MODEL

↳ how system will be developed
  ↳ emphasis essential details
  ↳ ommit irrelavent
↳ we need it
  ↳ comm. b/w all parties
  ↳ visualization of all facts
  ↳ verification of facts

System: solving some problem
  ↳ functional aspects → state structure, dynamic interactions
  ↳ non functional aspects → timing req
  ↳ organizational aspects → work organization
↳ has multiple diagrams

DIAGRAM

↳ represent some aspect of system
↳ can be part of more than 1 view

VIEW

→ not all system req views

↳ representation of system model
↳ diff views for one system
  e.g data view, security view,
  deployment view → single processor
  drop process view → single process
  drop implementational view → very small program

UML VIEWS

↳ structural classification
↳ Dynamic behavior
↳ physical layout
↳ model management

UML VIEWS AND DIAGRAMS

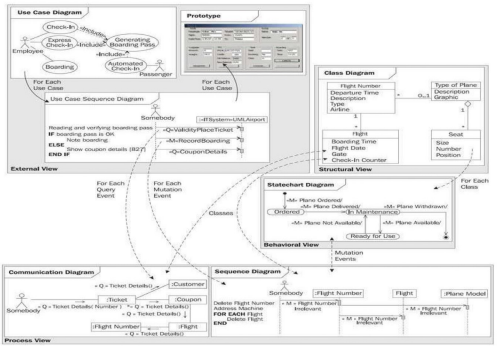
Major View	Diagram	Concepts
structural	class diagram	association, class, dependency, generalization, interface, realization
	internal structure collaboration diagram component diagram	connector, interface, part, port, provided interface, role, required interface
		connector, collaboration, collaboration 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

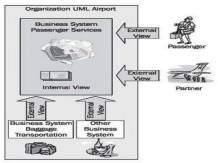


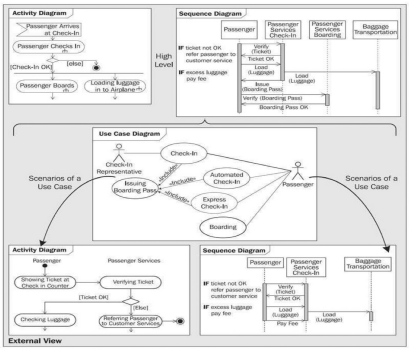
ONE MODEL → TWO VIEWS

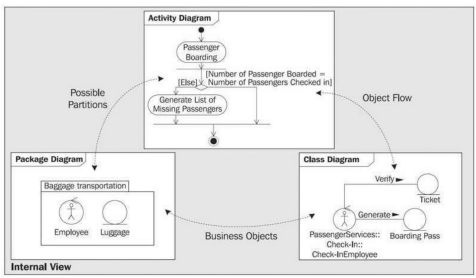
↳ External view

↳ Internal view → seavence diagram

EXTERNAL AND INTERNAL VIEW





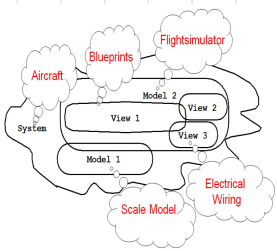


* views a models of a single system may overlap eachomer

e.g system: Aircraft

Model: 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.

4 + 1 View Model → Came before UML

↳ no view possible w/o it

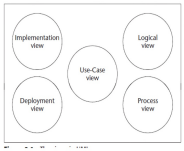
↳ details high level requirements → other views detail how those requirements are realized

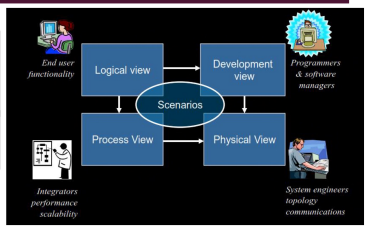
↳ is UML friendly → each UML diagram can be categorised in 4+1 view

PROS

- 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.

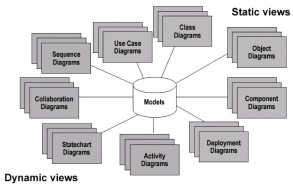
THE VIEWS IN UML





SDA

MODELS, VIEWS, DIAGRAMS



SDA

USECase view

customers designers developers testers

- describes functionality
- generic desc of function
- is central
- looks outside system

central view

- this view affects all others
- final goal is to provide functionality described in this view
- used to validate system

logical view

designers developers

- describes how functionality is provided
- looks inside system
- describes static structure, dynamic collaborations
  - class diagrams, object diagram
  - state machine interaction activity diagrams

implementation view

developers

describes
- main modules and their dependencies

consist of
- main software artifacts
- additional info about components
  - resource allocation
  - administrative info

process view

developers integrators

deals with
- division of system into processes and processors
  - efficient resource usage
  - parallel execution
  - asynchronous handling

deals with
- comm. and synchronization of threads

consists of
- dynamic diagrams, implementational diagrams (interaction, deployment diagram)
- timing diagram

deployment view

developers integrators testers

- shows physical deployment of system (computer, devices how they connect to e/o)
- mapping of artifacts (which program/object execute on each respective computer)

- 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 system

smooth into a mixture

↳ to change something so that its part are the same/similar

↳ is on-going in life cycle → not a one time thing

WHY

↳ if multiple teams are working

↳ people may use diff words/interpret words diff

- 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

1 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.

- 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.

2 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.

- 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.

3 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.
- 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.

4 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

5 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.

6 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.

7 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.

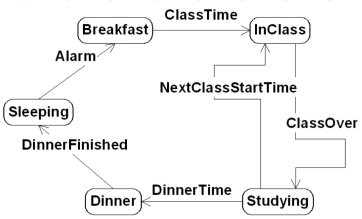
week 11 b

State chart/Transition/Diagram

→ used for real time modeling e.g traffic

- ↳ multiple usecases
- ↳ 1 object
- ↳ state to state

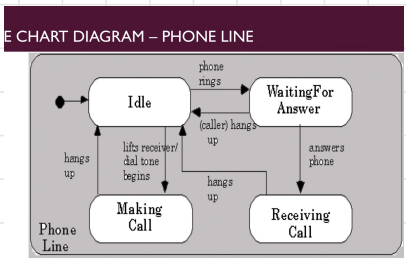
e.g borrower → talking state / listening state



Activity Diagram

- ↳ 1 usecase
- ↳ multiple objects
- ↳ activity to activity

e.g borrower librarian



States

- ↳ obj moves from 1 state to another when trigger encountered
- ↳ start, end, possible states
- ↳ state types
  - ↳ activity
  - ↳ time
  - ↳ function call
  - ↳ condition → trigger
- ↳ disjoint states → not concurrent
- ↳ sequential states
- ↳ concurrent states
- ↳ substate → aka nested state
- ↳ orthogonal state

* trigger less transition possible
* no end state possible

State of Object

- State may also label activities, e.g., do/check item
- Examples of object states are:
  - The invoice (object) is paid (state).
  - The car (object) is standing still (state).
  - The engine (object) is running (state).
  - Jim (object) is playing the role of a salesman (state).

Transitions

Triggers / event

- ↳ a condition becoming true
- ↳ receipt of an explicit signal
- ↳ receipt of a call
- ↳ passage for a specific period of time

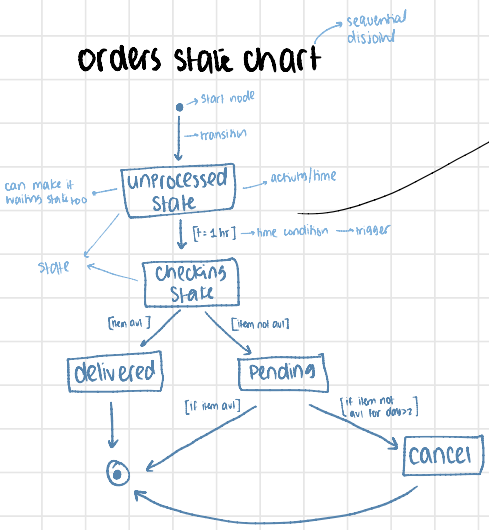
[t = 15 sec] → guard condition

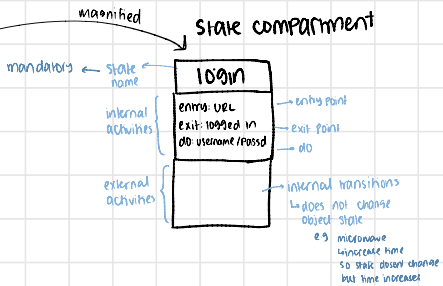
withdraw(amount) [balance > amount]

event

EXAMPLE- ORDER

- Here is just an example of how an online ordering system might look like :
- On the event of an order being received, we transit from our initial state to Unprocessed order state.
- The unprocessed order is then checked.
- If the order is rejected, we transit to the Rejected Order state.
- If the order is accepted and we have the items available we transit to the fulfilled order state.
- However if the items are not available we transit to the Pending Order state.
- After the order is fulfilled, we transit to the final state.





- The entry event can be used to specify actions on the entry of a state; for example, assigning an attribute or sending a message.
- The exit event can be used to specify actions on exit from a state.
- The do event can be used to specify an action performed while in the state; for example, sending a message, waiting, or calculating.

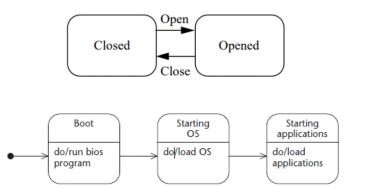


Figure 5.6 State transitions without explicit events. The transitions occur when the activities in each state have been performed.

internal transitions

↳ some actions occur, but state doesn't change

- For example, when one types on a keyboard, it responds by generating different character codes. However, unless the Caps Lock key is pressed, the state of the keyboard does not change (no state transition occurs).

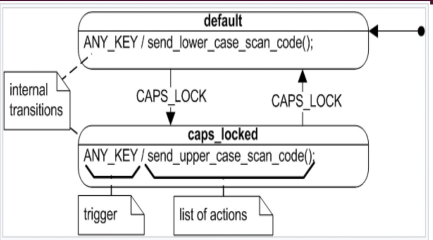


Figure 6: UML state diagram of the keyboard state machine with internal transitions

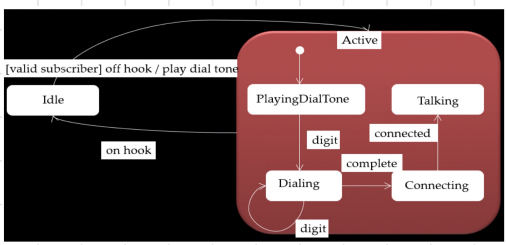
composite state

→ is active

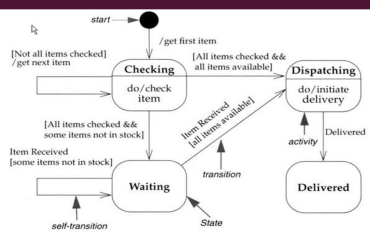
→ nested states

↳ state having substates

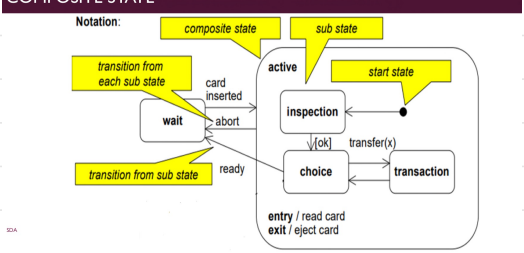
↳ substates can be seaquential/concurrent



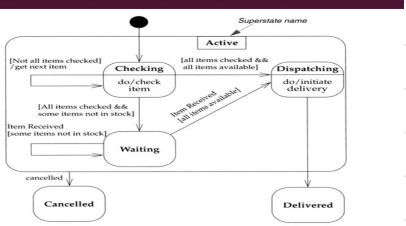
ORDER MANAGEMENT STATE CHART

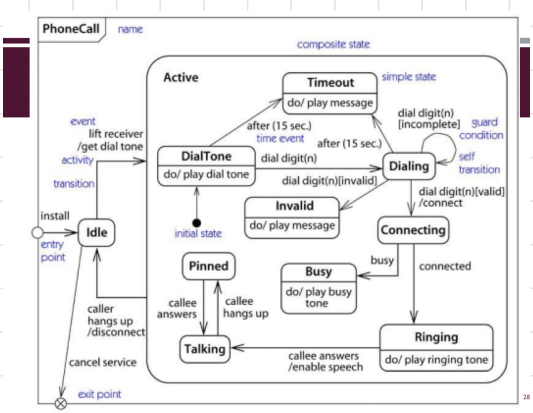


COMPOSITE STATE



ORDER MANAGEMENT STATE CHART





concurrency → orthogonal, fork, join

↳ expressed by orthogonal state → a composite state with multiple regions

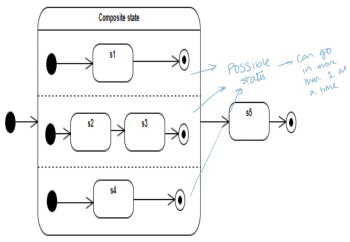


Figure 8. Orthogonal state

→ shown by fork, join

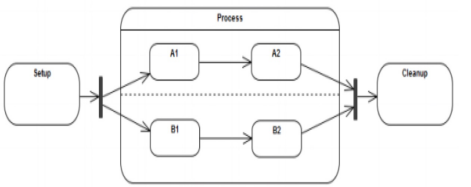
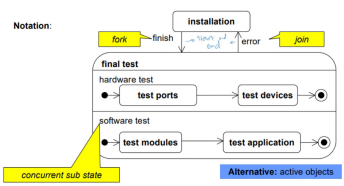


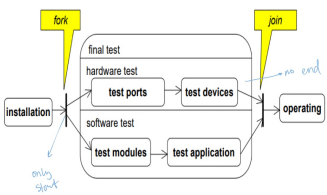
Figure 9. Fork and Join Pseudostates

CONCURRENT SUB STATES

- In a state several sequences of sub states described by own state machines can be performed concurrently.

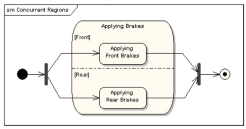


CONCURRENT SUB STATES:ALTERNATIVE

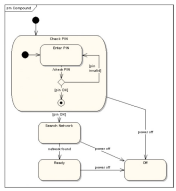


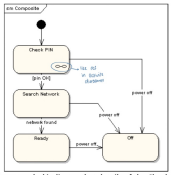
State Machine Diagrams-Concurrent

- ***Concurrent Regions*** - A state may be divided into regions containing sub-states that exist and execu concurrently. The example below shows that within the state "Applying Brakes", the front and re brakes will be operating simultaneously and independently.



- ***Compound States*** - A state machine diagram may include sub-machine diagrams, as in the example below.

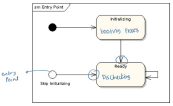


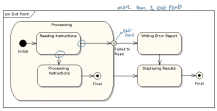


Alternative way to show the same information

•The ∞ symbol indicates that details of the Check PIN sub-machine are shown in a separate diagram.

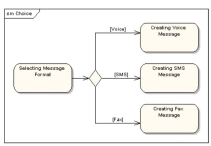
- ***Entry Point*** - Sometimes you won't want to enter a sub-machine at the normal initial state. For example, in the following sub-machine it would be normal to begin in the "Initializing" state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the "Ready" state by transitioning to the named entry point.



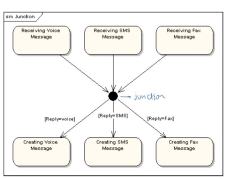


- ***Exit Point*** - In a similar manner to entry points, it is possible to have named alternative exit points. The following diagram gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.

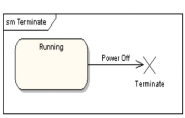
- ***Choice Pseudo-State*** - A choice pseudo-state is shown as a diamond with one transition arriving and two or more transitions leaving. The following diagram shows that whichever state is arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.



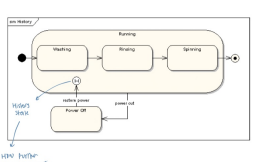
- ***Junction Pseudo-State*** - Junction pseudo-states are used to chain together multiple transitions. A single junction can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition. Junctions are semantic-free.

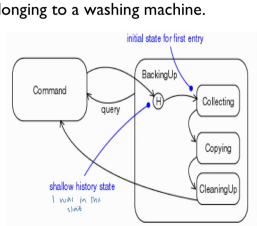


- ***Terminate Pseudo-State*** - Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A terminate pseudo-state is notated as a cross.

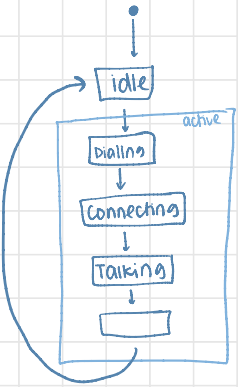


- ***History States*** - A history state is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.

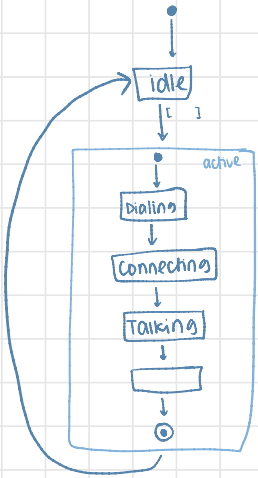




Substates eg



nested eg



compound
composite
superstate
→ a state having substates

EXAMPLE:

- ATM is initially turned off. After the power is turned on, ATM performs startup action and enters **Self Test** state. If the test fails, ATM goes into **Out of Service** state, otherwise there is **triggerless transition** to the **Idle** state. In this state ATM waits for customer interaction.
- The ATM state changes from **Idle** to **Serving Customer** when the customer inserts banking or credit card in the ATM's card reader. On entering the **Serving Customer** state, the entry action **readCard** is performed. Note, that transition from **Serving Customer** state back to the **Idle** state could be triggered by **cancel** event as the customer could cancel transaction at any time.
- **Serving Customer** state is a composite state with sequential substates **Customer Authentication**, **Selecting Transaction** and **Transaction**.**Customer Authentication** and **Transaction** are composite states by themselves which is shown with hidden decomposition indicator icon. **Serving Customer** state has **triggerless transition** back to the **Idle** state after transaction is finished. The state also has exit action **ejectCard** which releases customer's card on leaving the state, no matter what caused the transition out of the state.

Component

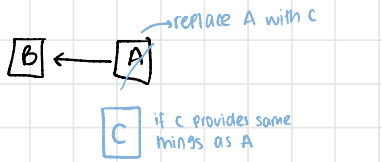
↳ communicate with e/o using interface

↳ idependent unit providing/requesting services from system

↳ internal, external view

↳ models physical implementation of model

1. Components → can be replaced by another component if provides interface similar
2. Provided interface
3. Required interface
4. Ports
5. Ports relations



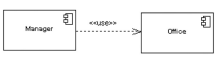
↳ can be offmeself OR API → eg paid software

COMMON STEREOTYPES → components

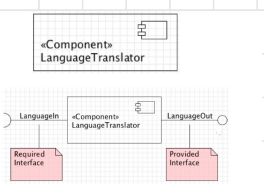
Stereotype	Indicates
<<application>>	A "front-end" of your system, such as the collection of HTML pages and ASP/JSPs that work with them for a browser-based system or the collection of screens and controller classes for a GUI-based system.
<<database>>	A hierarchical, relational, object-relational, network, or object-oriented database.
<<document>>	A document. A UML standard stereotype.
<<executable>>	A software component that can be executed on a node. A UML standard stereotype.
<<file>>	A data file. A UML standard stereotype.
<<infrastructure>>	A technical component within your system such as a persistence service or an audit logger.
<<library>>	An object or function library. A UML standard stereotype.
<<source code>>	A source code file, such as a .java file or a .cpp file.
<<table>>	A data table within a database. A UML standard stereotype
<<web service>>	One or more web services.
<<XML DTD>>	An XML DTD.

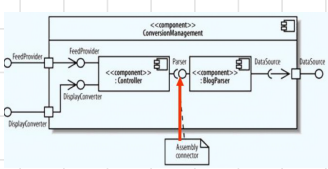
DEPENDENCIES

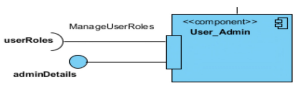
- Components can be connected by usage dependencies.

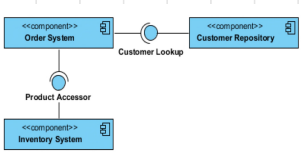


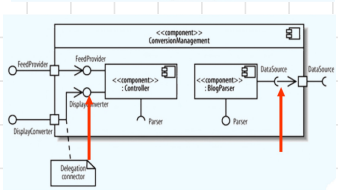
- **Usage Dependency**
- A usage dependency is relationship which one element requires another element for its full implementation
- Is a dependency in which the client requires the presence of the supplier
- Is shown as dashed arrow with a <<use>> keyword
- The arrowhead point from the dependent component to the one of which it is dependent

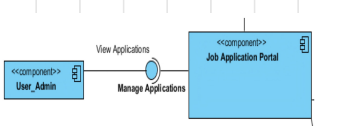


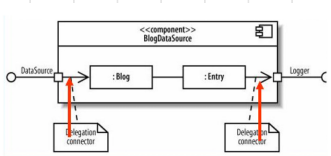




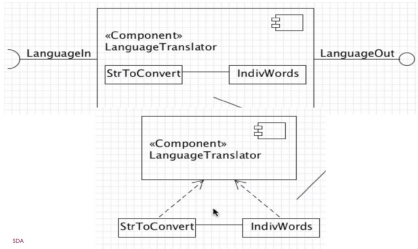




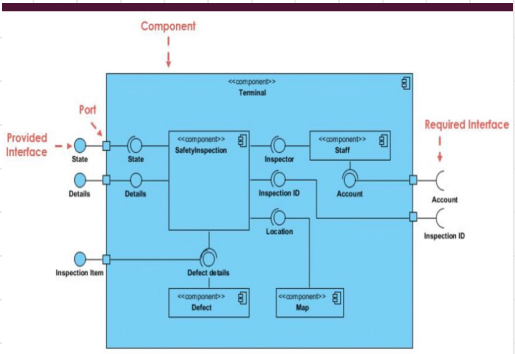




INTERNAL REALIZATION



- The example above shows the internal components of a larger component:
- The data (account and inspection ID) flows into the component via the port on the right-hand side and is converted into a format the internal components can use. The interfaces on the right are known as required interfaces, which represents the services the component needed in order to carry out its duty.
- The data then passes to and through several other components via various connections before it is output at the ports on the left. Those interfaces on the left are known as provided interface, which represents the services to deliver by the exhibiting component.
- It is important to note that the internal components are surrounded by a large 'box' which can be the overall system itself (in which case there would not be a component symbol in the top right corner) or a subsystem or component of the overall system (in this case the 'box' is a component itself).

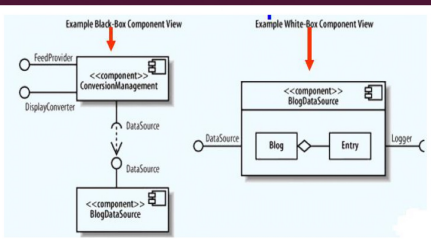


components have

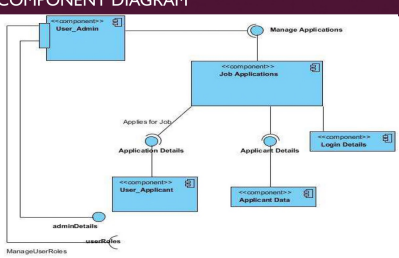
internal view → white box view

external view → black box view

BLACK-BOX AND WHITE-BOX VIEWS

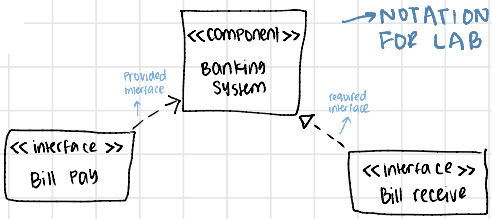


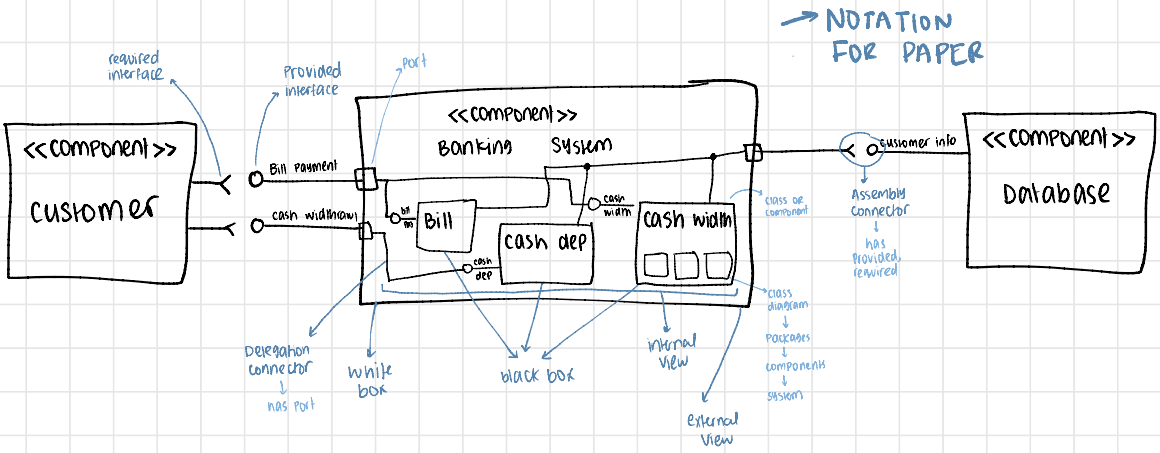
SAMPLE COMPONENT DIAGRAM



COMPONENT DIAGRAM GUIDELINES

- **Use Descriptive Names for Architectural Components**
  - Use Environment-Specific Naming Conventions for Detailed Design Components
  - Apply Textual Stereotypes to Components Consistently
- **Interfaces**
  - Prefer Lollipop Notation To Indicate Realization of Interfaces By Components
  - Prefer the Left-Hand Side of A Component for Interface Lollipops
  - Show Only Relevant Interfaces
- **Dependencies and Inheritance**
  - Model Dependencies From Left To Right
  - Place Child Components Below Parent Components
  - Components Should Only Depend on Interfaces





Deployment diagrams

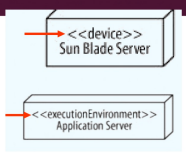
↳ physical r/s b/w hardware and software

(Computer, Processing devices)

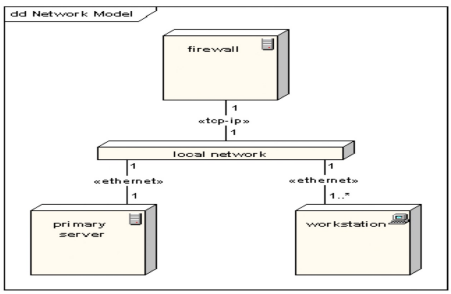
↳ has nodes, connectors, artifacts (Physical files eg files, tables in database executable files)

NODES

- Nodes can be:
- Hardware nodes:
  - Server
  - Desktop PC
  - Disk drive
- Execution nodes:
  - Operating system
  - J2EE container
  - Web server
  - Application server
- Nodes can be nested,

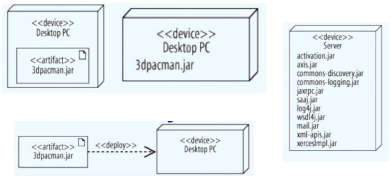


SDA

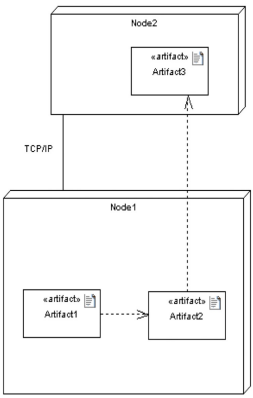


SDA

ARTIFACTS

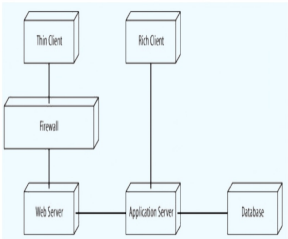


SDA



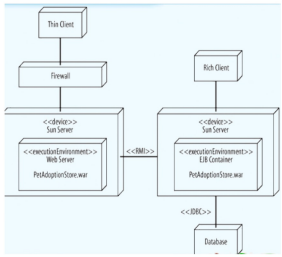
WHEN TO USE?

- At the early stage: helps figuring out the general configuration.
- Example: a web application will include:
  - A web server, application server and database
  - Clients access the application through browsers
  - The web server should have a firewall

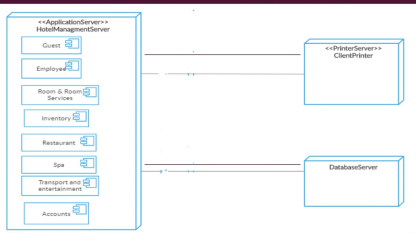


WHEN TO USE?

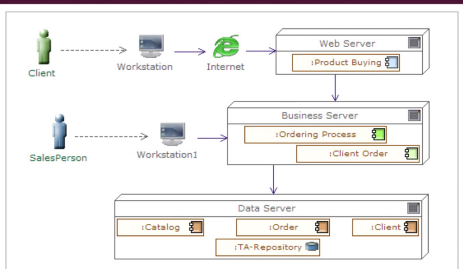
- At the later stages: the deployment diagram will come into details about the system architecture.
  - Which technology is used
  - What communication protocols are used
  - Software artifacts
  - etc.



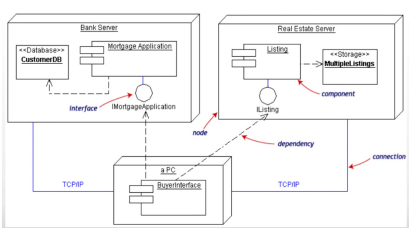
DEPLOYMENT DIAGRAMS EXAMPLE

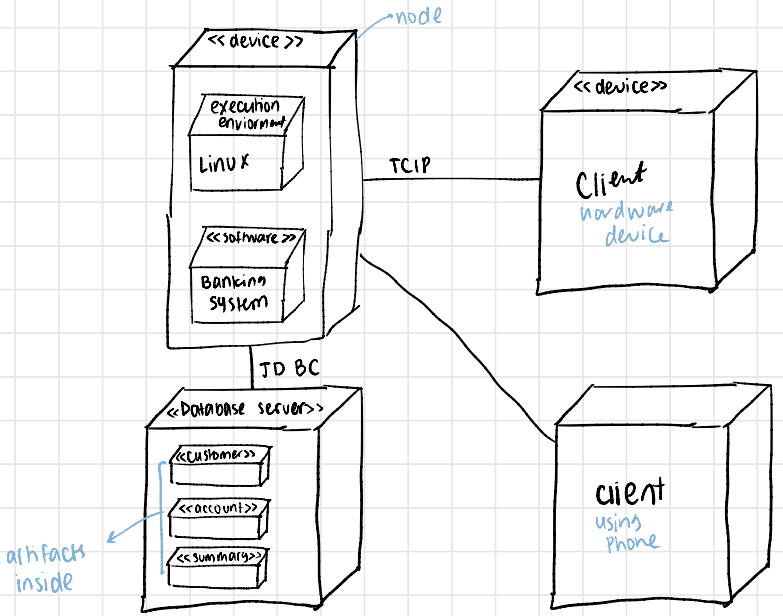


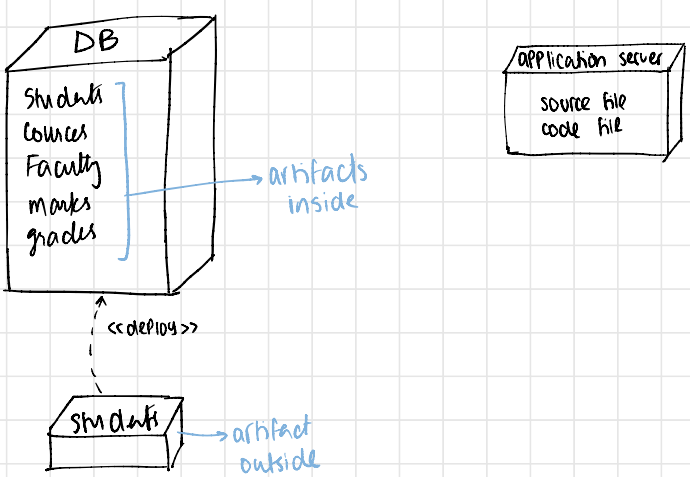
DEPLOYMENT DIAGRAMS EXAMPLE



DEPLOYMENT DIAGRAMS EXAMPLE




node
<<device>>
execution enviornment
Linux
<<software>>
Banking system
TCIP
<<device>>
Client
hardware device
JDBC
<<Database server>>
<<customer>>
<<account>>
<<summary>>
artifacts inside
client
using Phone


DB
Students
Courses
Faculty
marks
grades
artifacts inside
<<deploy>>
Students
artifact outside
application server
source file
code file

Design Pattern

made by → 'GANG OF 4'

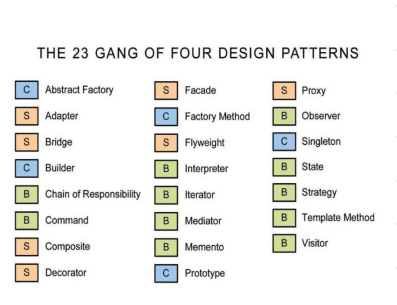
Week 13

singleton not coming

↳ Common Problem solution

↳ Template

↳ reusable solution to a common occuring pattern



↳ 23 design Patterns

↳ categorised into 3

Structural → diff things joined

↳ how to combine objects/classes to form large system

↳ r/s between entities

e.g using API
- bridge
- facade
- Adapter

Creational → create object Problems

↳ object creation

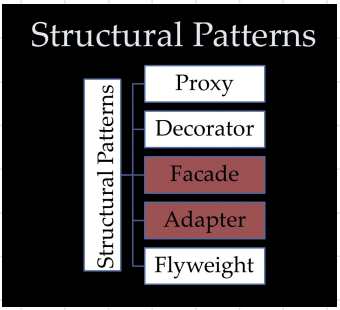
↳ gives more flexibility to decide which objects to create

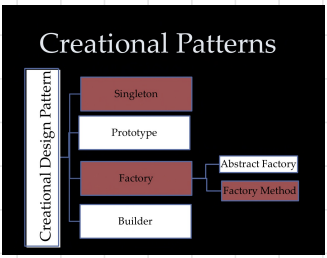
e.g creating a store
- Singleton
- factory
- abstract factory

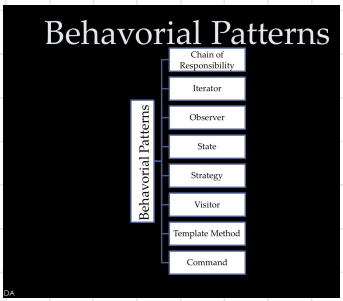
Behavioral → how a function is working

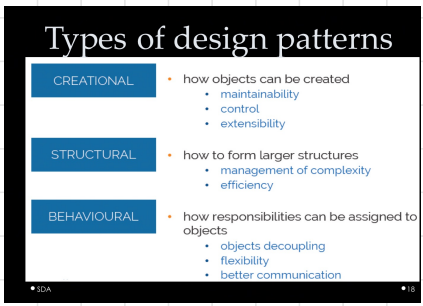
↳ comm. b/w entities

e.g chain of responsibility
- Command
- interpreter







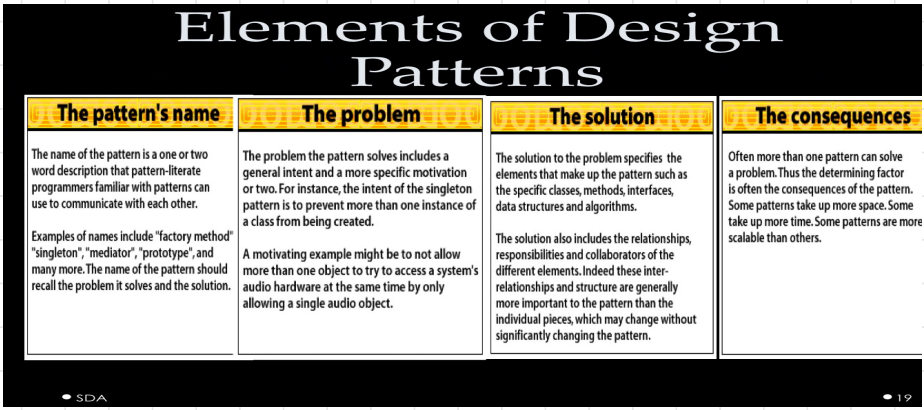


DESIN PATTEN EIEMENTS

↳ has 4 elements
- ↳ name
- ↳ problem
- ↳ solution
- ↳ conceqvences

WHY needed?

↳ provides developers with
- ↳ reusable solutions to common problems
- ↳ a shared library
- ↳ help design faster



↳ factory → creating → creational design patter
↳ adapter → join 2 things → structural design pattern
} → make class diagram

Factory pattern → creational

Factory → creational

Product — interface

directing to respected subclass

concrete Product class

circle, square } objects

concrete creator

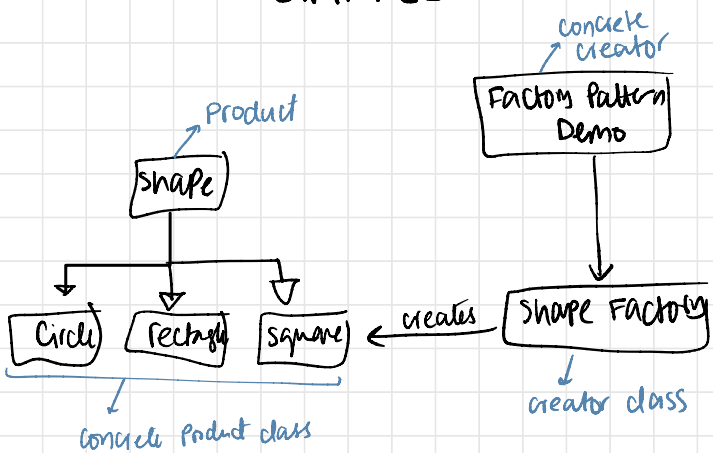
main

Passing objects

creator class

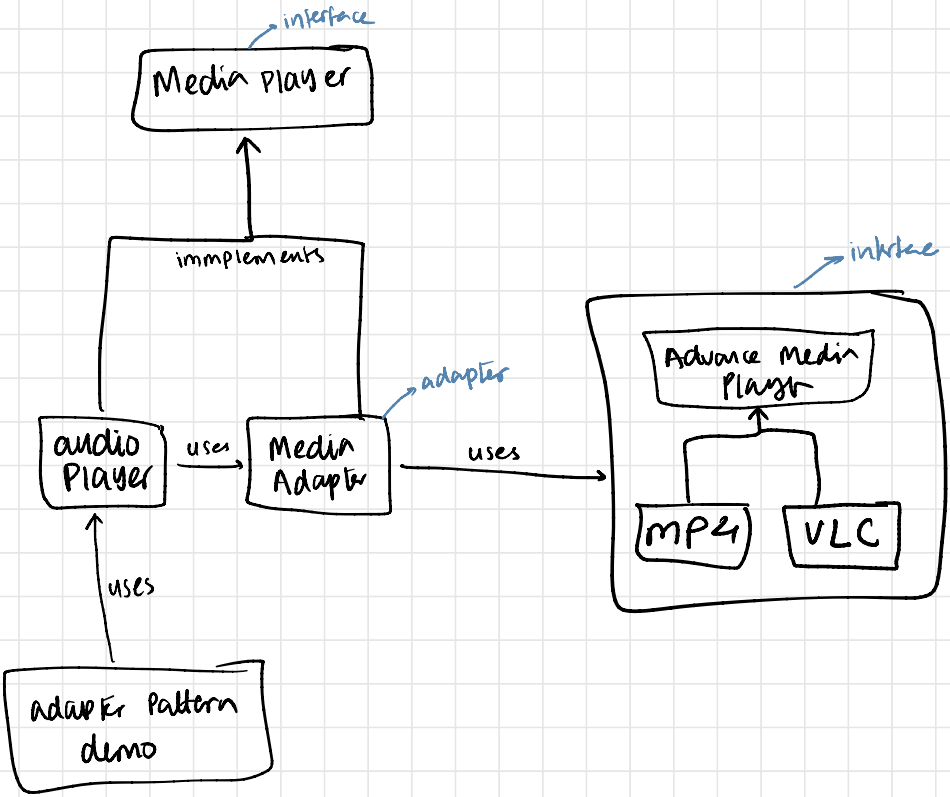
if circle Passed_draw circle

EXAMPLE



writing code will come

ADAPTER → structural



i. Static

a. Use case diagram

b. Class diagram

ii. Dynamic

a. Activity diagram

b. Sequence diagram

c. Object diagram

d. State diagram

e. Collaboration diagram

iii. Implementation

a. Component diagram

b. Deployment diagram