CS-3004
SOFTWARE DESIGN AND ANALYSIS

RUBAB JAFFAR
RUBAB.JAFFAR@NU.EDU.PK

4+1 view Model

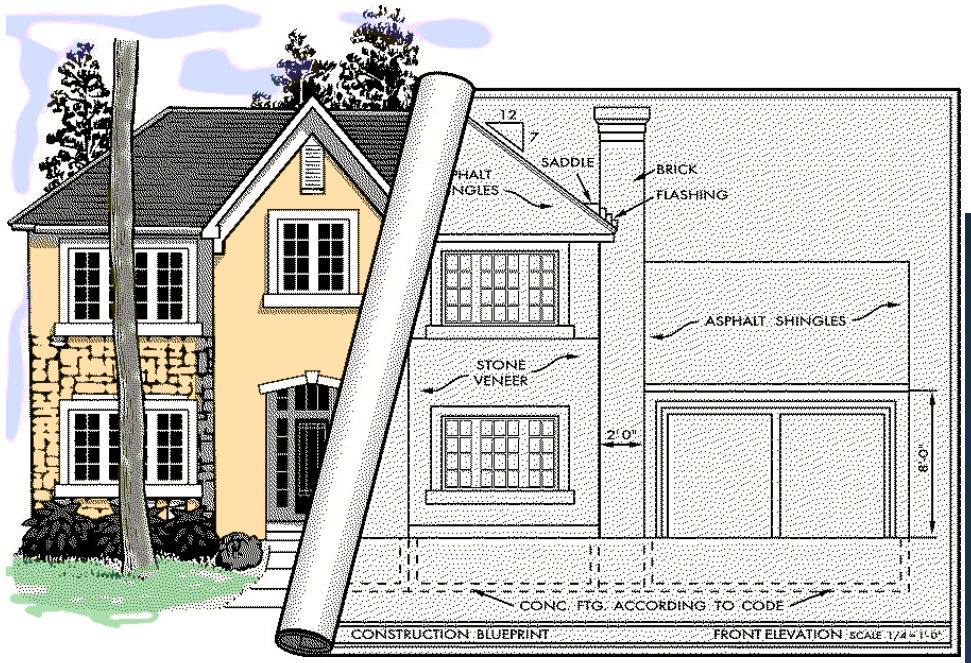
TODAY'S OUTLINE

- System
- Model
- View
- Diagrams
- 4+1 architecture
- Checking the Model
- Why Homogenize?
- Combining Classes
- Splitting Classes
- Eliminating Classes
- Consistency Checking
- Scenario Walk-Through
- Event Tracing
- Documentation Review

WHAT IS A MODEL?

- Models are often built in the context of business and IT systems in order to better understand existing or future systems.
- A model is an abstraction describing a subset of a system
- A useful model has the right level of detail and represents only what is important for the task in hand.
- However, a model never fully corresponds to reality.
- Modeling always means emphasizing and omitting:
  - emphasizing essential details and
  - omitting irrelevant ones.
  - Many things can be modelled: bridges, traffic flow, buildings, economic policy

MODELING A HOUSE



WHY DO WE NEED MODELS?

- Communication between all involved parties:
  - In order to build the right system, all parties think along the same lines. Important that everyone understands the same requirements, that developers understand these requirements, and that the decisions made can still be understood months later.
- Visualization of all facts:
  - All accumulated facts relevant to the system need to be presented in such a way that everyone concerned can understand them.
- Verification of facts in terms of completeness, consistency, and correctness: In particular, the clear depiction of interrelationships makes it possible to ask specific questions, and to answer them.

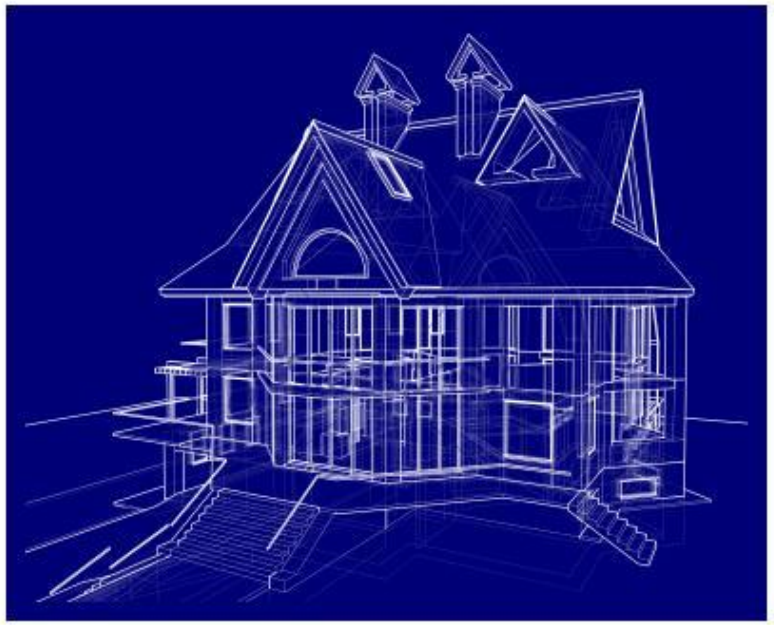
DIAGRAMS VS MODELS

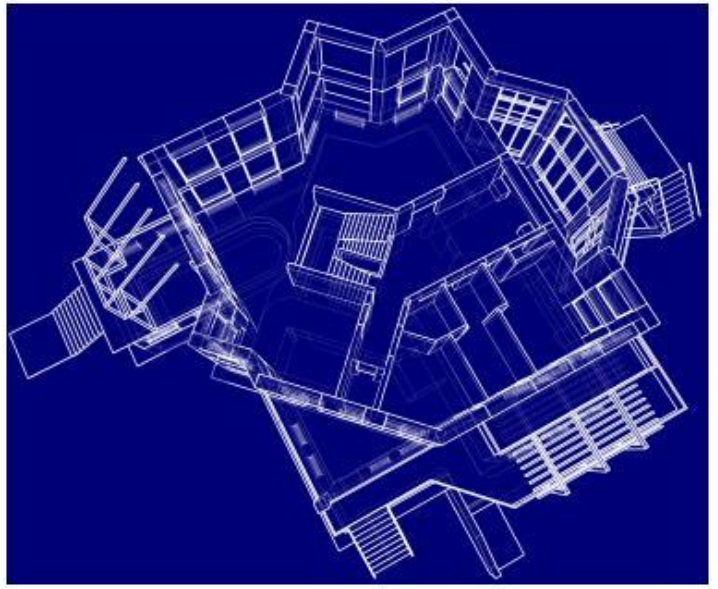
- A diagram illustrates some aspect of a system.
- A model provides a complete view of a system at a particular stage and from a particular perspective, e.g., Analysis model, Design model.
- A model may consist of a single diagram, but most consist of many related diagrams and supporting data and documentation.

VIEW

- The more information a model gives, the more complex and difficult it becomes.
- A system has many different aspects:
  - functional (its static structure and dynamic interactions),
  - nonfunctional (timing requirements, reliability, deployment, and so on),
  - organizational aspects (work organization, mapping to code modules, and so on).
- A system description requires a number of views, where each view represents a projection of the complete system that shows a particular aspect.
- Different views are formed of the objects. These views are interconnected in many ways.

VIEWS





UML VIEWS

- Structural classification
- Dynamic behaviour
- Physical layout
- Model management

UML VIEWS AND DIAGRAMS

Major View	Diagram	Concepts
structural	class diagram	association, class, dependency, generalization, interface, realization
	internal structure collaboration diagram component diagram	connector, interface, part, port, provided interface, role, required interface
		connector, collaboration, collaboration use, role
		component, dependency, port, provided interface, realization, required interface, subsystem
	use case diagram	actor, association, extend, include, use case, generalization

UML VIEWS AND DIAGRAMS CONT.

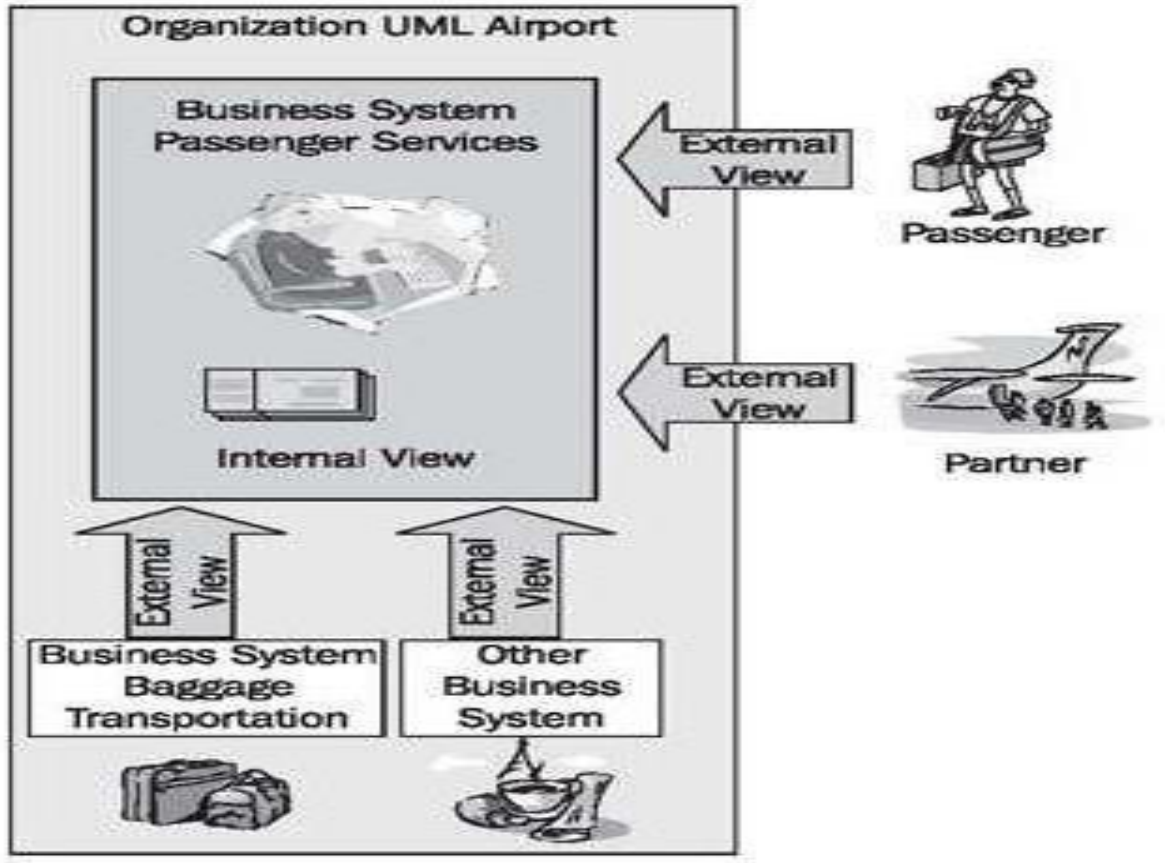
Major View	Diagram	Concepts
dynamic	state machine diagram	completion transition, do activity, effect, event, region, state, transition, trigger
	activity diagram	action, activity, control flow, control node, data flow, exception, expansion region, fork, join, object node, pin
	sequence diagram communication diagram	occurrence specification, execution specification, interaction, lifeline, message, signal
		collaboration, guard condition, message, role, sequence number

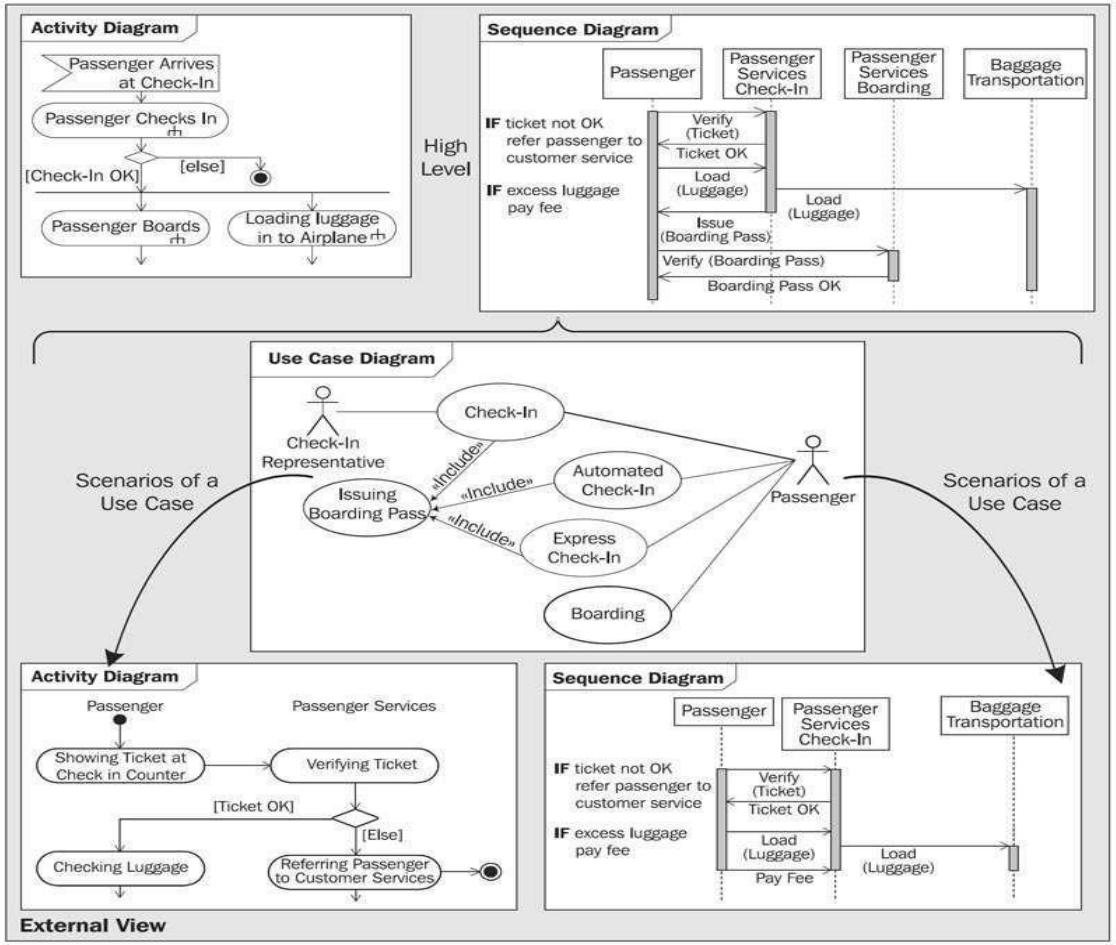
UML VIEWS AND DIAGRAMS CONT.

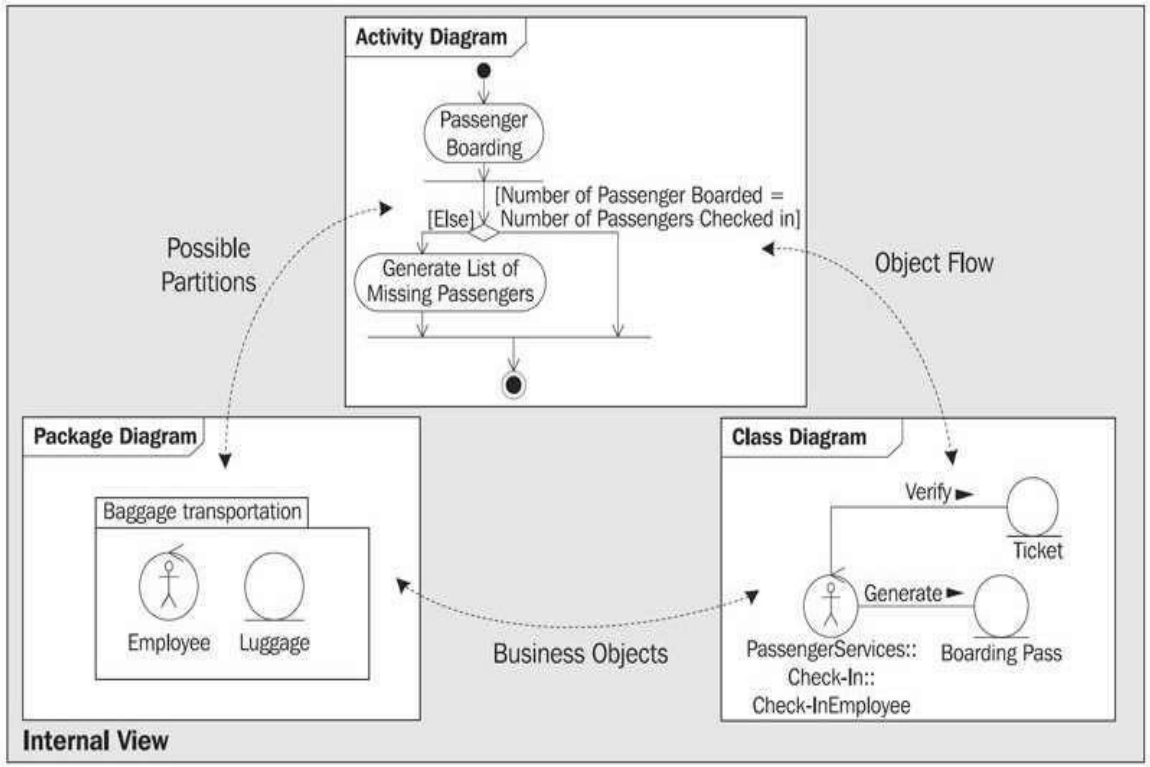
Major View	Diagram	Concepts
physical	deployment diagram	artifact, dependency, manifestation, node
model management	package diagram	import, model, package
	package diagram	constraint, profile, stereotype, tagged value

ONE MODEL—TWO VIEWS

- External View
- Internal View

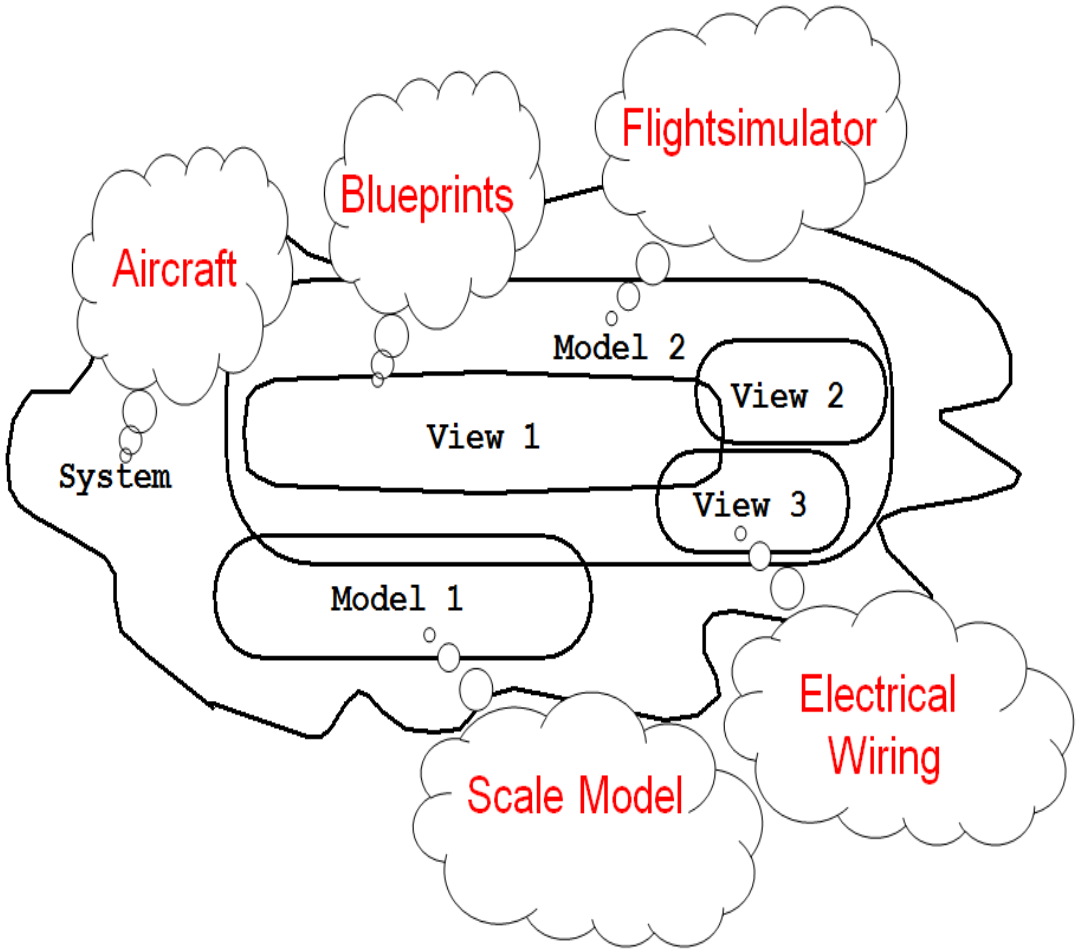



Activity Diagram
Passenger Arrives at Check-In
Passenger Checks In
[Check-In OK]
[else]
Passenger Boards
Loading luggage in to Airplane
High Level
Sequence Diagram
Passenger
Passenger Services Check-In
Passenger Services Boarding
Baggage Transportation
IF ticket not OK refer passenger to customer service
IF excess luggage pay fee
Verify (Ticket)
Ticket OK
Load (Luggage)
Issue (Boarding Pass)
Verify (Boarding Pass)
Boarding Pass OK
Use Case Diagram
Check-In Representative
Check-In
Issuing Boarding Pass
Automated Check-In
Express Check-In
Boarding
«Include»
Passenger
Scenarios of a Use Case
Scenarios of a Use Case
Activity Diagram
Passenger
Passenger Services
Showing Ticket at Check in Counter
Verifying Ticket
[Ticket OK]
[Else]
Checking Luggage
Referring Passenger to Customer Services
Sequence Diagram
Passenger
Passenger Services Check-In
Baggage Transportation
IF ticket not OK refer passenger to customer service
IF excess luggage pay fee
Verify (Ticket)
Ticket OK
Load (Luggage)
Pay Fee
Load (Luggage)
External View


Activity Diagram
Passenger Boarding
[Number of Passenger Boarded = Number of Passengers Checked in]
[Else]
Generate List of Missing Passengers
Possible Partitions
Object Flow
Package Diagram
Baggage transportation
Employee
Luggage
Business Objects
Class Diagram
Verify
Ticket
Generate
Boarding Pass
PassengerServices::
Check-In::
Check-InEmployee
Internal View

SYSTEMS, MODELS AND VIEWS

- A model is an abstraction describing a subset of a system
- A view depicts selected aspects of a model
- A notation is a set of graphical or textual rules for depicting views
- Views and models of a single system may overlap each other
- Examples:
- System: Aircraft
- Models: Flight simulator, scale model
- Views: All blueprints, electrical wiring, fuel system



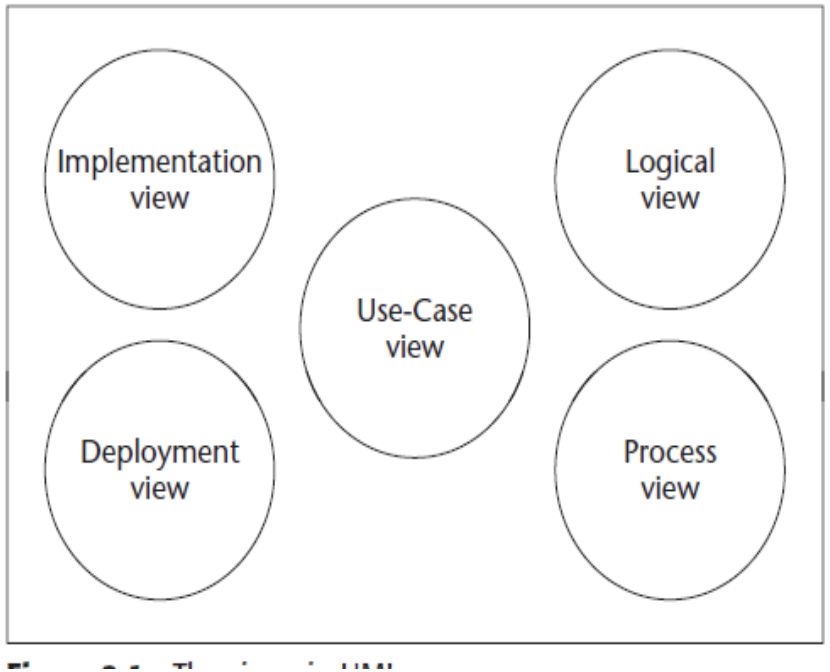
WHAT IS VIEW MODEL ?

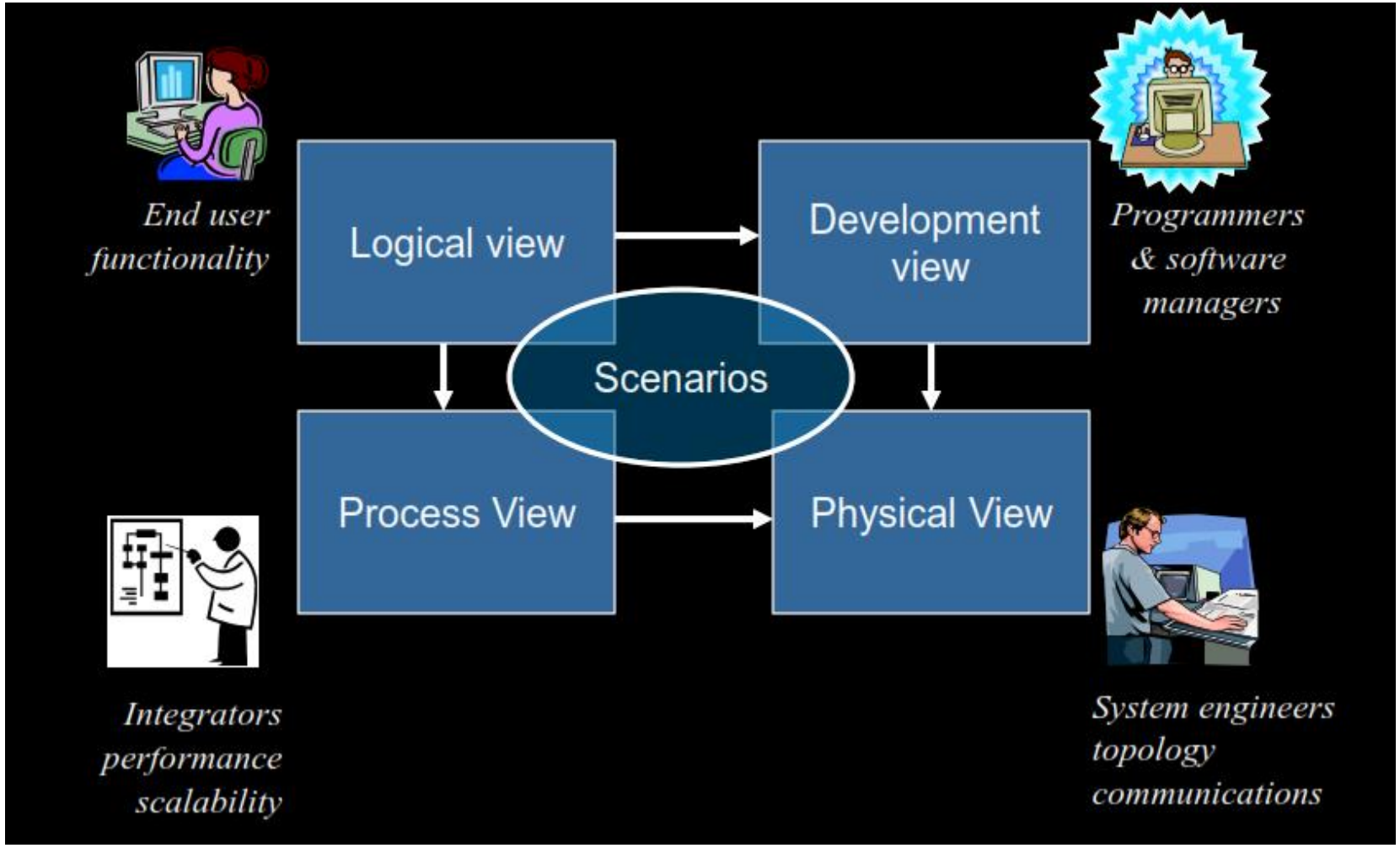
- A view model in systems engineering or software engineering is a framework.
- It defines a coherent set of views to be used in the construction of a system architecture or software architecture.
- A view is a representation of a whole system from the perspective of a related set of concerns.
- Viewpoint modeling has become an effective approach for dealing with the inherent complexity of large distributed systems.

INTENT OF 4+1 VIEW MODEL

- To come up with a mechanism to separate the different aspects of a software system into different views of the system.
- But why???? -> Different stakeholders always have different interest in a software system.
- DEVELOPERS – Aspects of Systems like classes
- SYSTEM ADMINISTRATOR – Deployment, hardware and network configuration.
- Similar points can be made for Testers, Project Managers and Customers.

THE VIEWS IN UML





HOW MANY VIEWS?

- Views should to fit the context
  - Not all systems require all views
  - Single processor: drop deployment view
  - Single process: drop process view
  - Very small program: drop implementation view
- A system might need additional views
  - Data view, security view, …

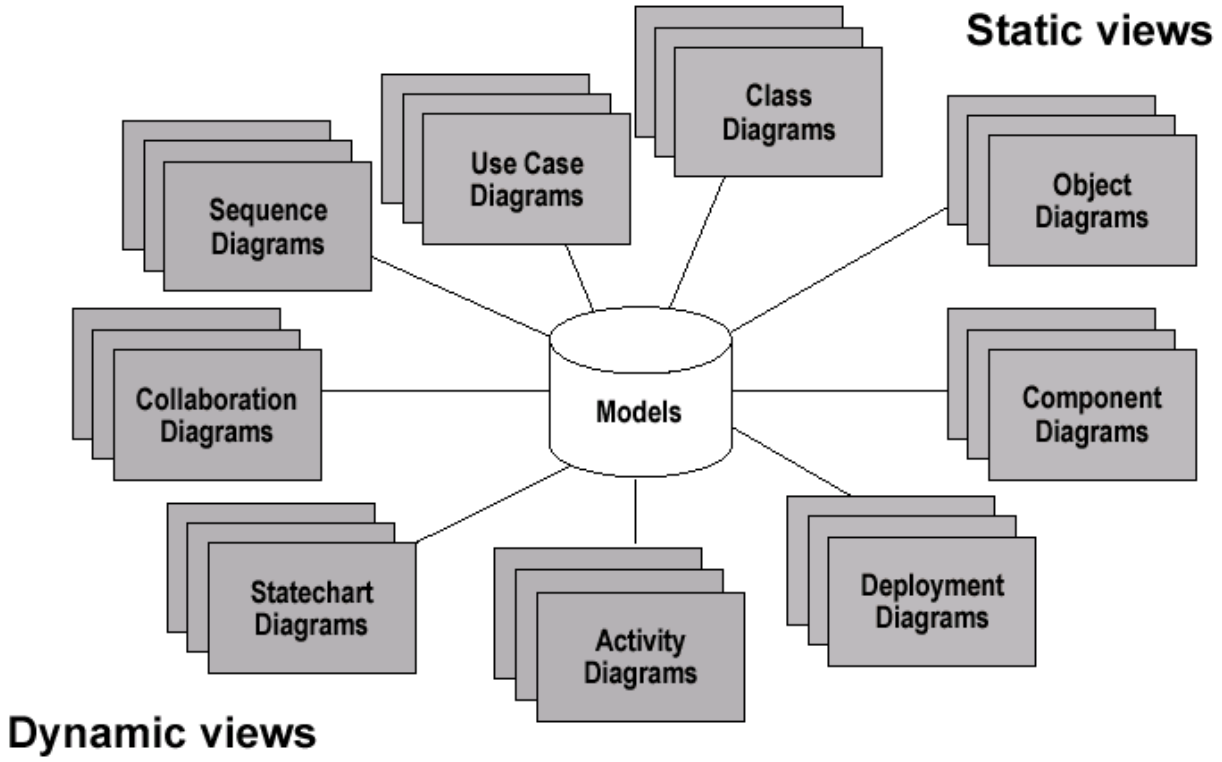
DIAGRAMS

- Each view requires a number of diagrams that contain information emphasizing a particular aspect of the system. A slight overlap does exist, so a diagram can actually be a part of more than one view.
- By looking at the system from different views, it is possible to concentrate on one aspect of the system at a time.
- A diagram in a particular view needs to be simple enough to communicate information clearly, yet coherent with the other diagrams and views so that the complete picture of the system is described by all the views put together.
- A system model typically has several diagrams of varying types, depending on the goal for the model.
- A diagram is part of a specific view, and when it is drawn, it is usually allocated to a view.
- Some diagram types can be part of several views, depending on the contents of the diagram.

UML MODELS, VIEWS, DIAGRAMS

- UML is a multi-diagrammatic language
  - Each diagram is a view into a model
    - Diagram presented from the aspect of a particular stakeholder
    - Provides a partial representation of the system
    - Is semantically consistent with other views

MODELS, VIEWS, DIAGRAMS



USE CASE VIEW

- The use-case view describes the functionality the system should deliver, as perceived by external actors.
- An actor interacts with the system; the actor can be a user or another system.
- The use-case view is used by customers, designers, developers, and testers; it is described in use-case diagrams, sometimes with support from activity diagrams.
- The desired usage of the system is described as a number of use cases in the use-case view, where a use case is a generic description of a function requested.

CENTRAL VIEW

- The use-case view is central, because its contents drive the development of the other views.
- The final goal of the system is to provide the functionality described in this view—along with some nonfunctional properties.
- Hence, this view affects all the others.
- This view is also used to validate the system and finally to verify the functioning of the system by testing the use-case view with the customers (asking, “Is this what you want?”) and against the finished system (asking, “Does the system work as specified?”).

LOGICAL VIEW

- The logical view describes how the system's functionality is provided.
- It is mainly for designers and developers. In contrast to the use-case view, the logical view looks inside the system. It describes both the static structure (classes, objects, and relationships) and the dynamic collaborations that occur when the objects The static structure is described in class and object diagrams.
- The dynamic modeling is described in state machines, and interaction and activity diagrams.

IMPLEMENTATION VIEW

- The implementation view describes the main modules and their dependencies.
- It is mainly for developers and consists of the main software artifacts. The artifacts include different types of code modules shown with their structure and dependencies.
- Additional information about the components, such as resource allocation (responsibility for a component) or other administrative information, such as a progress report for the development work, can also be added.
- The implementation view will likely require the use of extensions for a specific execution environment.

PROCESS VIEW

- The process view deals with the division of the system into processes and processors. This aspect allows for efficient resource usage, parallel execution, and the handling of asynchronous events from the environment.
- This view must also deal with the communication and synchronization of these threads.
- This view provides critical information for developers and integrators of the system.
- The view consists of dynamic diagrams (state machines, and interaction and activity diagrams) and implementation diagrams (interaction and deployment diagrams). A timing diagram also provides a specialized tool for the process view.
- A timing diagram provides a way to show the current status of an object in terms of time.

DEPLOYMENT VIEW

- Finally, the deployment view shows the physical deployment of the system, such as the computers and devices (nodes) and how they connect to each other.
- The various execution environments within the processors can be specified as well.
- The deployment view is used by developers, integrators, and testers and is represented by the deployment diagram.
- This view also includes a mapping that shows how the artifacts are deployed in the physical architecture, for example, which programs or objects execute on each respective computer.

WHY IS IT CALLED THE 4 + 1 INSTEAD OF JUST 5?

- The use case view has a special significance.
- When all other views are finished, it's effectively redundant.
- However, all other views would not be possible without it.
- It details the high levels requirements of the system.
- The other views detail how those requirements are realized.

4+1 VIEW MODEL CAME BEFORE UML

- It's important to remember the 4 + 1 approach was put forward two years before the first introduction of UML.
- UML is how most enterprise architectures are modeled and the 4 + 1 approach still plays a relevance to UML today.
- UML has 13 different types of diagrams – each diagram type can be categorized into one of the 4 + 1 views.
- UML is 4 + 1 friendly!

IS IT IMPORTANT?

- It makes modeling easier.
- Better organization with better separation of concern.
- The 4 + 1 approach provides a way for architects to be able to prioritize modeling concerns.
- The 4 + 1 approach makes it possible for stakeholders to get the parts of the model that are relevant to them.

- Use Case View
  - Use Case Analysis is a technique to capture business process from user's perspective.
  - Static aspects in use case diagrams; Dynamic aspects in interaction (state-chart and activity) diagrams.
- Design View
  - Encompasses classes, interfaces, and collaborations that define the vocabulary of a system., Supports functional requirements of the system.
  - Static aspects in class and object diagrams; Dynamic aspects in interaction diagrams.
- Process View
  - Encompasses the threads and processes defining concurrency and synchronization, Addresses performance, scalability, and throughput.
  - Static and dynamic aspects captured as in design view; emphasis on active classes.
- Implementation View
  - Encompasses components and files used to assemble and release a physical system.
  - Addresses in component diagrams; Dynamic aspects in interaction diagrams.
- Deployment View
  - Encompasses the nodes that form the system hardware topology, Addresses distribution, delivery, and installation.
  - Static aspects in deployment diagrams; Dynamic aspects in interaction diagrams..

HOMOGENIZATION OF THE SYSTEM

- The word homogenize means to blend into a smooth mixture, to make homogeneous.
- In parallel design process, several stimuli with the same purpose or meaning are defined by several designers. These stimuli should be consolidated to obtain as few stimuli as possible. It is called homogenization.
- Homogenize - to change (something) so that its parts are the same or similar.

WHY HOMOGENIZE?

- As more use cases and scenarios are developed it is necessary to make the model homogeneous. This is especially true if multiple teams are working on different parts of the model.
- Since use cases and scenarios deal with the written word, people may use different words to mean the same thing or they may interpret words differently. This is just the natural maturation of the model during the project life cycle.
- Homogenization does not happen at one point in the life cycle—it must be on-going.
- Projects that wait until the end to synch up information developed by multiple groups of people are doomed to failure.
- The most successful projects are made up of teams that have constant communication mechanisms employed. Communication may be as simple as a phone call or as formal as a scheduled meeting—it all depends on the project and the nature of the need to talk.
- The only thing that matters is the fact that the teams are not working in isolation.

ELEMENTS OF HOMOGENIZATION

- Combining Classes
- Splitting Classes
- Eliminating Classes
- Consistency Checking
- Scenario Walk-Through
- Event Tracing
- Documentation Review

COMBINING CLASSES

- If different teams are working on different scenarios, a class may be called by different names. The name conflicts must be resolved.
- This is accomplished mainly through model walkthroughs. i.e.
  - Examine each class along with its definition.
  - Also examine the attributes and operations defined for the classes, and look for the use of synonyms.
  - Once you determine that two classes are doing the same thing, choose the class with the name that is closest to the language used by the customers.

COMBINING CLASSES

- Pay careful attention to the control classes created for the system.
- Initially, one control class is allocated per use case. This might be overkill—control classes with similar behavior may be combined.
- Examine the sequencing logic in the control classes for the system. If it is very similar, the control classes may be combined into one control class.
- E. g In the Course Registration System there is a control class for the Maintain Course Information use case and one for the Create Course Catalog use case. Each control class gets information from a boundary class and deals with course information. It is possible that these two control classes could be combined since they have similar behavior and access similar information.

SPLITTING CLASSES

- Classes should be examined to determine if they are following the golden rule of OO, which states that a class should do one thing and do it really well.
- They should be cohesive;
- Example :
- A StudentInformation class contains:
  - Info about student Actor
  - And info about the courses that student has successfully completed.
- This is better modeled as two classes—StudentInformation and Transcript, with an association between them.

SPLITTING CLASSES

- Often, what appears to be only an attribute ends up having structure and behavior unto itself and should be spilt off into its own class.
- For example, we'll look at Departments in the university. Each Course is sponsored by a Department. Initially, this information was modeled as an attribute of the Course class.
- Further analysis showed that it was necessary to capture the number of students taking classes in each department, the number of professors that teach department courses, and the number of courses offered by each department.
- Hence, a Department class was created. The initial attribute of Department for a Course was replaced with an association between Course and Department.

ELIMINATING CLASSES

- A class may be eliminated altogether from the model.
- This happens when:
  - The class does not have any structure or behavior
  - The class does not participate in any use cases
- Guideline , examine control classes.
- Lack of sequencing responsibility may lead to the deletion of the control class. This is especially true if the control class is only a passthrough— that is, the control class receives information from a boundary class and immediately passes it to an entity class without the need for sequencing logic.

ELIMINATING CLASSES

- In the Course Registration System, initially we would create a control class for the Select Courses to Teach use case.
- This use case provides the capability for professors to state what course offerings they will teach in a given semester.
- There is no sequencing logic needed for the control class—the professor enters the information on the GUI screen and the Professor is added to the selected offering.
- Here is a case where the control class for the use case could be eliminated.

CONSISTENCY CHECKING

- Consistency checking is needed since the static view of the system, as shown in class diagrams, and the dynamic view of the system, as shown in use case diagrams and interaction diagrams, are under development in parallel.
- Because both views are under development concurrently they must be cross-checked to ensure that different assumptions or decisions are not being made in different views.
- Consistency checking does not occur during a separate phase or a single step of the analysis process. It should be integrated throughout the life cycle of the system under development.
- Consistency checking is best accomplished by forming a small team (five to six people at most) to do the work.
- The team should be composed of a cross-section of personnel—analysts and designers, customers or customer representatives, domain experts, and test personnel

SCENARIO WALK-THROUGH

- A primary method of consistency checking is to walk through the high-risk scenarios as represented by a sequence or collaboration diagram.
- Since each message represents behavior of the receiving class, verify that each message is captured as an operation on the class diagram.
- Verify that two interacting objects have a pathway for communication via either an association or an aggregation.
- Especially check for reflexive relationships that may be needed since these relationships are easy to miss during analysis. Reflexive relationships are needed when multiple objects of the same class interact during a scenario.
- For each class represented on the class diagram, make sure the class participates in at least one scenario. For each operation listed for a class, verify that either the operation is used in at least one scenario or it is needed for completeness.
- Finally, verify that each object included in a sequence or collaboration diagram belongs to a class on the class diagram.

EVENT TRACING

- For every message shown in a sequence or collaboration diagram,
  - verify that an operation on the sending class is responsible for sending the event and an operation on the receiving class expects the event and handles it.
  - Verify that there is an association or aggregation on the class diagram between the sending and receiving classes.
  - Add the relationship to the class diagram if it is missing.
  - Finally, if a state transition diagram for the class exists, verify that the event is represented on the diagram for the receiving class.
  - This is needed because the diagram shows all the events that a class may receive.

DOCUMENTATION REVIEW

- Each class should be documented!
- Check for uniqueness of class names and review all definitions for completeness.
- Ensure that all attributes and operations have a complete definition.
- Finally, check that all standards, format specifications, and content rules established for the project have been followed.

That is all