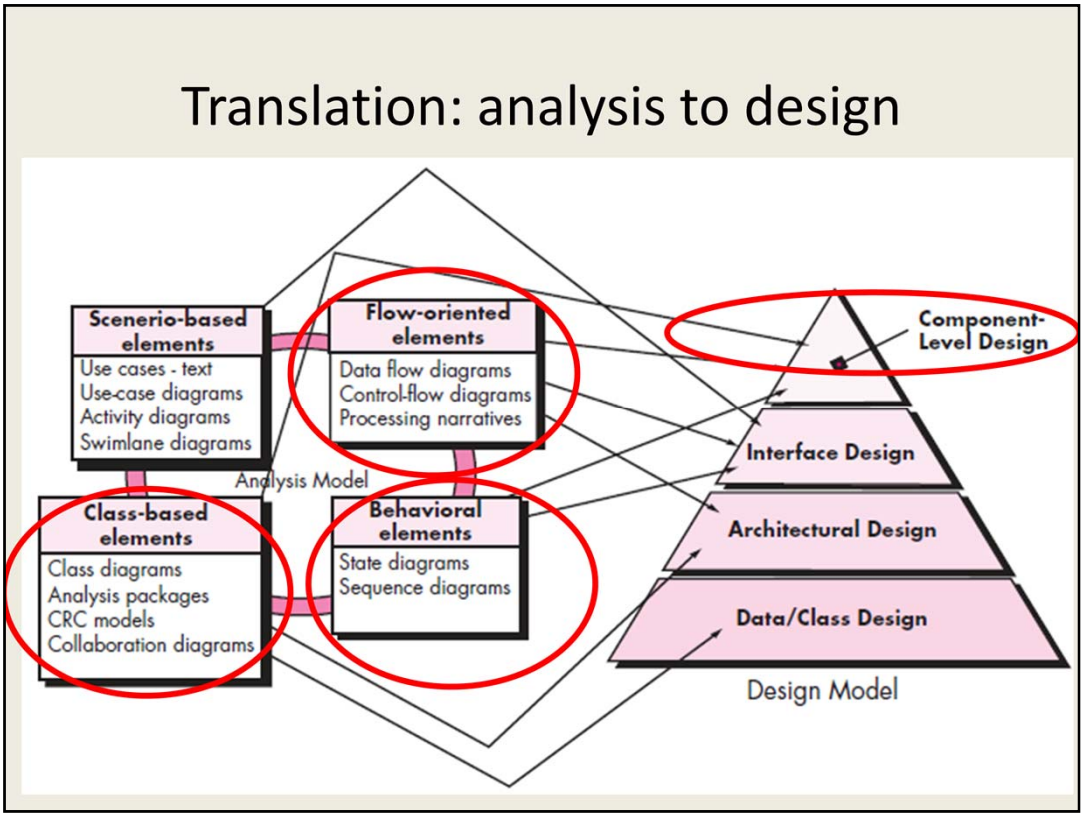
Component level design
(II)



Data/Class design: transforms analysis classes into implementation classes and data structures
Architectural design: defines relationships among the major structural elements of the software, the architectural styles and patterns
Interface design–defines how software elements, hardware elements, and end-users communicate. Usage scenarios and behavioral models are used
Component-level design–transforms structural elements into procedural descriptions of software components. Class-based models and behavioral models server as the basis

Only scenario-based elements serve just one design model

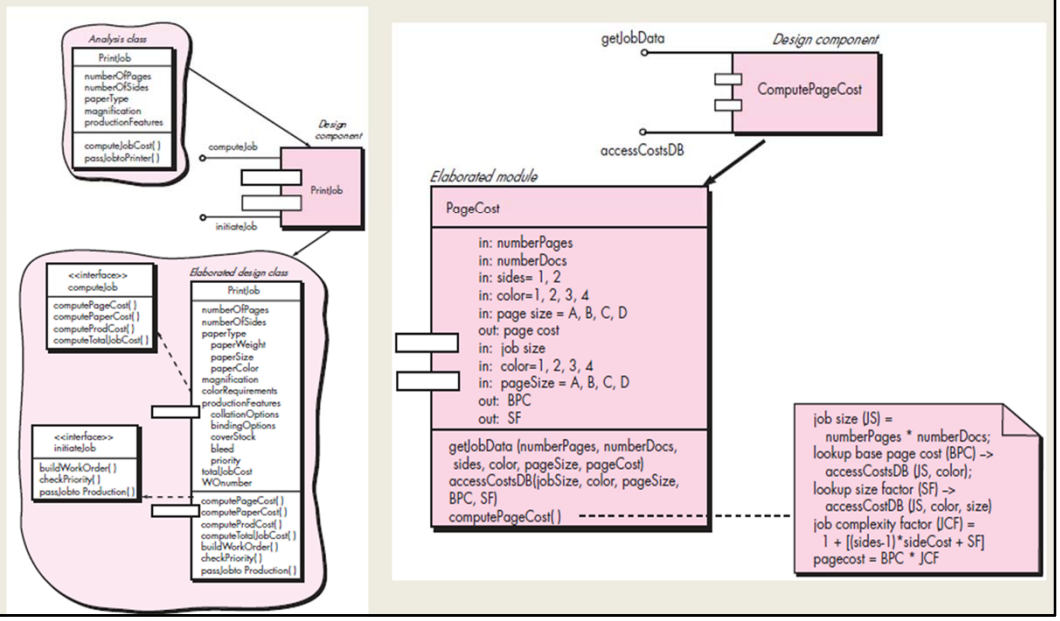
Component level design

- Major steps / typical task set

1. Identify all design classes that correspond to the problem domain.
2. Identify all design classes that correspond to the infrastructure domain.
3. Elaborate all design classes that are not acquired as reusable components.
4. Describe persistent data sources and identify classes to manage them.
5. Develop and elaborate behavioral representations for a class/component.
6. Elaborate deployment diagrams to provide additional implementation detail.
7. Factor every component-level design representation and consider alternatives.

Component level design – step 1

1. Identify all design classes that correspond to the problem domain.



Using the requirements and architectural model, each analysis class and architectural component is elaborated

We talked about this last time when looking at the concept of 'component' in different views: left – OO view, right – traditional/structural view

Component level design – step 2

2. Identify all design classes that correspond to the infrastructure domain.

- In infrastructure domain
  - GUI components
  - OS components
  - Data management components

note: components/classes are used exchangeably now as each component is derived from a class through elaboration

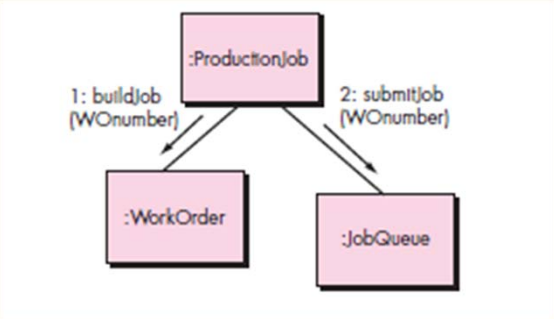
These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point.

Classes and components in this category include GUI components (often available as reusable components), operating system components, and object and data management components.

Component level design – step 3

3. Elaborate all design classes that are not acquired as reusable components.

- 3.1 Specify message details when classes or components collaborate
  - Collaboration/communication diagram



[guard condition] sequence expression (return value) : message name (argument list)

Requirement analysis

Component level design

3. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail.

Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted

3.1 The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another.

As component-level design proceeds, it is sometimes useful to show *the details of these collaborations by specifying the structure of messages* that are passed between objects within a system.

Three objects, **ProductionJob, WorkOrder,** and **JobQueue,** collaborate to prepare a print job for submission to the production stream.

Messages are passed between objects as illustrated by the arrows in the figure.

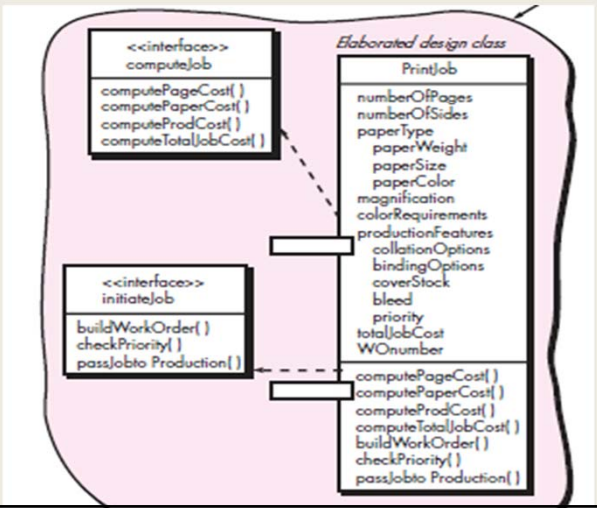
A [guard condition] is written in Object Constraint Language (OCL)5 and specifies any set of conditions that must be met before the message can be sent; Sequence expression is an integer value that indicates the sequential order in which a message is sent; (return value) is the name of the information that is returned by the operation invoked by the message; message name identifies the

operation that is to be invoked; (argument list) is the list of attributes that are passed to the operation.

Component level design – step 3

3. Elaborate all design classes that are not acquired as reusable components.

- 3.2 Identify appropriate interfaces for each component
  - Interface: externally visible (public) operations



- The interface needs to be cohesive
- What about *initiateJob* here?

Within the context of component-level design, a UML interface is “a group of externally visible (i.e., public) operations;

The interface contains no internal structure, it has no attributes, no associations; an interface is the equivalent of an abstract class that provides a controlled connection between design classes.

In essence, operations defined for the design class are categorized into one or more abstract classes.

Every operation within the abstract class (the interface) should be cohesive; that is, it should exhibit processing that focuses on one limited function or subfunction.

*initiateJob*does not exhibit sufficient cohesion.

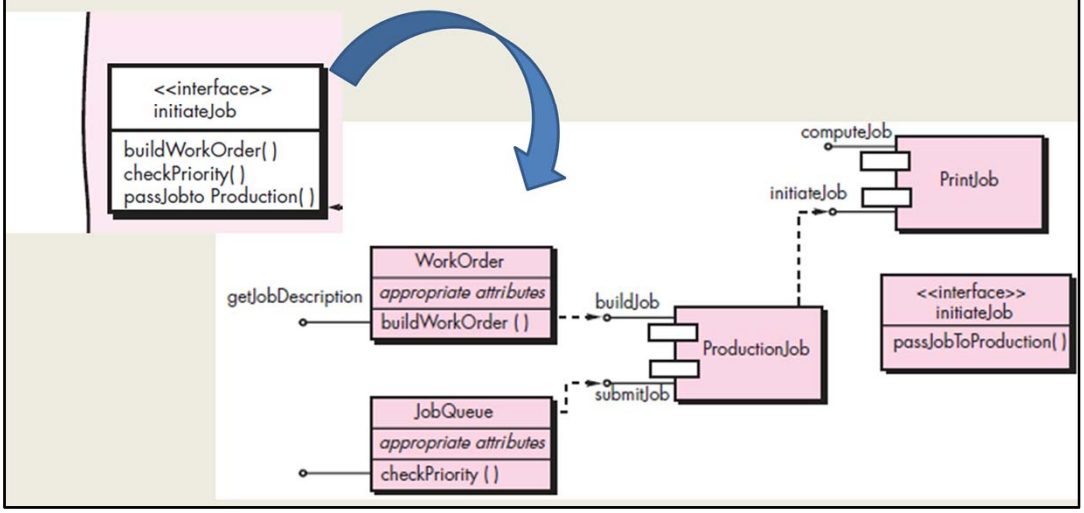
In actuality, it performs three different subfunctions—building a work order, checking job priority, and passing a job to production.

The interface design should be refactored.

Component level design – step 3

3. Elaborate all design classes that are not acquired as reusable components.

- 3.2 Identify appropriate interfaces for each component
  - Refactoring non-cohesive interface



The interface design should be refactored. One approach might be to reexamine the design

classes and define a new class **WorkOrder** that would take care of all activities

associated with the assembly of a work order. The operation *buildWorkOrder()* becomes

a part of that class. Similarly, we might define a class **JobQueue** that would

incorporate the operation *checkPriority()*. A class **ProductionJob** would encompass

all information associated with a production job to be passed to the production

facility. The interface *initiateJob* would then take the form shown in Figure 10.7.

The interface *initiateJob* is now cohesive, focusing on one function. The interfaces

associated with **ProductionJob, WorkOrder,** and **JobQueue** are similarly

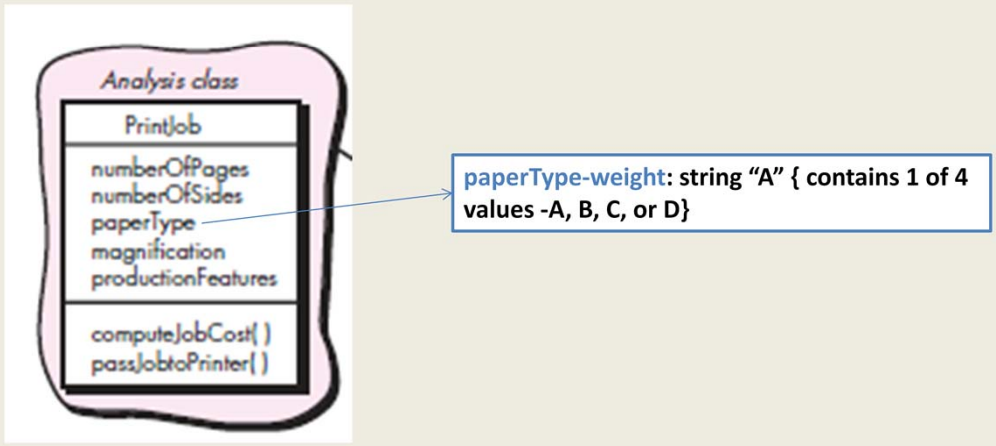
single-minded.

Component level design – step 3

3. Elaborate all design classes that are not acquired as reusable components.

- 3.3 Elaborate attributes and define data types and data structures required to implement them.

name : type-expression initial-value {property string}



In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation; UML defines an attribute's data type using the following syntax:

name is the attribute name

Type expression is the data type,

initial value is the value that the attribute takes when an object is created, and

property-string defines a property or characteristic of the attribute.

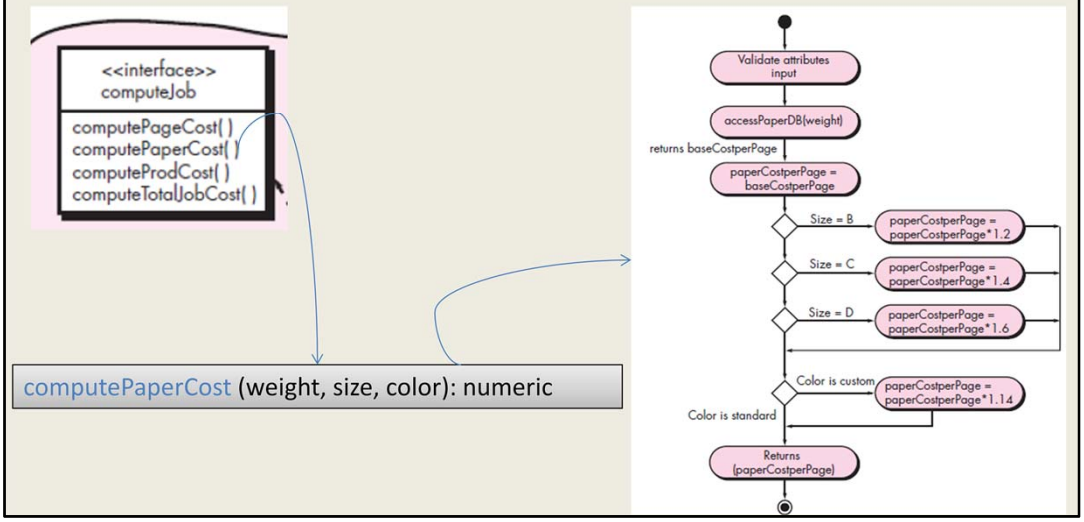
Example: defines paperType-weight as a string variable initialized to the value A that can take on one of four values from the set {A,B,C, D}.

If an attribute appears repeatedly across a number of design classes, and it has a relatively complex structure, it is best to create a separate class to accommodate the attribute.

Component level design – step 3

3. Elaborate all design classes that are not acquired as reusable components.

- 3.4 Describe processing flow within each operation in detail.



This may be accomplished using a programming language-based pseudocode or with a UML activity diagram.

Each software component is elaborated through a number of iterations that apply the *stepwise refinement* concept.

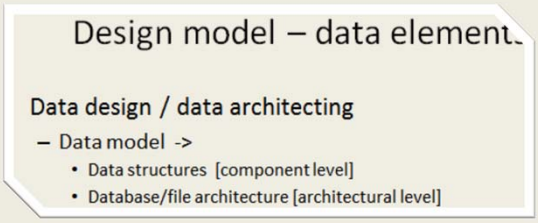
This indicates that *computePaperCost()* requires the attributes weight, size, and color as input and returns a value that is numeric (actually a dollar value) as output. If the algorithm required to implement *computePaperCost()* is simple and widely understood, no further design elaboration may be necessary---The software engineer who does the coding will provide the detail necessary to implement the operation.

However, if the algorithm is more complex or arcane, further design elaboration is required at this stage; *an alternative is using pseudo code.*

Component level design – step 4

4. Describe persistent data sources and identify classes to manage them.

- Persistent data structures
  - E.g., databases, files
  - Transcend the design of an individual component
- Initially specified as part of architecture design
  - Recall "data design"



- Now elaborate
  - Structure/organization

Databases and files normally transcend the design description of an individual component.

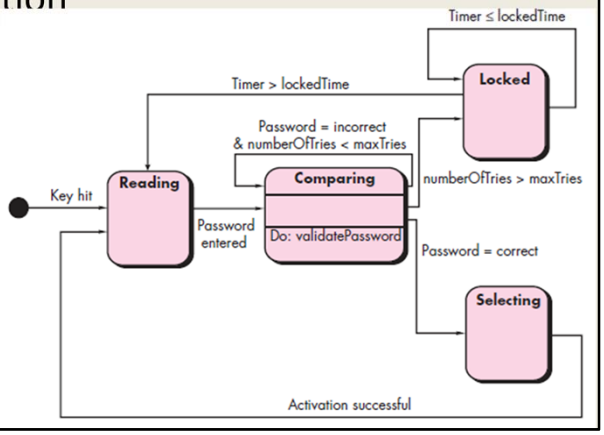
In most cases, these persistent data stores are initially specified as part of architectural design.

However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

Component level design – step 5

5. Develop and elaborate behavioral representations for a class/component.

- Examine use cases
  - Delineate objects and associated events and states
    - Recall "State diagrams"
- Behavioral representation
  - Statechart diagrams



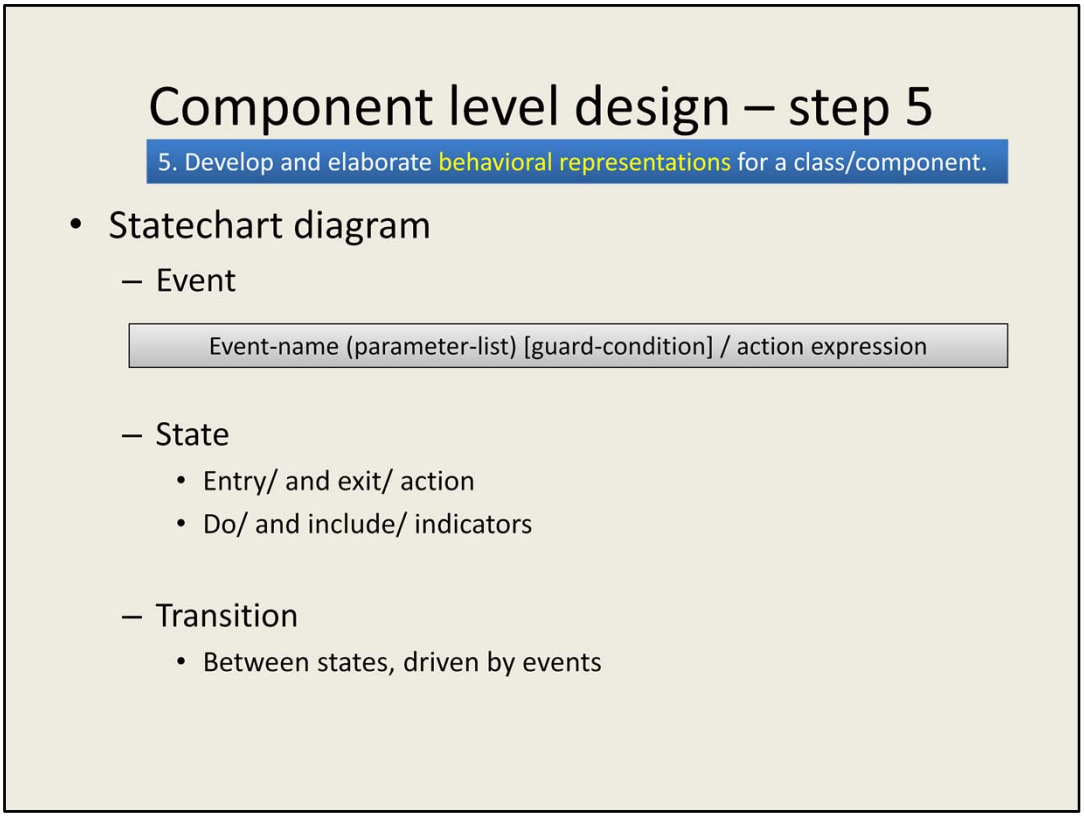
UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes.

During component-level design, it is sometimes necessary to model the behavior of a design class.

The dynamic behavior of an object (an instantiation of a design class as the program executes) is affected by events that are external to it and the current state (mode of behavior) of the object.

To understand the dynamic behavior of an object, we should examine **all use cases** that are relevant to the design class **throughout its life.**

The transitions between states are represented using a **UML statechart**



The transition from one state (represented by a rectangle with rounded corners) to another occurs as a consequence of an event that takes the form: **event-name** identifies the event, **parameter-list** incorporates data that are associated with the event, **guard-condition** is written in Object Constraint Language (OCL) and specifies a condition that must be met before the event can occur, and **action expression** defines an action that occurs as the transition takes place.

Each state may define *entry/* and *exit/* actions that occur as transition into the state occurs and as transition out of the state occurs, respectively.

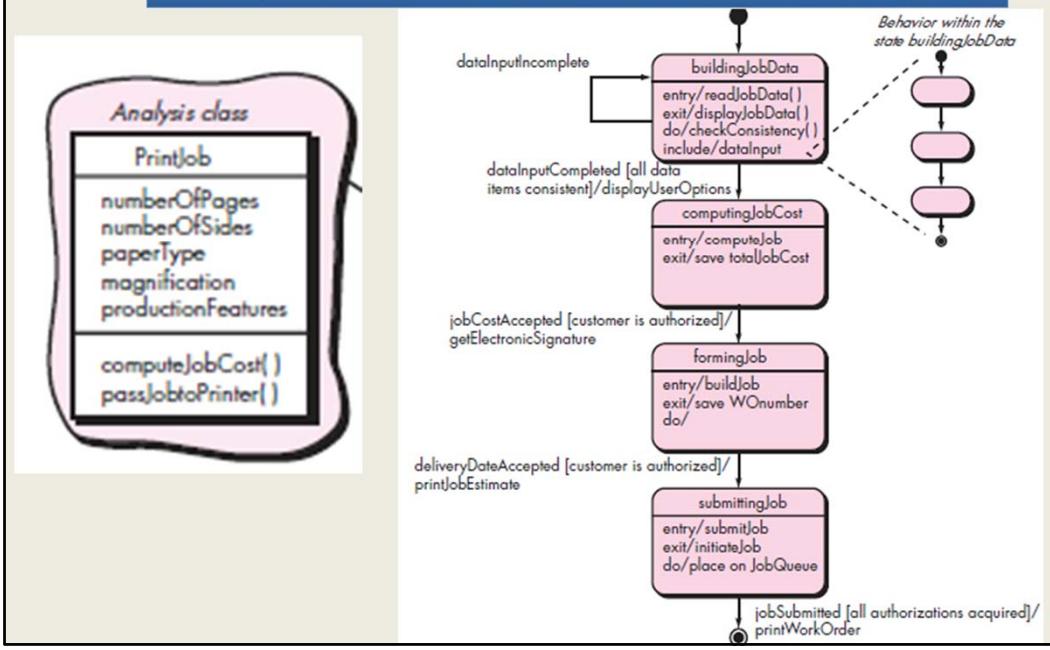
These actions correspond to operations that are relevant to the class that is being modeled.

The *do/* indicator provides a mechanism for indicating activities that occur while in the state, and

the *include/* indicator provides a means for elaborating the behavior by embedding more statechart detail within the definition of a state.

Component level design – step 5

5. Develop and elaborate behavioral representations for a class/component.



The behavioral model often contains information that is not immediately obvious in other design models.

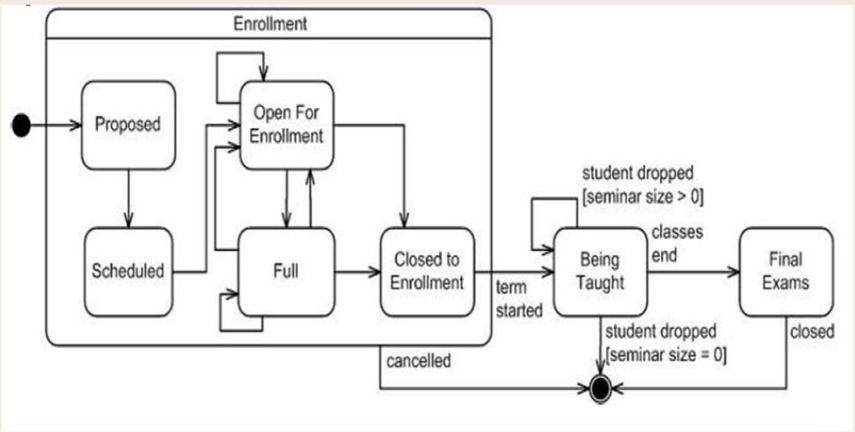
E.g.) Printshop project

The dynamic behavior of the **PrintJob** class is contingent upon two customer approvals as costs and schedule data for the print job are derived.

Without approvals (the guard condition ensures that the customer is authorized to approve) the print job cannot be submitted because there is no way to reach the ***submittingJob*** state

Statechart Diagram

- Shows the lifecycle of an analysis/design unit.
  - How events change an object over its life.

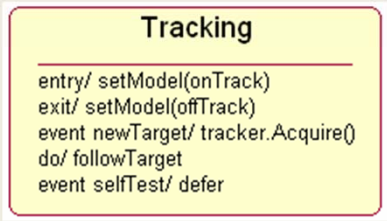


Statechart Diagram

- Originate from state transition diagrams that are used for the Automaton Theory.
- Transitions are supposed to represent actions which occur quickly and are not interruptible.
- States are supposed to represent longer-running activities.
- What constitutes “quickly” and “longer-running” depends on the application.

Statechart Diagram Elements

- A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
  - Name
  - Entry/exit action
  - Internal transition
  - Substate
  - Deferred event



do/stateDiagramName(parameterList) – "calls" another state diagram
entry/action -- carry out the action when entering the activity
exit/action -- carry out the action when exiting

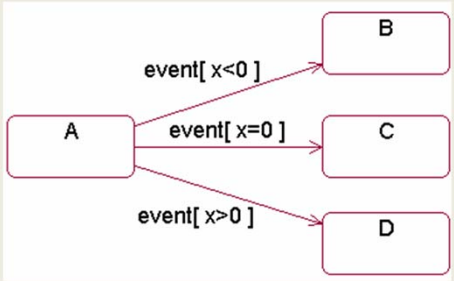
Statechart Diagram Elements

- A transition is a relationship between two states indicating that an object in the first state will perform specified actions and enter the second state when a specified event occurs and specified guard conditions are satisfied.

- Format:
  - event-signature '[' guard-condition ']' '/' action
    - event-name '(' comma-separated-parameter-list ')'

Statechart Diagram Elements

- Actions are processes that occur quickly and are not interruptible
- Conditions (Guard) return a logical *True/False* – the transitions occurs on *True*
- When there is no Event in the label the transition occurs as soon as the activities of the current state are completed.

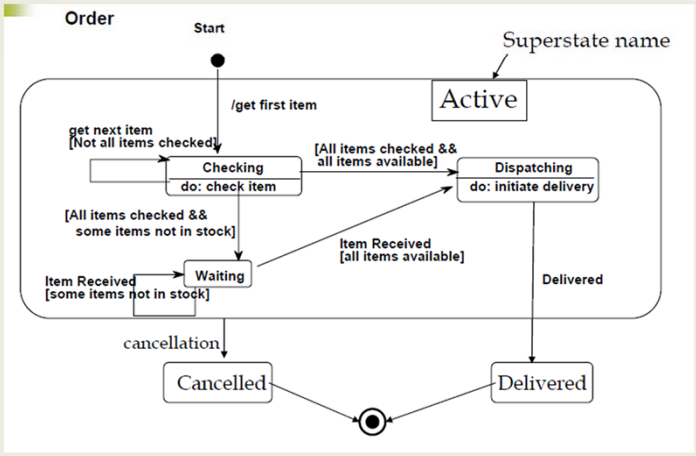


Statecharts and Relationship to Other Diagrams

- Events on the statechart diagram should appear as an incoming message for the appropriate object on a sequence and collaboration diagram.
- A statechart diagram is prepared for every object class in the Class Diagram with non-trivial behavior.
- Every event should correspond to an operation on the appropriate class.

Superstates and Supertransitions

- Where transitions may occur from several states to a single state the Statechart Diagram may show this single state as a Superstate



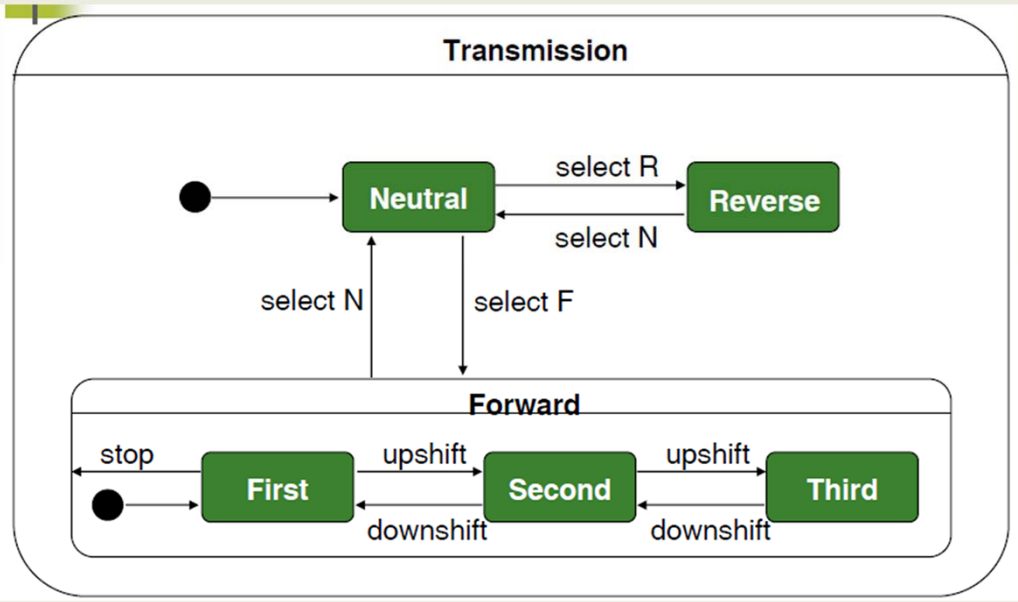
Supertransitions

- Transition to superstate boundary ≡ transition to initial state of the superstate.
  - entry actions of all regions entered are performed
- Transition from a superstate boundary ≡ transition from the final state of the superstate
  - exit actions of all regions exited are performed
- There may also be transitions directly into a complex state region (like program “gotos”).

Composite, Super/Sub states

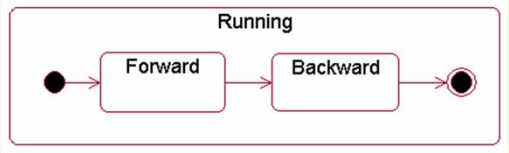
- For UML statecharts diagrams, states can be composed into nested states. Such compositions make it possible to view a state at different levels of abstraction.
- A **composite state** is a state that consists of either concurrent substates or sequential substates.
- A **substate** is a state that is nested inside another one.

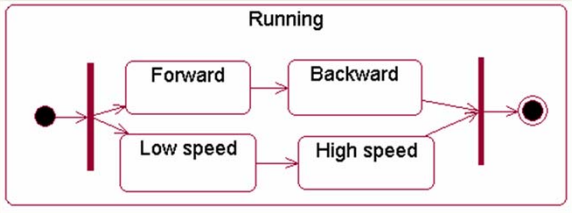
Composite, Super/Sub states



Substates

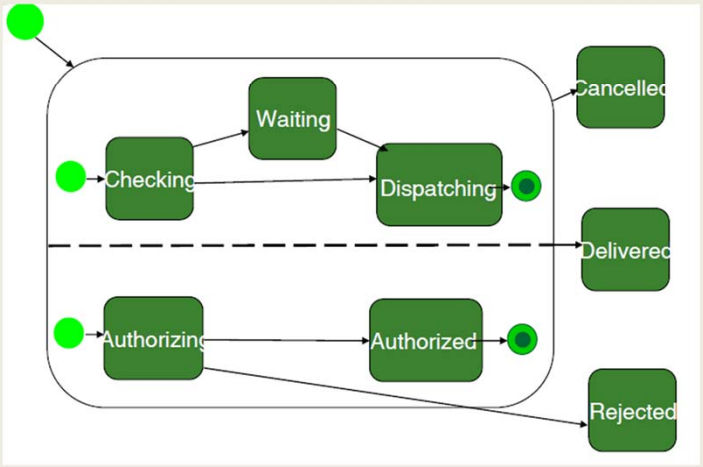
- “and” “or” relationships





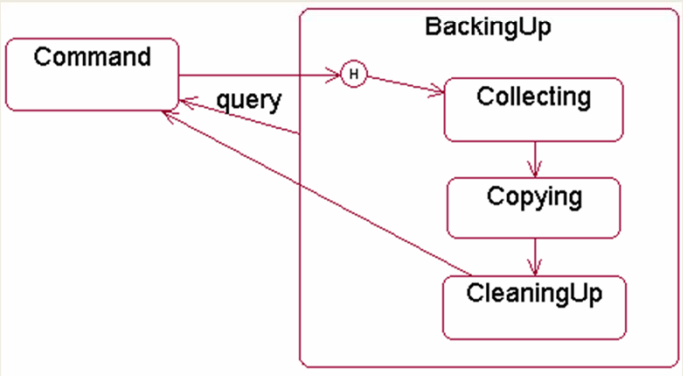
Concurrent States

- Used when a given object has sets of independent behaviours
- When the superstate is “on”, all of their component states are “on”



History State

- A pseudostate whose activation restores the previously active state within a composite state
  - H, H*

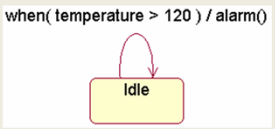


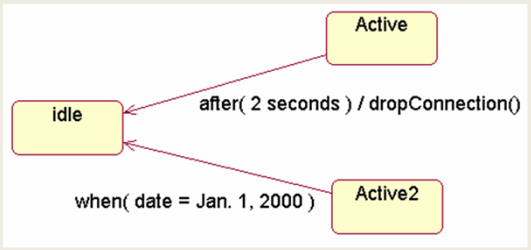
H: outest, latest

H*: any layer

Statechart Diagram Elements

- An event is the specification of a noteworthy occurrence that has a location in time and space.
  - Call event
    - event_name([parameter list, …])
  - Change event
    - Keyword: when
  - Time event
    - Keyword: after/when
  - Signal event
    - Same as call event





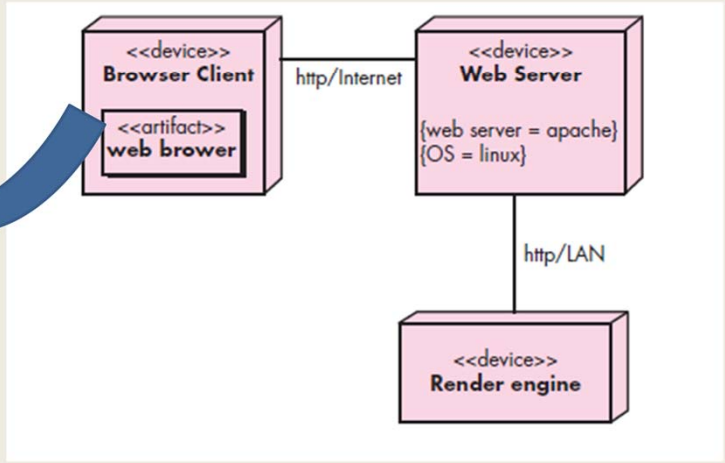
Statechart Diagram Elements

- An action is an executable atomic computation.
  - entry/exit action
- Relation to other diagrams:
  - Every action should correspond to the execution of an operation on the appropriate class.
  - Entry/Action, Exit/Action and Do/Activity within a State will normally be equivalent to Operations in the Class Diagram.
  - Every outgoing message sent from a statechart must correspond to an operation on another class.

Component level design – step 6

6. Elaborate deployment diagrams to provide additional implementation detail.

- Additional details
  - Location of key packages of components
  - Hardware
  - OS



During component-level design, deployment diagrams can be elaborated to represent the location of key packages of components.

The specific hardware and operating system environment(s) that will be used is (are) specified and the location of component packages within this environment is indicated

Component level design – step 7

7. Factor every component-level design representation and consider alternatives.

- Iterative nature of design
- Refactor as needed
- Consider all possible alternatives (design solutions) before finalization

Design is an iterative process!

The first component-level model we create will not be as complete, consistent, or accurate as the *n*th iteration you apply to the model.

It is essential to refactor as design work is conducted

you should not suffer from tunnel vision. There are always alternative design solutions, and the best designers consider all (or most) of them before

settling on the final design model.

Develop alternatives and consider each carefully, using the design principles and concepts

Object Constraint Language (OCL)

- Complements UML by allowing a software engineer to use a formal grammar and syntax to construct unambiguous statements about various design model elements
- Simplest OCL language statements are constructed in four parts:
  - (1) a *context* that defines the limited situation in which the statement is valid;
  - (2) a *property* that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute)
  - (3) an *operation* (e.g., arithmetic, set-oriented) that manipulates or qualifies a property, and
  - (4) keywords (e.g., if, then, else, and, or, not, implies) that are used to specify conditional expressions.

OCL Example

```
context PrintJob::validate(upperCostBound : Integer,
custDeliveryReq :
    Integer)
  pre:  upperCostBound > 0
      and custDeliveryReq > 0
      and self.jobAuthorization = 'no'
  post: if self.totalJobCost <= upperCostBound
        and self.deliveryDate <= custDeliveryReq
      then
        self.jobAuthorization = 'yes'
      endif
```

Summary

- Component-level design
  - Seven steps
  - Statechart diagrams