer, **C**omponent, **C**lass (Simple Sketches for Diagramming your Software Architecture)
  - These are ***structural*** diagrams



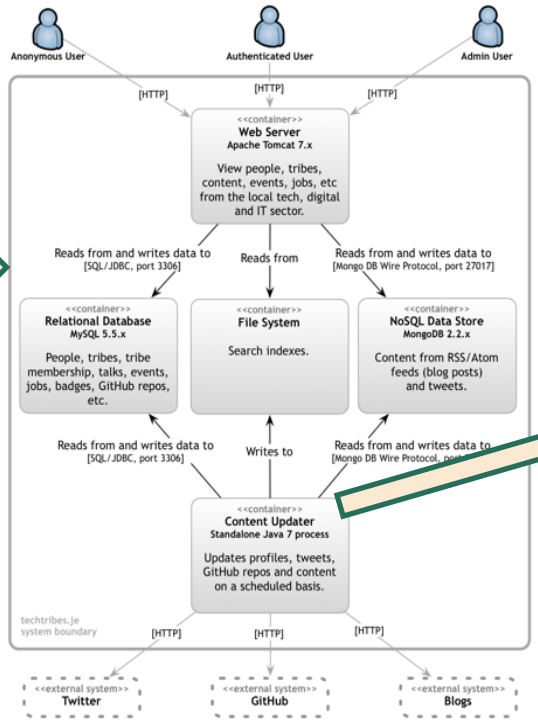
“4C” EXAMPLE

From: Simple Sketches for Diagramming your Software Architecture

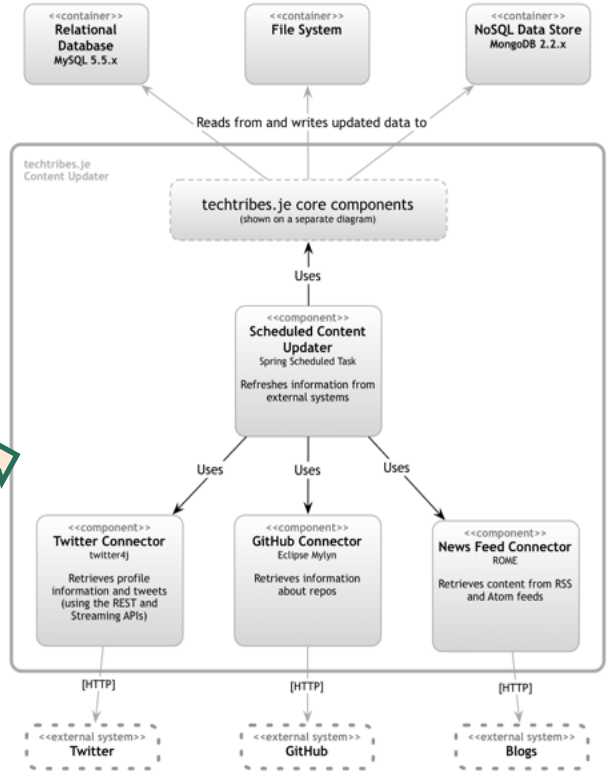
- The techtribes.je website provides a way to find people, “tribes” (businesses, communities, interest groups, etc) and content related to the tech, IT and digital sector. At the most basic level, it's a content aggregator for local tweets, news, blog posts, events, talks, jobs and more.



techtribes.je - Context



techtribes.je - Containers



techtribes.je - Components - Content Updater
Standalone Java Process

RIGOR OF SOFTWARE MODELS AND NOTATIONS

Less Rigor

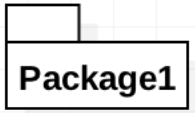
More Rigor

- What notation to use?
- Depends on the **purpose of the model**
  - For facilitating discussion about an existing or proposed system
    - These models may be **incomplete** (so long as they cover the key points of the discussion) and may use the modeling notation **informally**. This is how models are normally used in so-called 'agile modeling'
  - For documenting an existing system
    - These models **do not have to be complete** (could be for only parts of a system. However, these models have to be **correct** —they should use the notation correctly and be an **accurate** description of the system.
  - As a detailed system description that can be used to generate a system implementation
    - These models are used as part of a model-based development process, so they must be both **complete and correct**. **Rigorous notation must be used.**

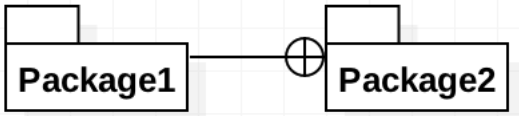
UML NOTATIONS FOR ARCHITECTURE REPRESENTATION

- *Structure* (static) aspects – UML package diagram and component (or class) diagram
- *Behavior* – UML activity diagram, state diagram, sequence diagram, timing diagram
- *Deployment* – UML deployment diagram

UML PACKAGE DIAGRAM

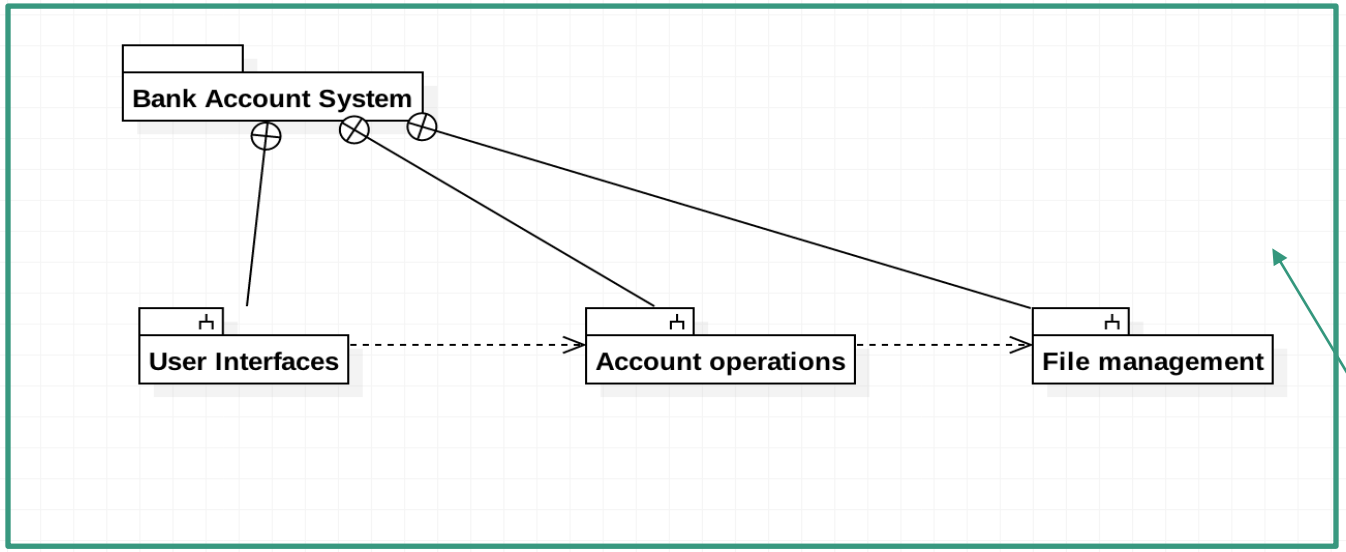
- A UML *package* is a collection of model elements, called *package members*
- The package symbol is a “file folde 

  - Package name appears in tab if body is , otherwise in the body
  - Members of the package are shown in the body of the package, or using a containment symbol (circled plus sign)



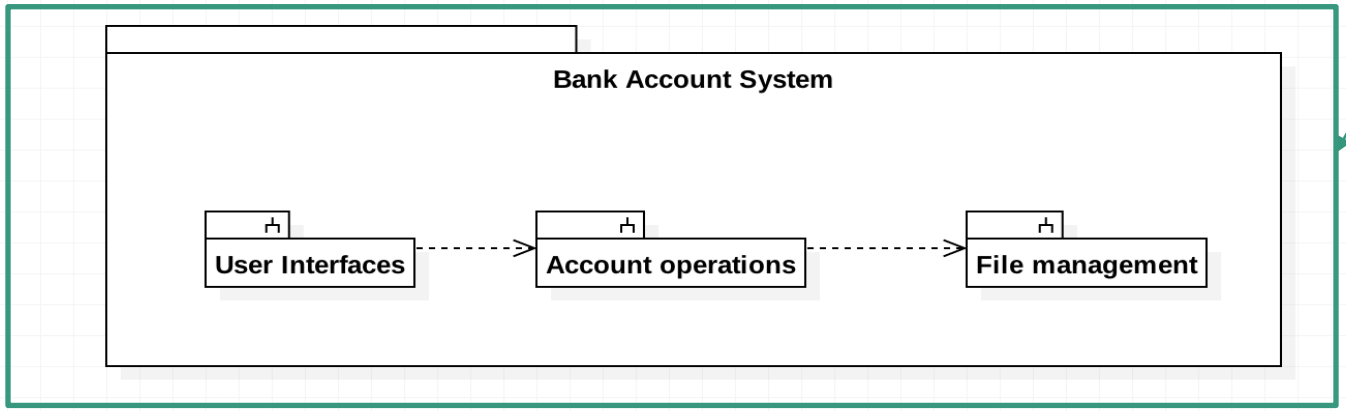
- A UML **package diagram** is a diagram whose primary symbols are package symbols
- Used for representing
  - Static/structural models of *subsystems*, their *parts*, and *relationships*

ARCHITECTURE STRUCTURAL MODEL REPRESENTED WITH UML PACKAGE DIAGRAM – BANK ACCOUNT SYSTEM



Using UML Package Diagram to show sub-systems

Equivalent notations



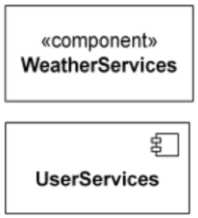
SUBSYSTEMS' RESPONSIBILITIES

Supplement the information in diagram with text description of elements' responsibilities!

Subsystem name	Subsystem Responsibilities
User interfaces	Reads input from users (from computer for teller or from ATM pinpad from customer) and displays systems response either on a computer (for teller), or on the ATM screen (for customer)
Account operations	Performs all operations for creating, deleting, or modifying an account
File management	Performs all operations at the file level, on records that store account information

UML COMPONENT DIAGRAM

- A UML **component** is a modular, replaceable unit with well-defined interfaces.
  - Component symbols are rectangles containing name
  - Stereotyped «component» or
    has component symbol in the upper right-hand corn

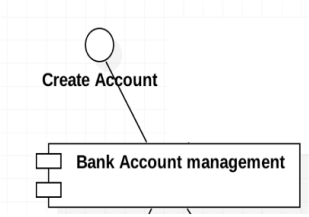


- A UML **component diagram** shows *components*, their *relationships* to their environment, and may also show their internal structure (sub-components)
- A UML **interface** is a named collection of public attributes and abstract operations
  - Represented by special *ball and socket* symbol
  - May also be represented by a stereotyped class symbol.

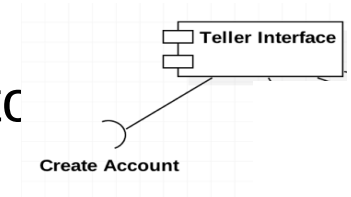
https://www.uml-diagrams.org/component.html

UML COMPONENT DIAGRAM

- A component *provides* or *realizes* an interface when it *implements the interface's operations*
  - It may also include the interface's attributes
- *Provided* interface
  - What a component provides to other components
    - In other words, it is *realized* by a class or component
  - Represented by a ball or *lollipop* symbol
- *Required* interface
  - Needed by a class or component
  - Represented by a *socket* symbol
- The *assembly* connector wires interfaces tc (in a diagram)

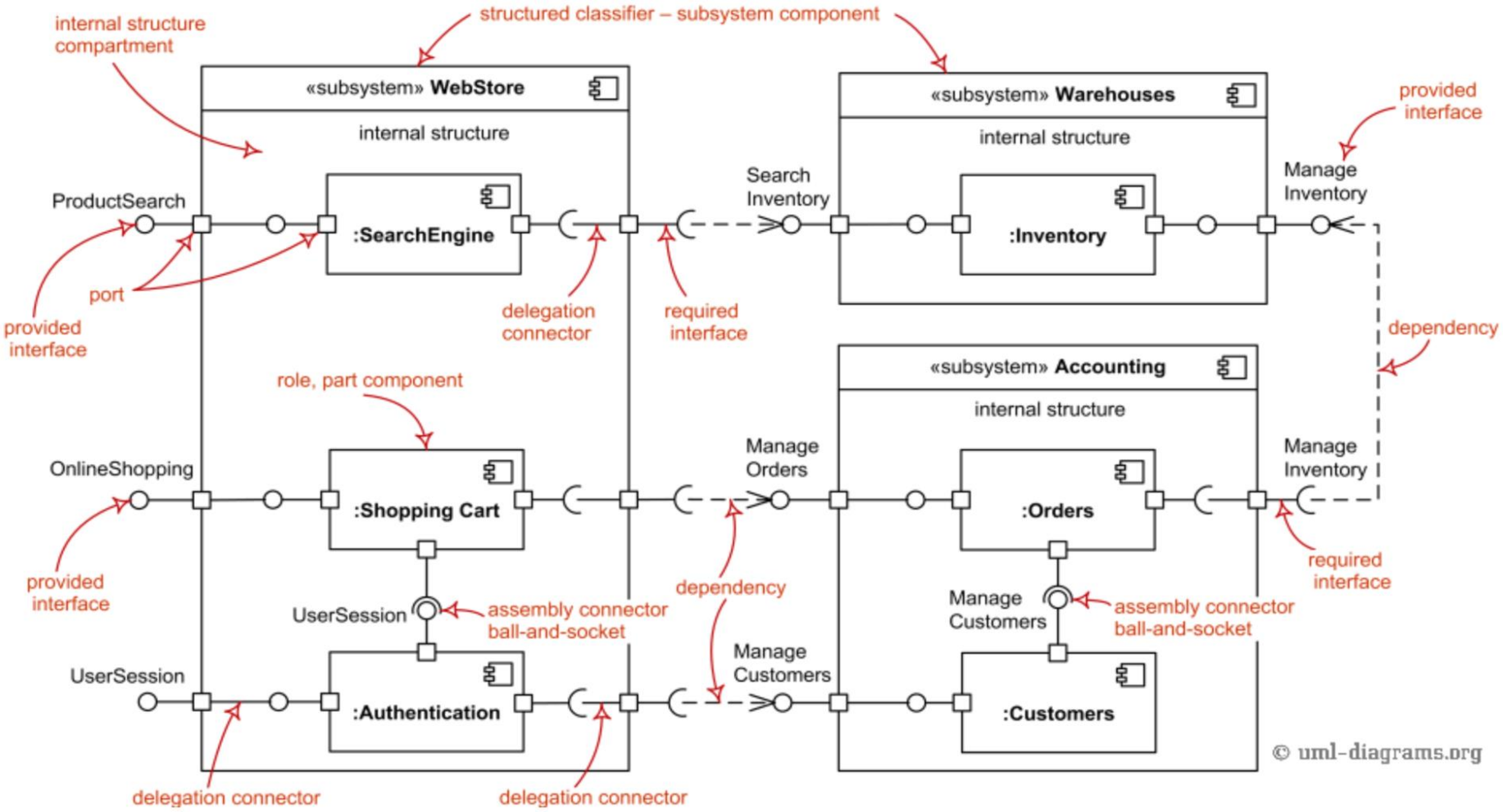


The *Bank Account management* component **provides** (or **realizes**) the *Create Account* interface



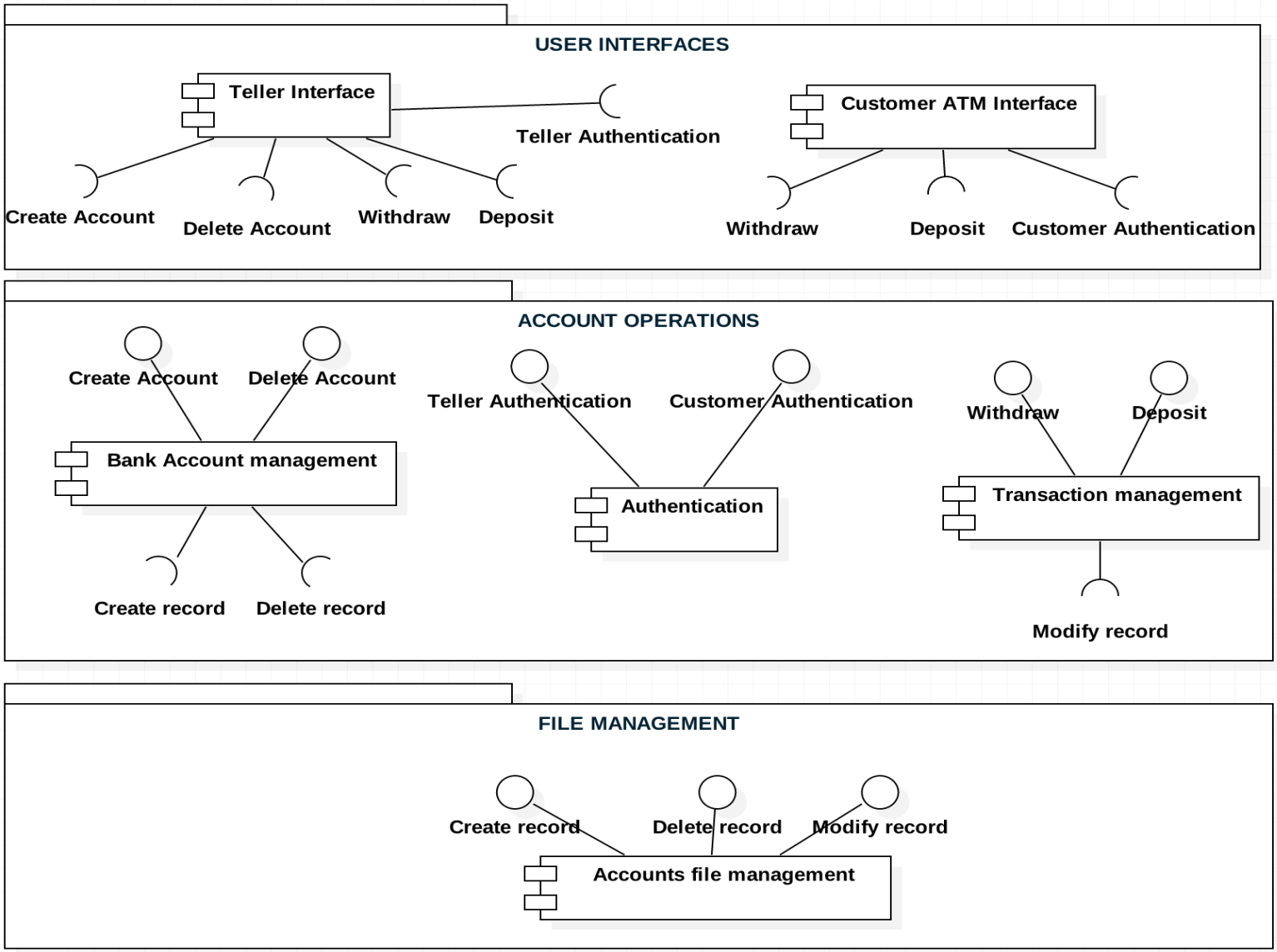
The *Teller Interface* component **requires** (or **uses**) the *Create Account* interface

EXAMPLE –SW ARCHITECTURE STRUCTURE USING UML COMPONENT DIAGRAM - *ONLINE SHOPPING SYSTEM*



The major elements of UML component diagram - component, provided interface, required interface, port, connectors.

EXAMPLE ARCHITECTURAL STRUCTURE USING UML COMPONENT DIAGRAM – *BANK ACCOUNT SYSTEM*



Using UML packages to represent subsystems

OUTLINE

- Lecture
  - Documenting architecture
    - Architecture Views
    - Notations for documenting architecture
    - Documenting components and interfaces
    - Design decisions and their documentation
- Assignments

We are here

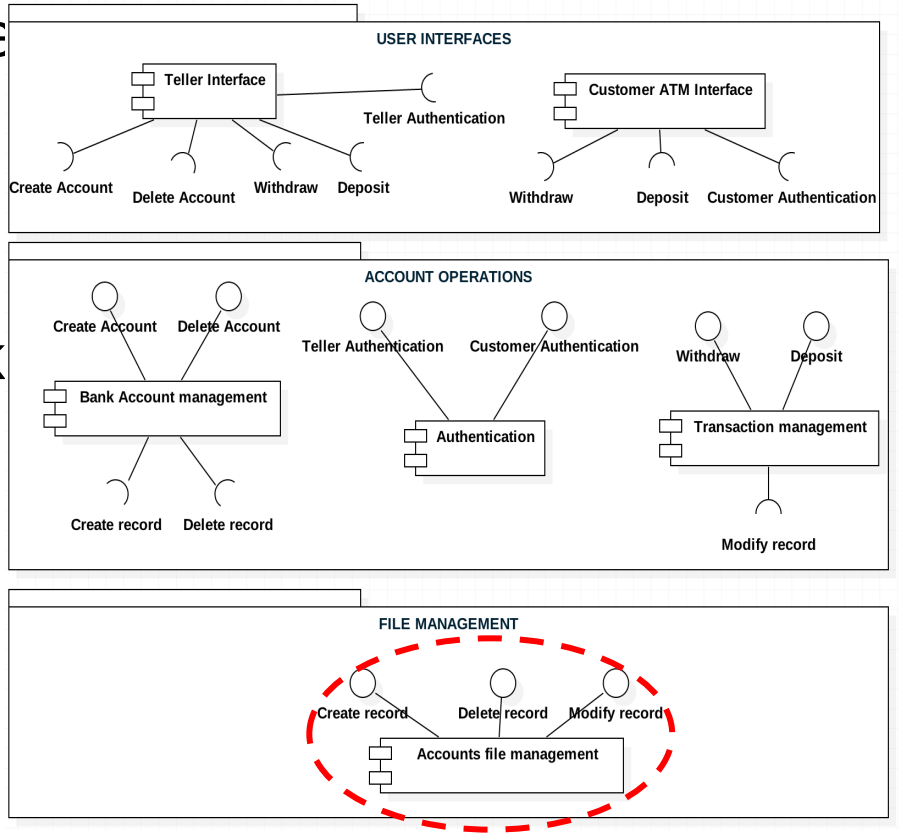
COMPONENTS DESCRIPTION TEMPLATE

Supplement the diagrams with other information for component and interfaces – use text and a template

- Component name: *Component Name*
- Component Responsibilities: *Description of responsibilities*
- Provided Interfaces: *List all provided interfaces (names)*
- Required Interfaces: *List all required interfaces (names)*

COMPONENT DESCRIPTION EXAMPLE

- **Component name**: Accounts file management
- **Component Responsibilities**: Performs all operations on the repository that contains the bank accounts
- **Provided Interfaces**: Create record, Delete record, Modify record
- **Required interfaces**: None



INTERFACE DESCRIPTION TEMPLATE

For each interface of a component specify:

- Interface name: *Interface Name*
- Interface responsibilities: *Description of the interface*
- Interface operations: *List of operations for this interface*

OPERATION DESCRIPTION TEMPLATE

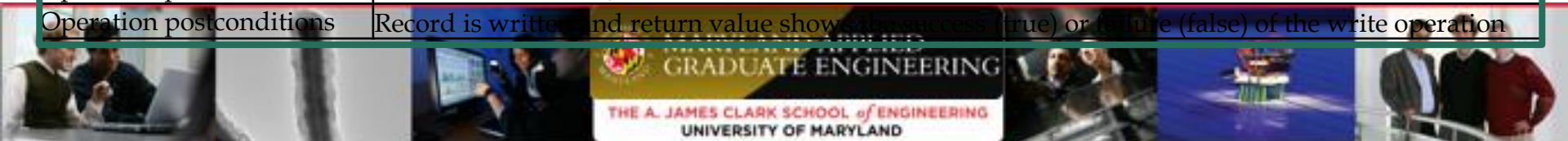
For each operation associated with an interface, specify:

- Operation name
- Operation signature
- Operation description
- Operation preconditions
- Operation postconditions

INTERFACE AND OPERATIONS DESCRIPTION EXAMPLE

- **Interface name**: Create record
- **Interface responsibilities**: Creates records in the file
- **Interface Operations**: OpenFile, CheckRecordExists, Write
- **Operations description**: *in table below*

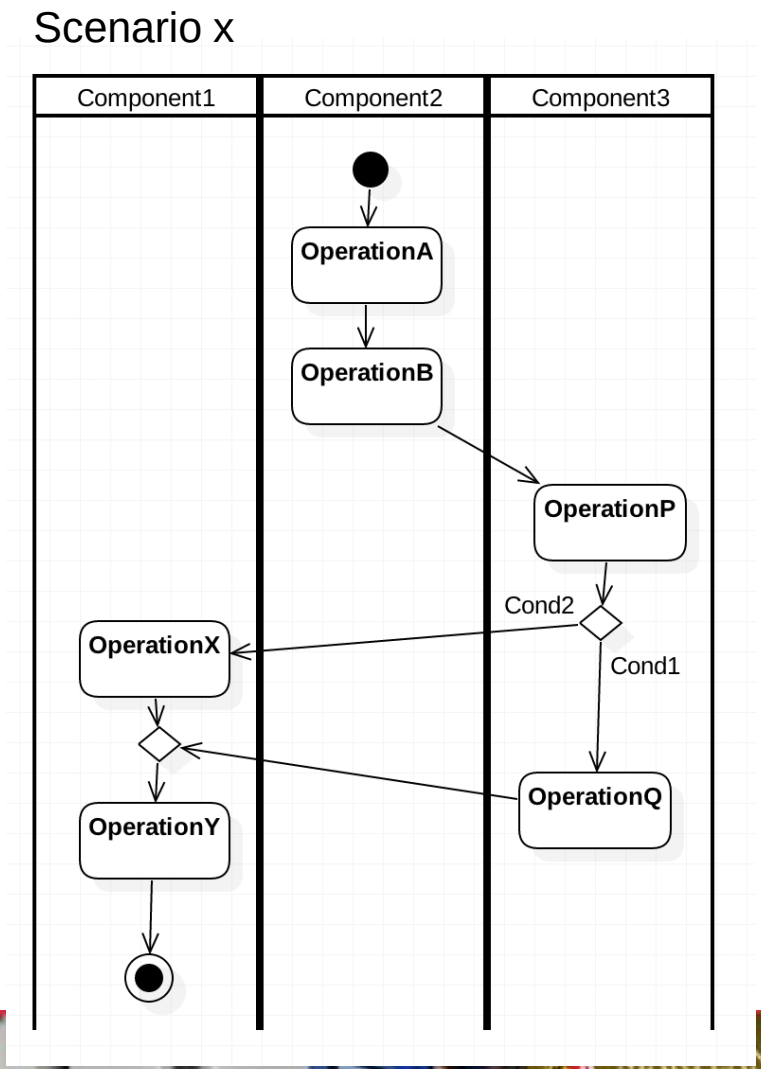
Open File	
Operation signature	**OpenFile (FileName: String):Boolean**
Operation description	Opens the file with the name FileName and returns true or false, depending on the success of file opening
Operation preconditions	FileName not null
Operation postconditions	File is open, or False is returned
CheckRecordExists	
Operation signature	**CheckRecordExists (FileName: String, AccountNumber:Int):Boolean**
Operation description	Checks if the record with AccountNumber exists in the file FileName
Operation preconditions	FileName is not null, AccountNumber is valid (integer >=11111 and =<99999)
Operation postconditions	Returns true if the record already exists, and false if it doesn't
Write	
Operation signature	**Write (FileName: String, Account:Account):Boolean**
Operation description	Writes a record of type Account to the FileName
Operation preconditions	FileName is not null, and NewAccount is in a valid format
Operation postconditions	Record is written and return value shows the success (true) or failure (false) of the write operation



IMPORTANCE OF WELL DEFINED AND WELL DOCUMENTED INTERFACES

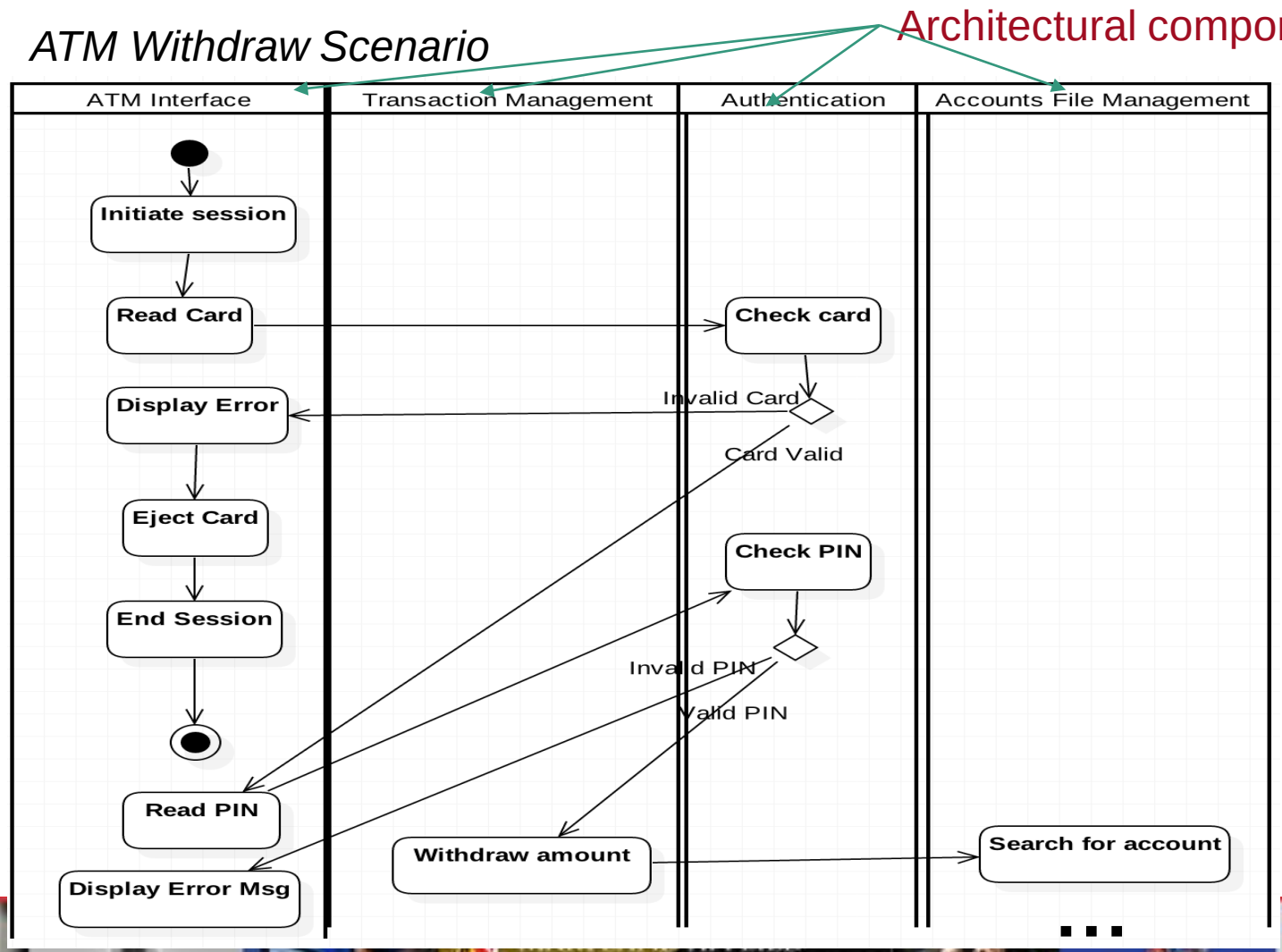
- Avoid interface mismatch

ARCHITECTURE BEHAVIOR MODEL DESCRIPTION - USING UML ACTIVITY DIAGRAM



- Partitions/Swimlanes are used for representing ***architectural components***
  - One swimlane for each architectural component involved in realizing the behavior for the given scenario
  - Actions are used to represent component operations behavior
- **Not** to be confused with the use of UML activity diagram notation in requirements, for representing *use case scenarios* or *abuse case scenarios*, where swimlanes represent actors or "the system" and their actions

ARCHITECTURE BEHAVIOR MODEL REPRESENTATION USING UML ACTIVITY DIAGRAM – EXAMPLE *BANK SYSTEM*

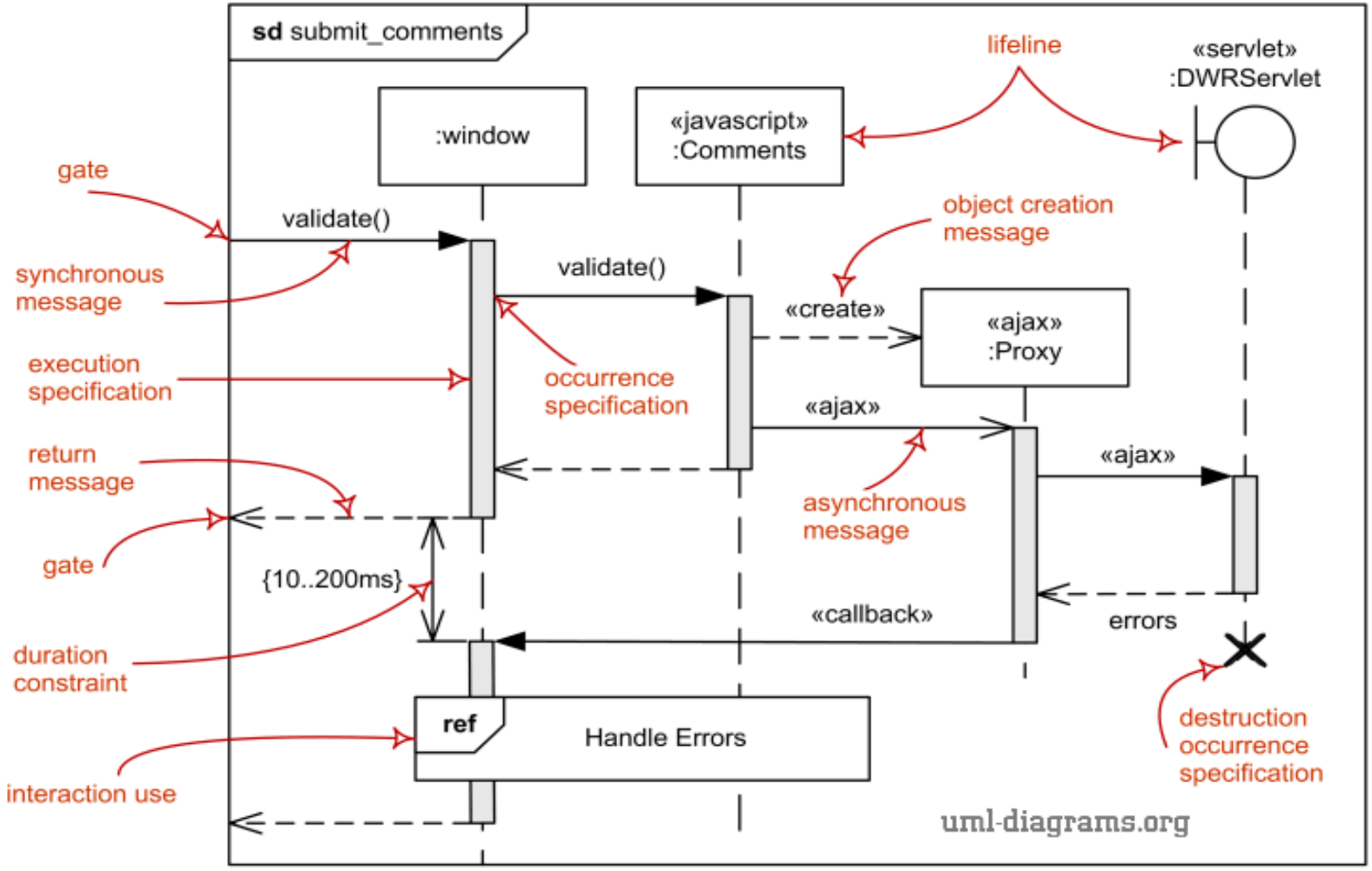


UML SEQUENCE DIAGRAM

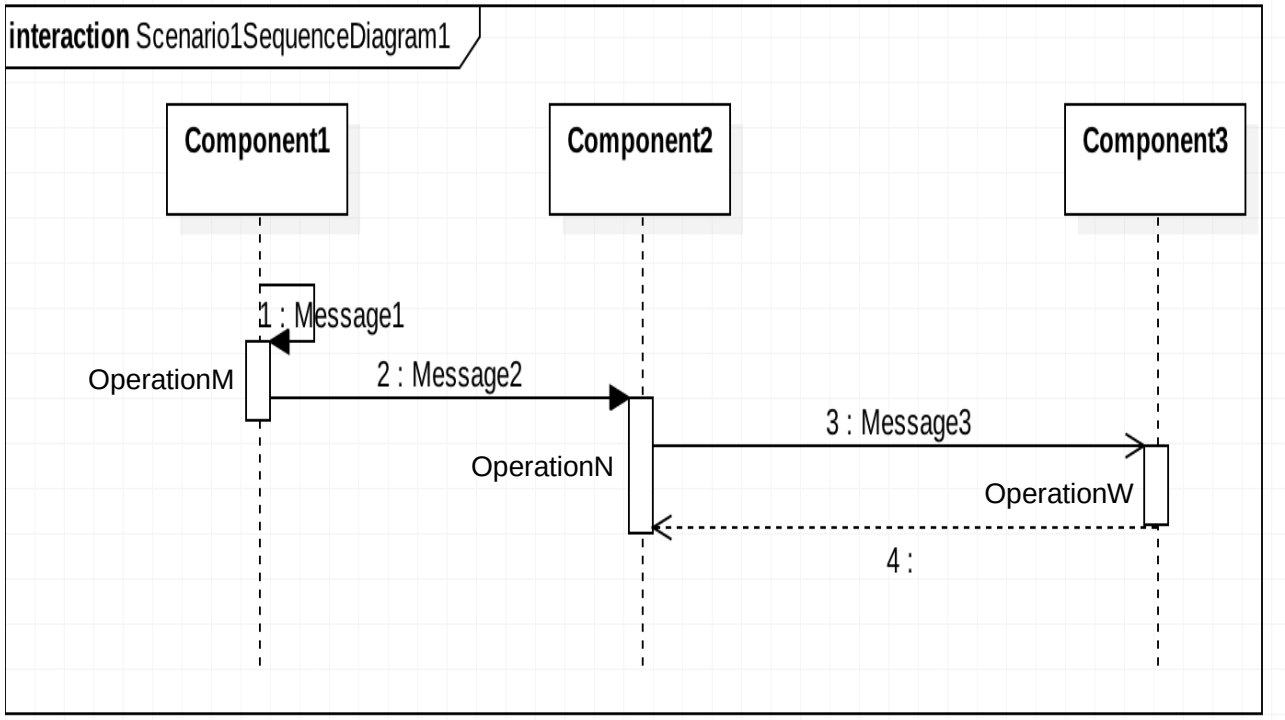
- A kind of interaction diagram, which focuses on the *message* interchange between a number of *lifelines*.
- Sequence diagram describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines.
- Elements typically contained in a UML sequence diagram: lifeline, execution specification, message, combined fragment, interaction use, state invariant, continuation, destruction occurrence.
- An interaction (combined) fragment is a marked part of an interaction specification that shows Branching, Loops, or Concurrent execution

- See Chapter 12 of the textbook

UML SEQUENCE DIAGRAM



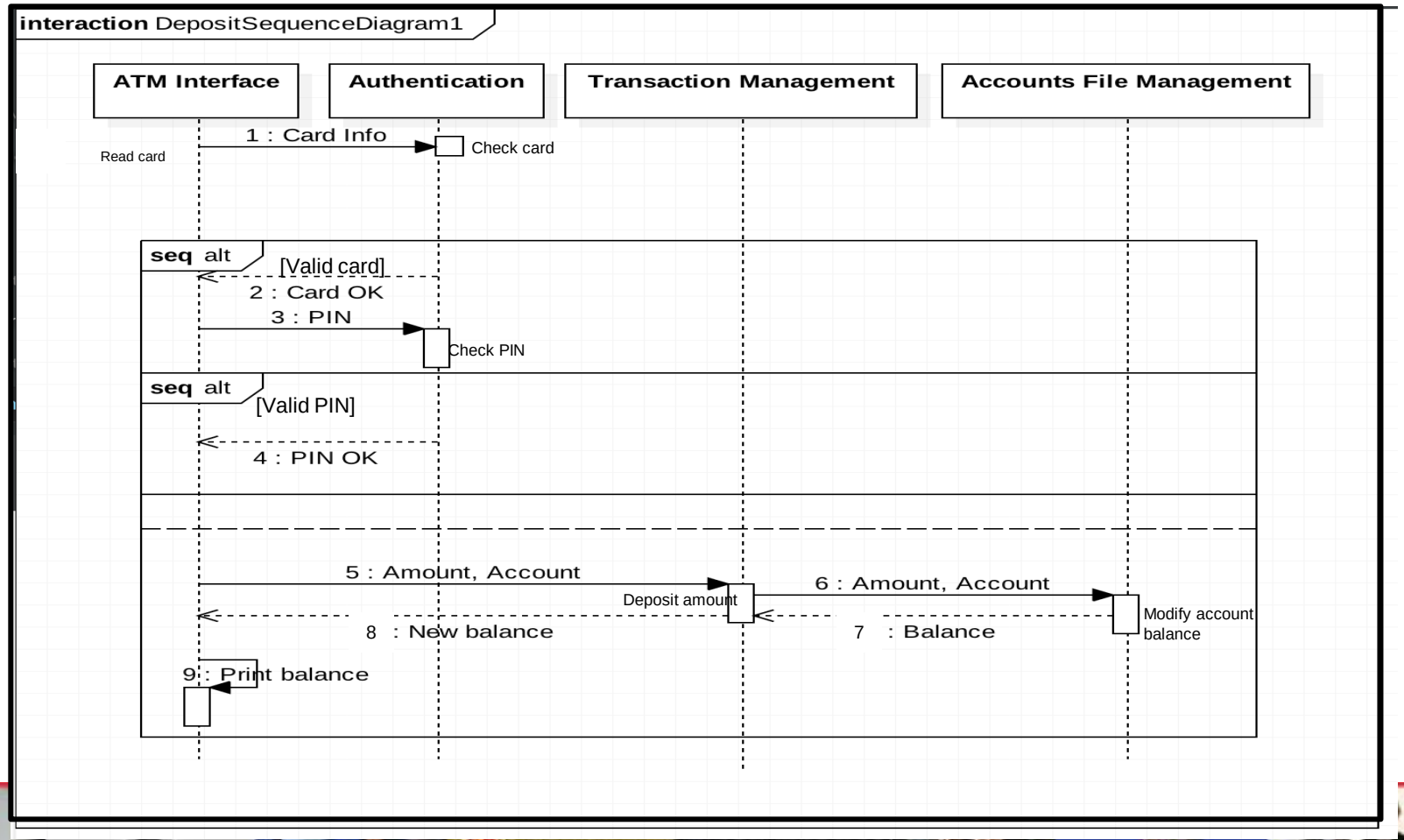
ARCHITECTURE BEHAVIOR MODEL REPRESENTATION USING UML SEQUENCE DIAGRAM



- Lifelines are used to represent ***architectural components***
  - One lifeline for each architectural component involved in realizing the behavior for the given scenario

ARCHITECTURE BEHAVIOR USING UML SEQUENCE DIAGRAM – EXAMPLE *BANK SYSTEM*

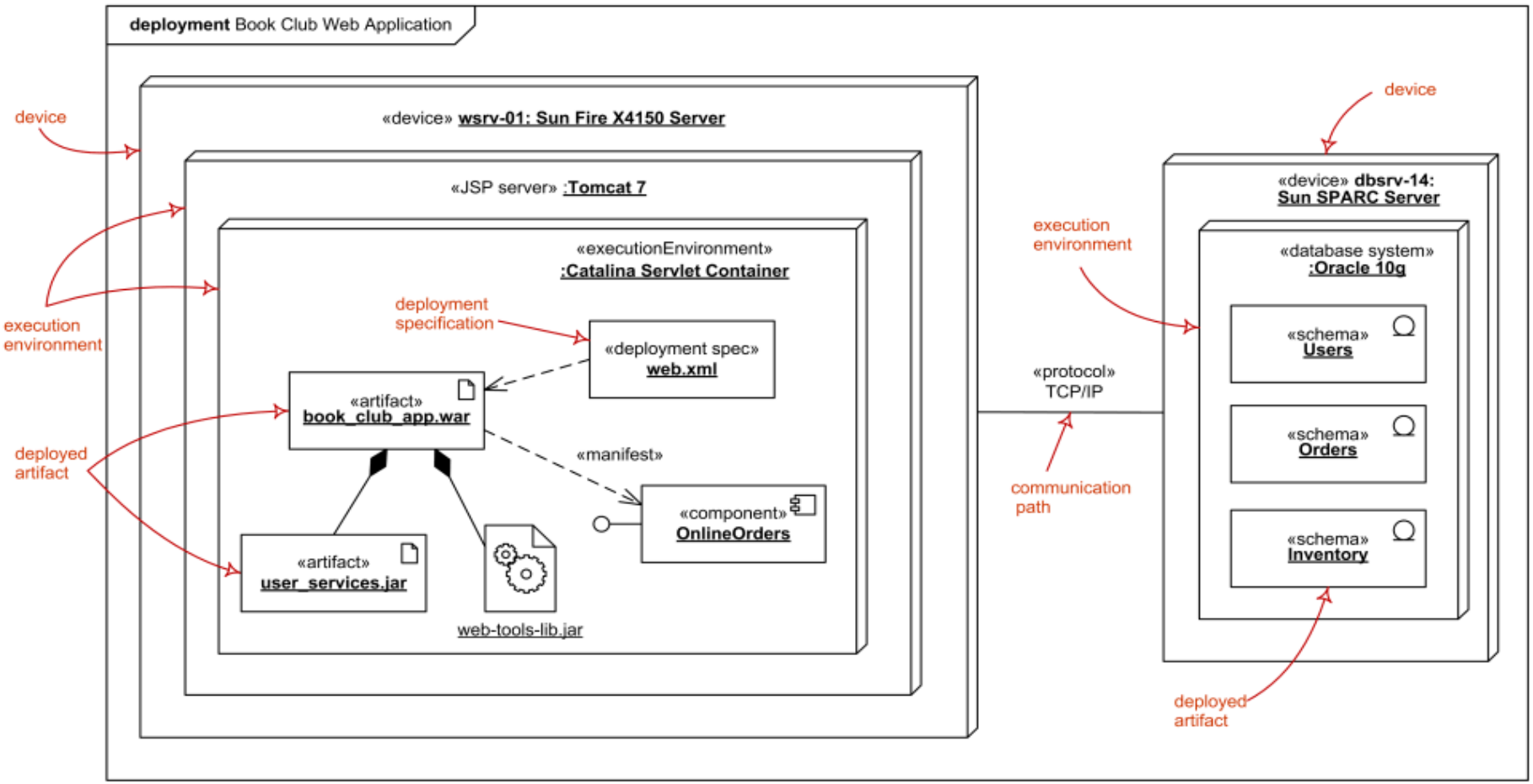
ATM Deposit Scenario



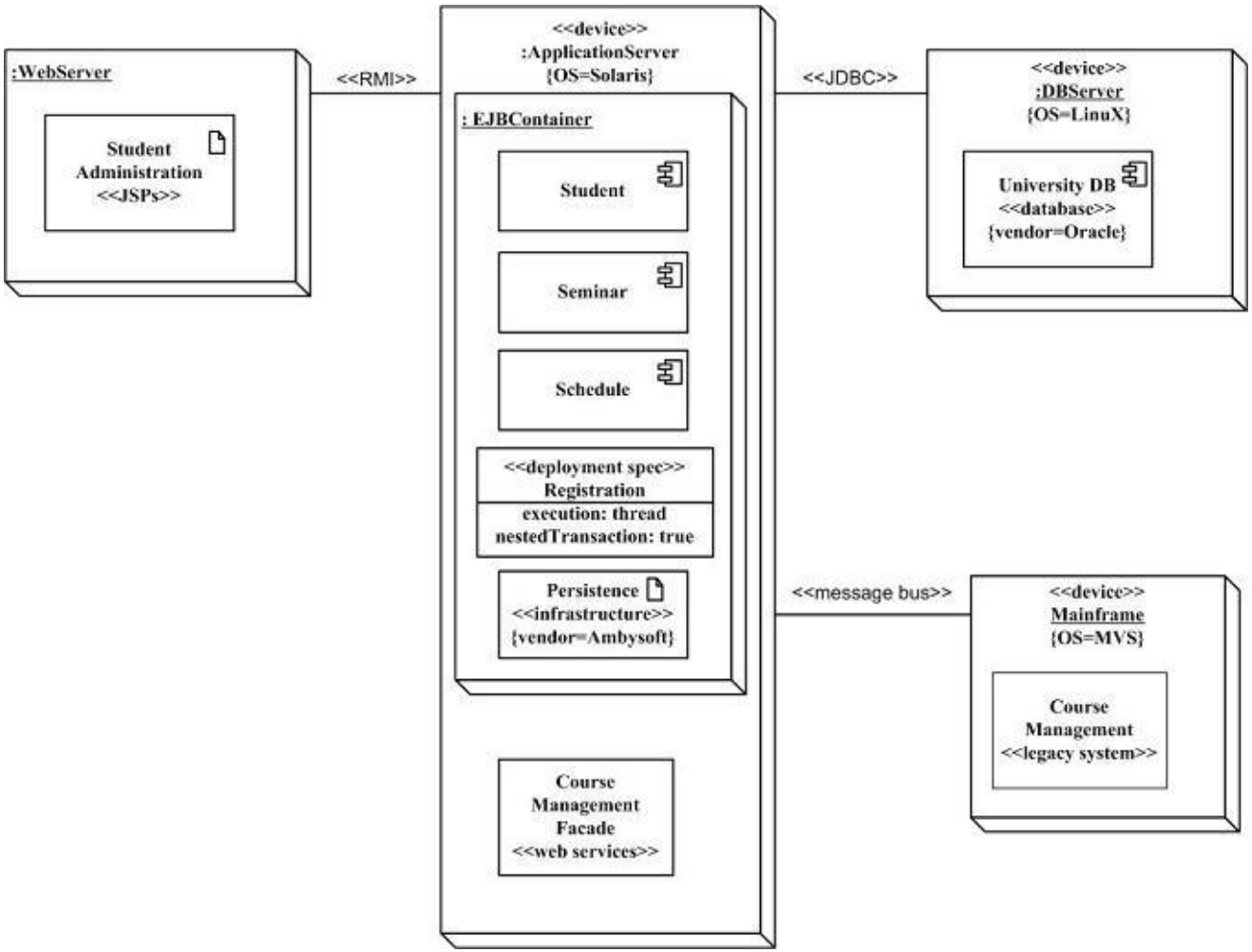
UML DEPLOYMENT DIAGRAM

- A UML **deployment diagram** models *computational resources*, communication paths among them, and *artifacts* that reside and execute on them.
- Used to show
  - Real and virtual machines used in a system
  - Communication paths between machines
  - Program and data files realizing the system
    - Residence
    - Execution
- Computational resources are *nodes*
- Communication paths are solid lines between nodes
  - May be labeled
  - May have multiplicities and role names
- *Artifact* symbols may
  - Appear within node symbols
  - Be listed within node symbols
  - Be connected to node symbols by dependency arrows stereotyped with «deploy»

EXAMPLE *DEPLOYMENT VIEW* USING A UML DEPLOYMENT DIAGRAM – *BOOK CLUB SW*



EXAMPLE DEPLOYMENT VIEW USING A UML DEPLOYMENT DIAGRAM – *UNIVERSITY INFORMATION SYSTEM*



http://www.agilemodeling.com/artifacts/deploymentDiagram.htm

OUTLINE

- Lecture
  - Documenting architecture
    - Architecture Views
    - Notations for documenting architecture
    - Documenting components and interfaces
    - Design decisions and their documentation
- Assignments

We are here

DESIGN DECISIONS

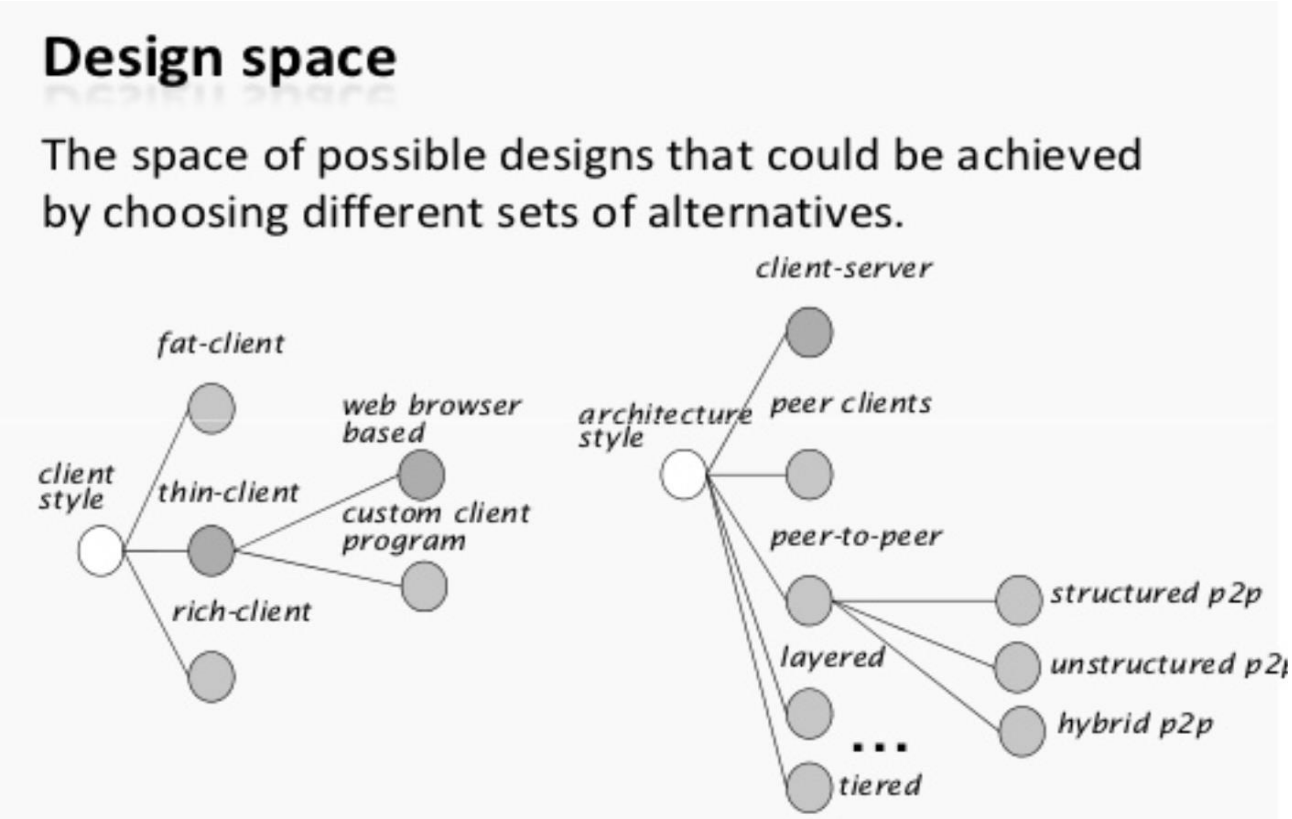
- Design is a decision making process
  - Consists of a series of decisions
- Designer:
  - Faces **questions/issues**
  - Finds **answers/solutions**
  - **Evaluates/compares solutions** against given **criteria**
  - **Selects** a set of decisions solutions
    - Decisions can depend on other decisions
  - The design reflects the set of all decisions
  - Documents the solution (decisions) and the rationale

EXAMPLE OF SW DESIGN QUESTIONS

- What technology to use?
- How to decompose the SW in components?
- Can I use an architectural style, and which one is appropriate?
- What are the control and components' communication mechanisms?
- What tactics and mechanisms to use to achieve quality attributes?
- How to distribute SW on HW?
- How to assign work to people?

EXAMPLE DESIGN DECISIONS

- Decomposition/composition, modularization, control, connectors, styles, tactics…



From: http://www.slideshare.net/henry.muccini/software-architecture-design-decisions

DESIGN DECISIONS DOCUMENTATION

- Make *implicit* knowledge and decisions *explicit*
- *WHY*?
  - So design decisions will be followed in implementation
    - Better understanding of decision rationale
  - Support reuse and change
    - Traced to requirements and constraints (rationale)
    - Traced to code
    - Obsolete decisions can be removed/changed
  - Support knowledge preservation and transfer

Design Decisions: The Bridge between Rationale and Architecture, by J.S. van der Ven, A.G. J. Jansen, J.A. G. Nijhuis, J. Bosch , Chapter in *Rationale Management in Software Engineering* pp 329-348

SUMMARY

- Multiple views are needed to document software architecture
- Views reflect different stakeholder's concerns
- Views must be consistent with each other
- There are different sets of views
- Various notations can be used for architecture representation – have pros and cons
- UML diagrams can be used for architecture representation
  - Make distinction between **what** we represent (a model) and **how**/the notation used to represent it (the UML diagram)
  - Talk in terms of models, not notations

OUTLINE

- Lecture
  - Documenting architecture
    - Architecture Views
    - Notations for documenting architecture
    - Documenting components and interfaces
    - Design decisions and their documentation
- Assignments

We are here

QUIZ ARCHITECTURE DOCUMENTATION

- Demonstrate knowledge and understanding of architecture documentation
- Individual assignment, online

SOFTWARE DEVELOPMENT PROJECT ASSIGNMENT

- Tasks:
  - Start documenting your software architecture, using views and appropriate UML notations

- Team assignment – only one team member needs to submit on behalf of the team
- Online submission, submit to *Architecture Views* assignment

SOFTWARE DEVELOPMENT PROJECT ASSIGNMENT

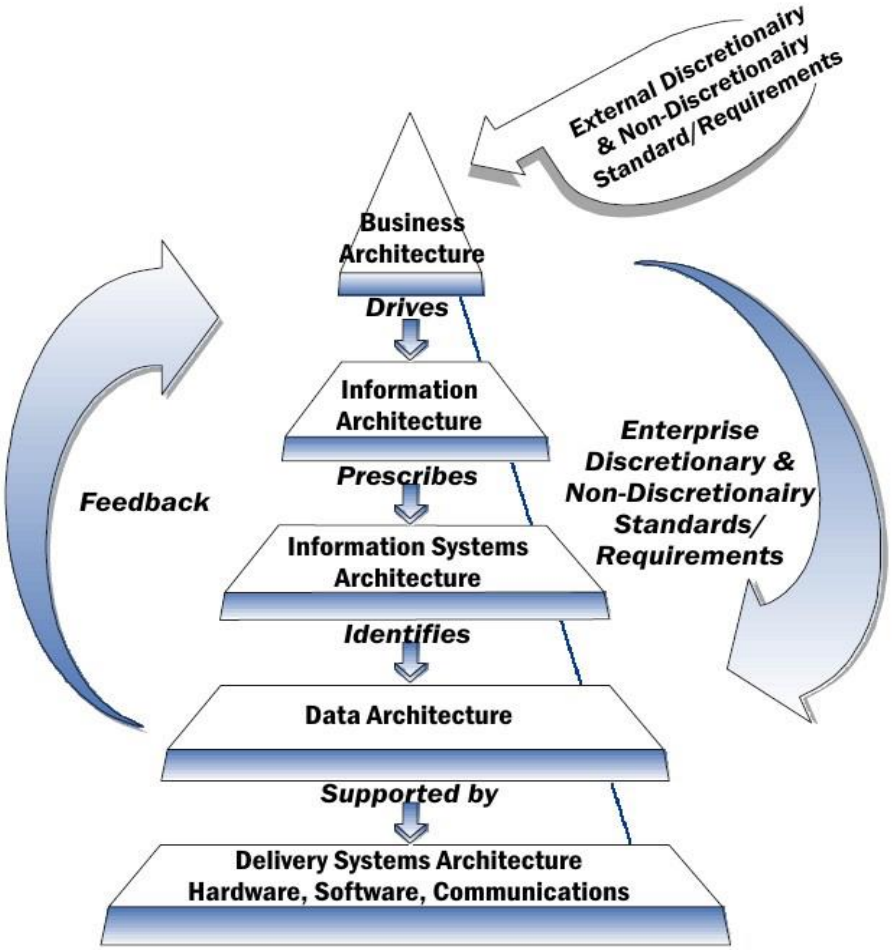
- **Team Presentation of the preliminary architecture**
  - Present your logical, behavior, information, deployment, requirements/ features to architecture allocation, and architecture work allocation
  - Presentation + 5 minutes Q&A/Discussion
  - Purpose: for teams to get early feedback on the architectural design solution and its documentation

BONUS MATERIAL: ENTERPRISE ARCHITECTURE (EA) DOCUMENTATION FRAMEWORKS

SYSTEM AND ENTERPRISE ARCHITECTURE

- Systems architecture
  - Systems engineering
  - A *system* can include software, hardware, networks, etc. (e.g., for a car: electric system, fuel system, climate control system)
  - System of systems (e.g., the whole car)
- Enterprise architecture (EA)
  - Is a well-defined practice for conducting enterprise analysis, design, planning, and implementation, using a holistic approach at all times, for the successful development and execution of strategy.
  - Enterprise architecture applies architecture principles and practices to guide organizations through the business, information, process, and technology changes necessary to execute their strategies. (Wikipedia)
  - *Information technology* /Information system architecture

ENTERPRISE ARCHITECTURE (EA)

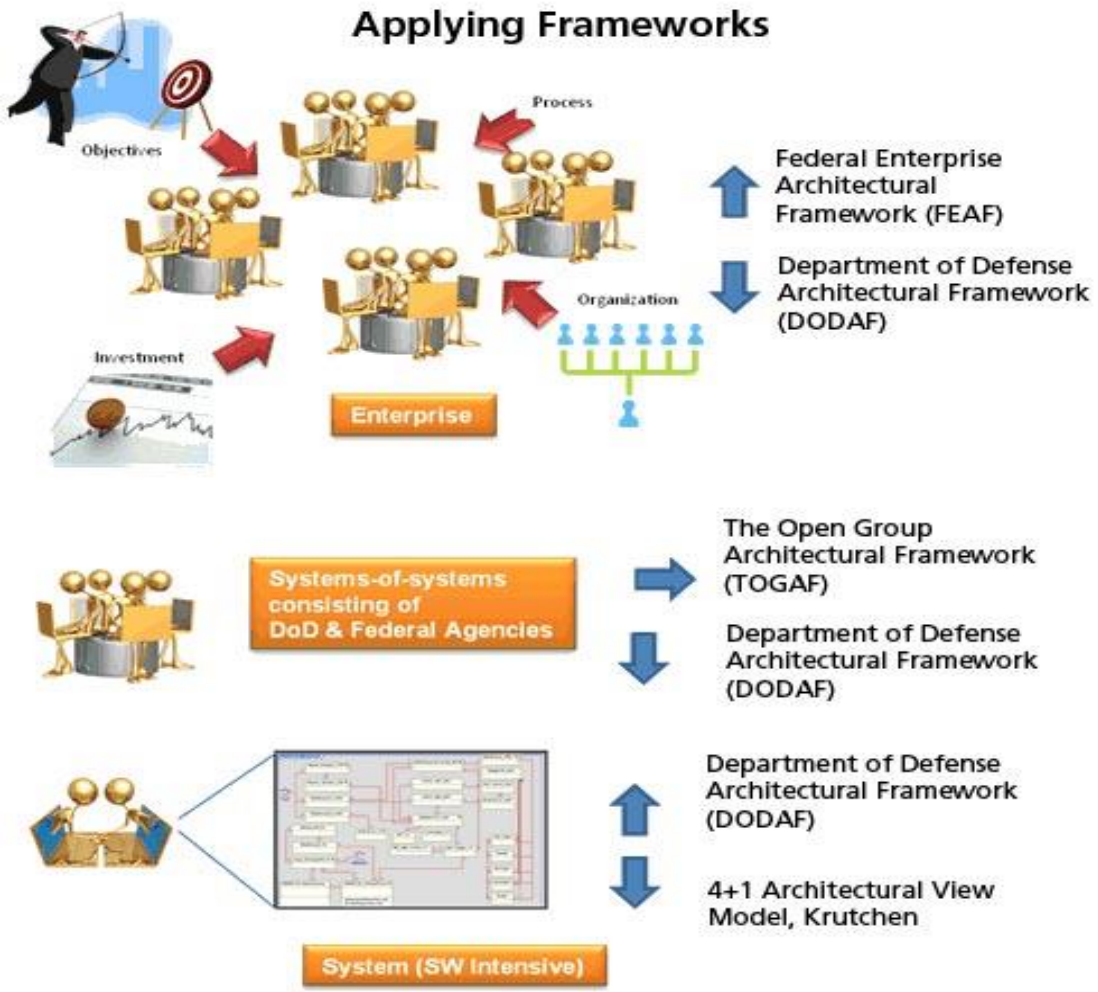


NIST Enterprise Architecture Model (initiated in 1989)

EA FRAMEWORKS

- An EA framework defines how to create and use an enterprise architecture.
  - Provides principles and practices for creating and using the architecture description of a system.
- Enterprise-architectural methodologies (frameworks)
  - The ***Zachman Framework*** for Enterprise Architectures — actually more accurately defined as a taxonomy
  - The ***Open Group Architectural Framework*** (***TOGAF***)— actually more accurately defined as a process
  - The ***Federal Enterprise Architecture—Can*** be viewed as either an implemented enterprise architecture or a prescriptive methodology for creating an enterprise architecture
  - The ***Gartner Methodology*** —Can be best described as an enterprise architectural practice

APPLYING ARCHITECTURE FRAMEWORKS - EXAMPLE



[From: Architectural Frameworks, Models, and Views]

RECOMMENDED READING

- Architectural Frameworks, Models, and Views
- A COMPARISON OF ENTERPRISE ARCHITECTURE FRAMEWORKS, by L. Urbaczewski and S. Mrdalj