2d. The Unified Modeling Language

Use Case Diagrams
Class Diagrams
Attributes, Operations and Constraints
Generalization and Aggregation
Sequence and Collaboration Diagrams
State and Activity Diagrams

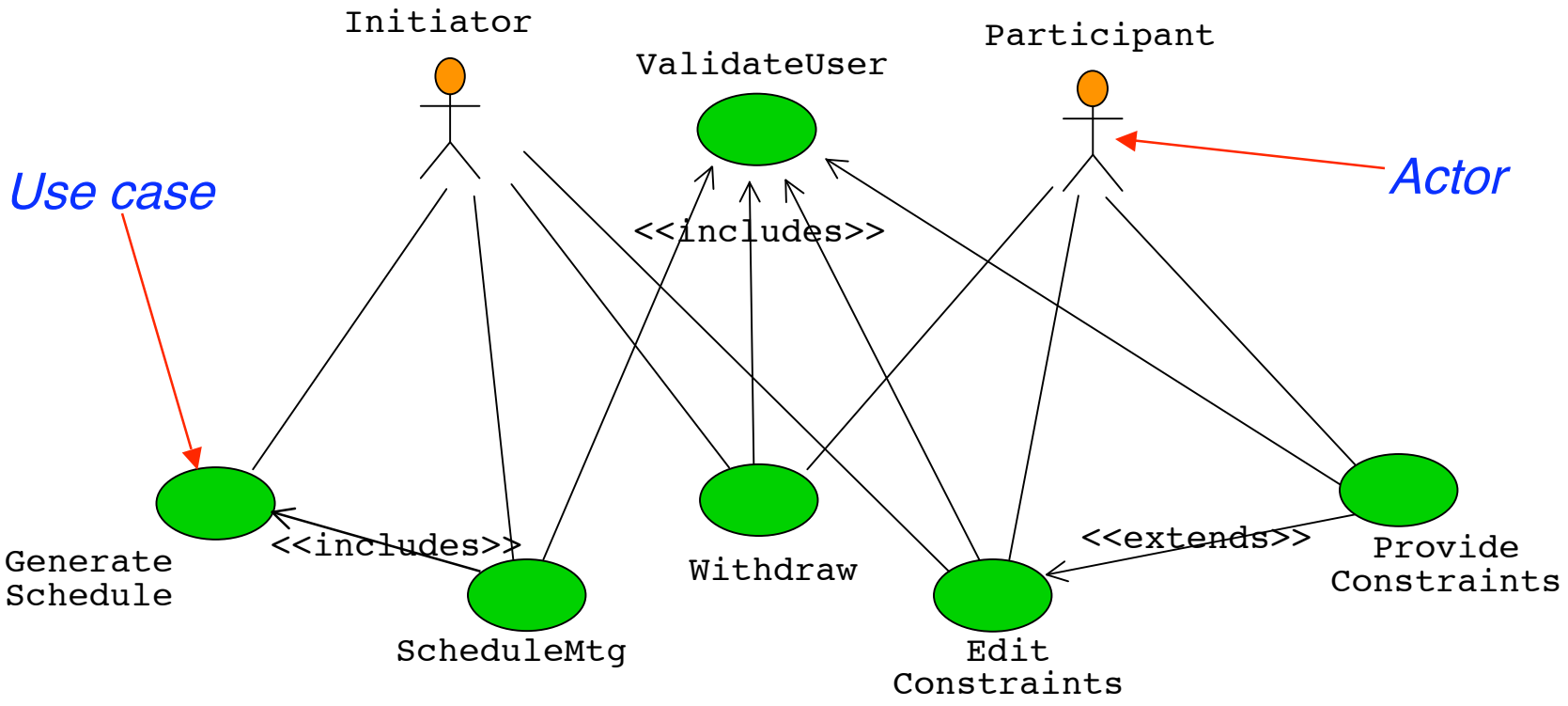
UML Diagrams

- UML was conceived as a language for modeling software. Since this includes requirements, UML supports world modeling (...at least to some extend).
- UML offers a variety of diagrammatic notations for modeling static and dynamic aspects of an application.
- The list of notations includes use case diagrams, class diagrams, interaction diagrams -- describe sequences of events, package diagrams, activity diagrams, state diagrams, …more...

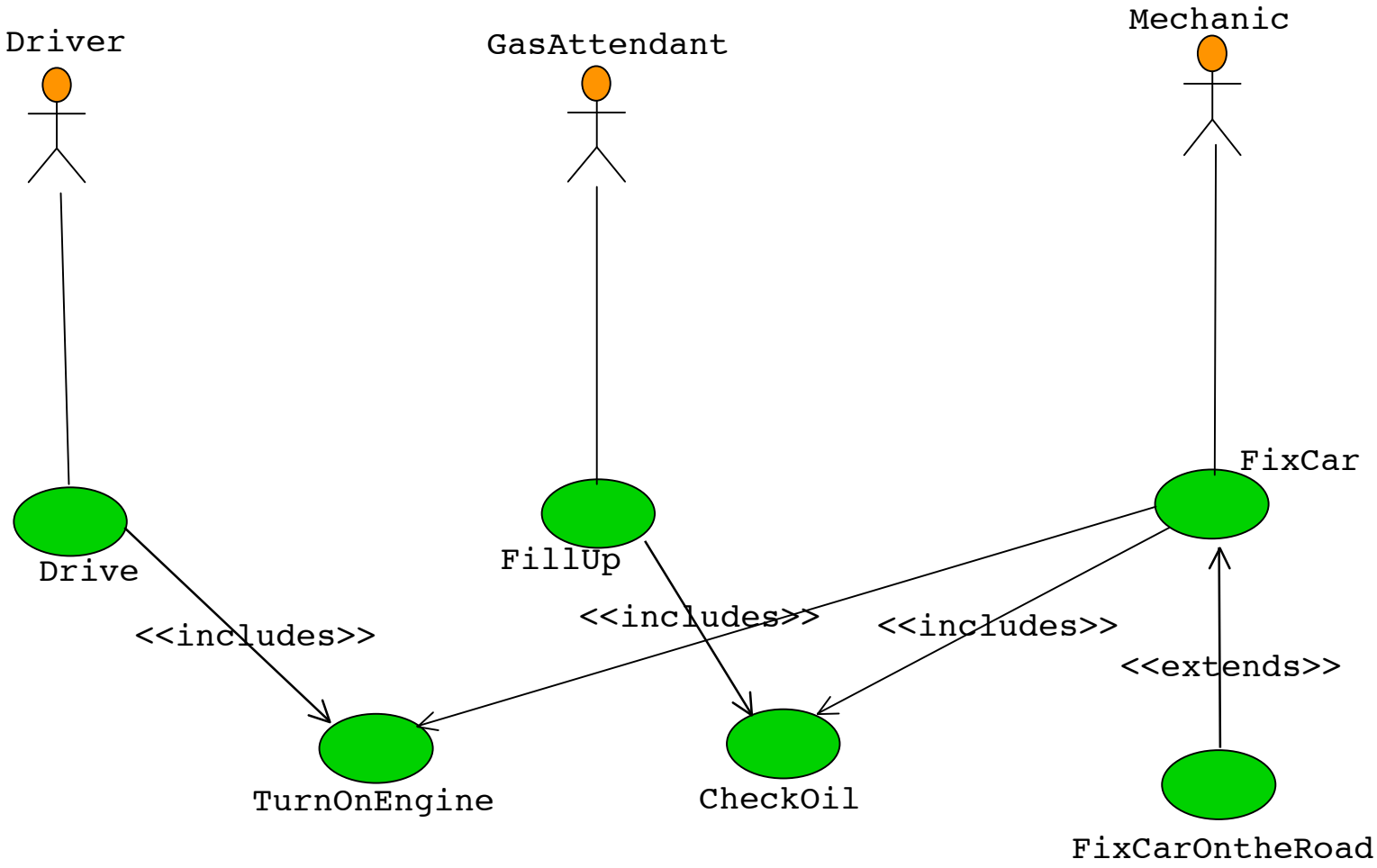
Use Case Diagrams

- A use case [Jacobson92] represents "typical use scenaria" for an object being modeled.
- Modeling objects in terms of use cases is consistent with Cognitive Science theories which claim that every object has obvious suggestive uses (or affordances) because of its shape or other properties. For example,
  - ✓ Glass is for looking through (...or breaking)
  - ✓ Cardboard is for writing on...
  - ✓ Radio buttons are for pushing or turning…
  - ✓ Icons are for clicking…
  - ✓ Door handles are for pulling, bars are for pushing…
- Use cases offer a notation for building a coarse-grain, first sketch model of an object, or a process.

Use Cases for a Meeting Scheduling System



Use Cases for a Car



Use Cases

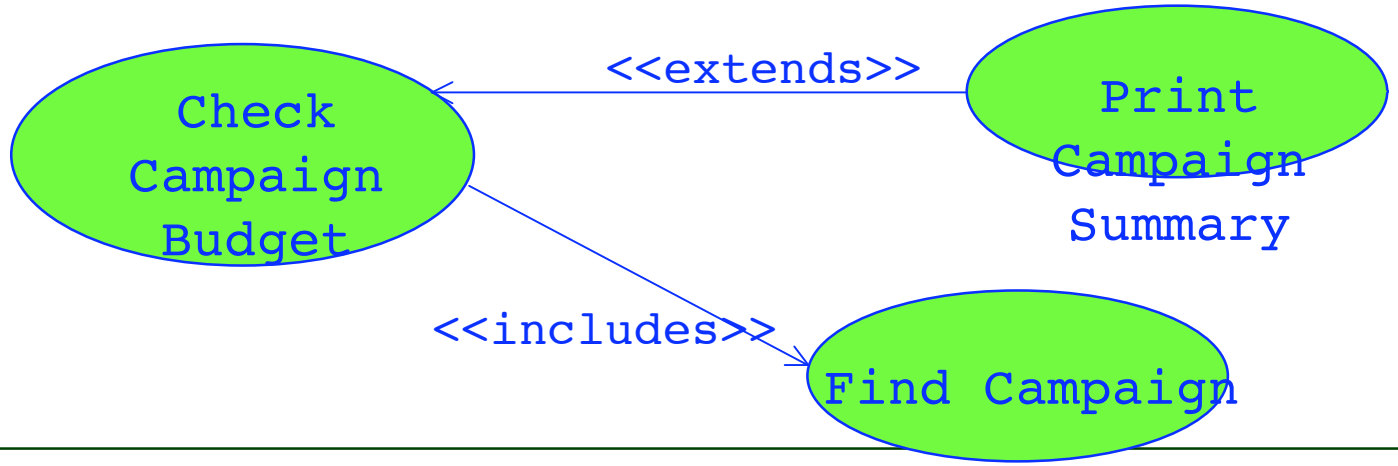
- Use cases may represent user goals, or user interactions; for example, `ScheduleMtg` can be thought as a goal (there are many ways to schedule a meeting), but `ValidateUser` is probably not.
- Use cases make sense for usable things, such as designed artifacts, including processes; they don't make sense for unusable things (e.g., the sky).
- (Consequently) Use cases constitute a special-purpose modeling construct for software or other artifacts.
- [The notion of scenario, as a typical course of actions or events, is probably more appropriate for a general purpose modeling language.]

Features of Use Cases

- An actor is a role that a user plays with respect to the object being described; don't think of actors as either users (e.g., Maria), or positions (e.g., department chair).
- (When do I stop??…) For any one software development project, you probably don't want more than 100 use cases.

<<extends>> vs <<includes>>

- <<extends>> implies that one use case adds behaviour to a base case; used to model a part of a use case that the user may see as optional system behavior; also models a separate sub-case which is executed conditionally.
- <<includes>>: adds behavior to a base case (like a procedure call); used to avoid describing the same flow of events several times, by putting the common behavior in a use case of its own.



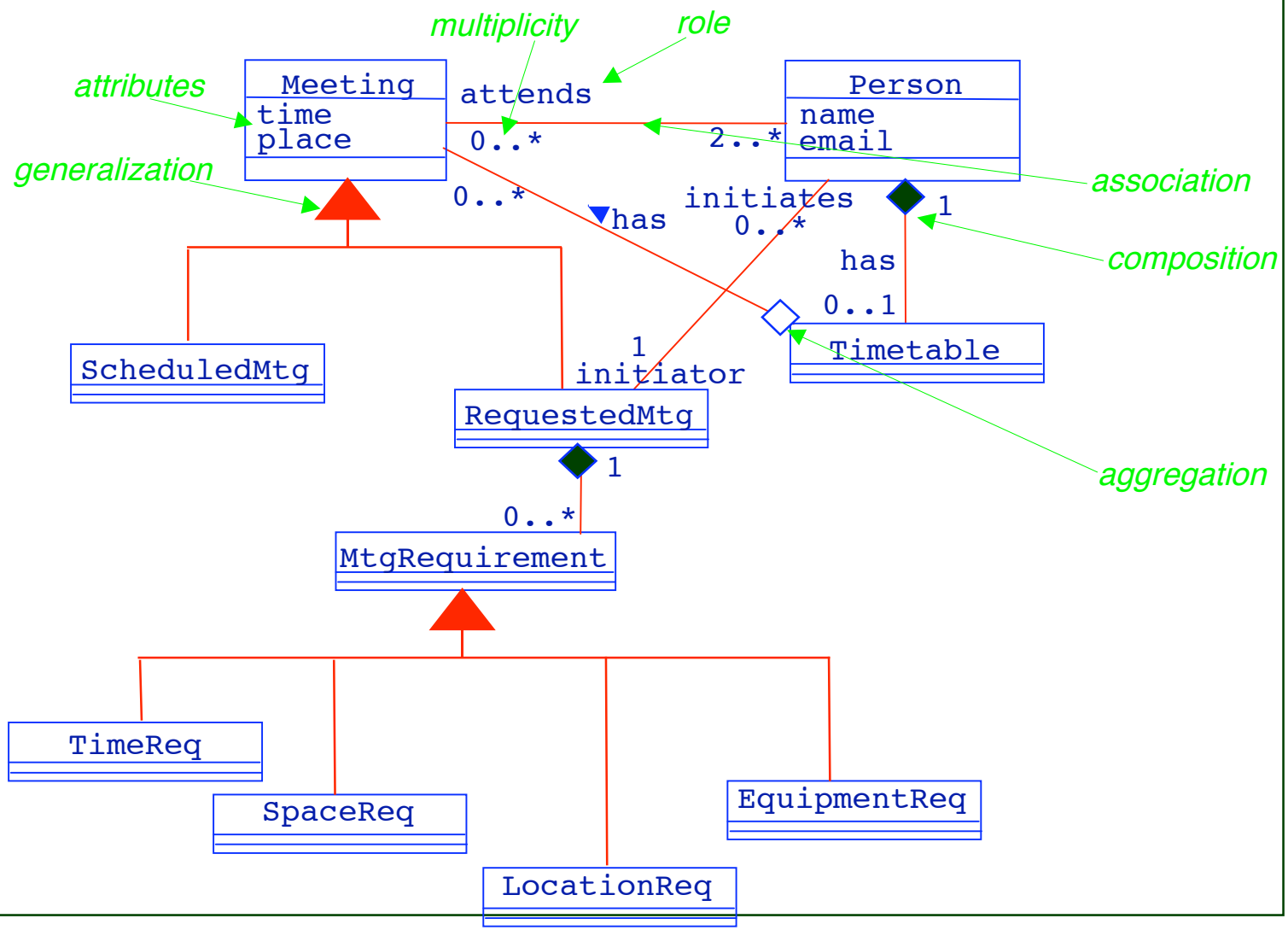
Class Diagrams

- Class diagrams describe the kinds of objects found in the application, and their inter-relationships.
- There are two types of inter-relationships: associations and subtypes [Fowler97]
- Class diagrams are basically an adaptation of EER diagrams, with some minor differences.
- UML class diagrams may model some part of the real world (e.g., the world of meetings and schedulings), a design specification (e.g., for a system that does meeting scheduling), or an implementation.

Comparison: EER vs Class Diagrams

- EER diagrams allow N-ary relationships, N≥2; Class diagrams only allow binary relationships.
- EER diagrams allow multi-valued attributes, class diagrams do not.
- EER diagrams allow the specification of identifiers (an often-encountered type of constraint), while class diagrams do not.
- Class diagrams allow dynamic classification, but EER diagrams do not.

Meeting Scheduling



Notes on Associations

- Associations represent semantic relationships. Each association can have up to two roles for participating objects. Each role can also have an associated cardinality range ("multiplicity").
- Associations represented with directed arrows are navigatable only in one direction; for example, if the `Meeting-Person` association was represented with an arrow pointing towards `Person`, this would indicate that from a `Meeting` we can navigate to all meeting persons, but given a person, we can't find all the meetings she has participated in.

Notes on Attributes

- Attributes are always single-valued.
- Attributes can have an associated type (a class), a default value, and a visibility value of + (public), # (protected) and - (private). They can also be derived (`/attr`) or not.
- There is no semantic difference in UML between attributes and directional associations.
- Other models treat attributes as associations with an existence constraint which says: if an object is deleted, so are its attributes and their values.

Notes on Operations

- These are "the processes a class knows how to carry out" [Fowler97, p63]. They are specified in the third layer of a class box.
- Specification includes a visibility value, name, parameter list and returned value type.
- For conceptual modeling, Fowler argues -- rather vaguely -- that operations should be used to define the responsibilities of a class.
- It makes better sense to distinguish a subclass of objects -- agents/positions/roles -- which can participate in activities, and describe for each the activities they know how to carry out.

Operations and Constraints

Meeting
+ time: Time = 9am + place: Place = LP266
+ schedule(ConstrLst):Meeting + cancel() if place=SF2201 then time≠12pm; if Meeting.initiator is Disabled then place is AccessibleRm;

- How do you say that "If one of the participants is disabled, then the place must be disabled-accessible??
- We need some sort of a First Order language for constraints:

```
if (exists p:Meeting.participant) p is Disabled
                        then place is AccessibleRm
```

Object Constraint Language (OCL)

- Some constraints can be adequately expressed in the graphical language (ex. cardinality of an association).
- Some can not. For example, constraints within operation specifications (pre- and post-conditions)
- OCL expressions are constructed from a collection of pre-defined elements and types.
- The language has a formal syntax and semantics and supplements the expressiveness of UML.

[Warmer99] Warmer, J. Kleppe, A. *The Object Constraint Language: Precise Modeling with UML* Addison-Wesley 1999.

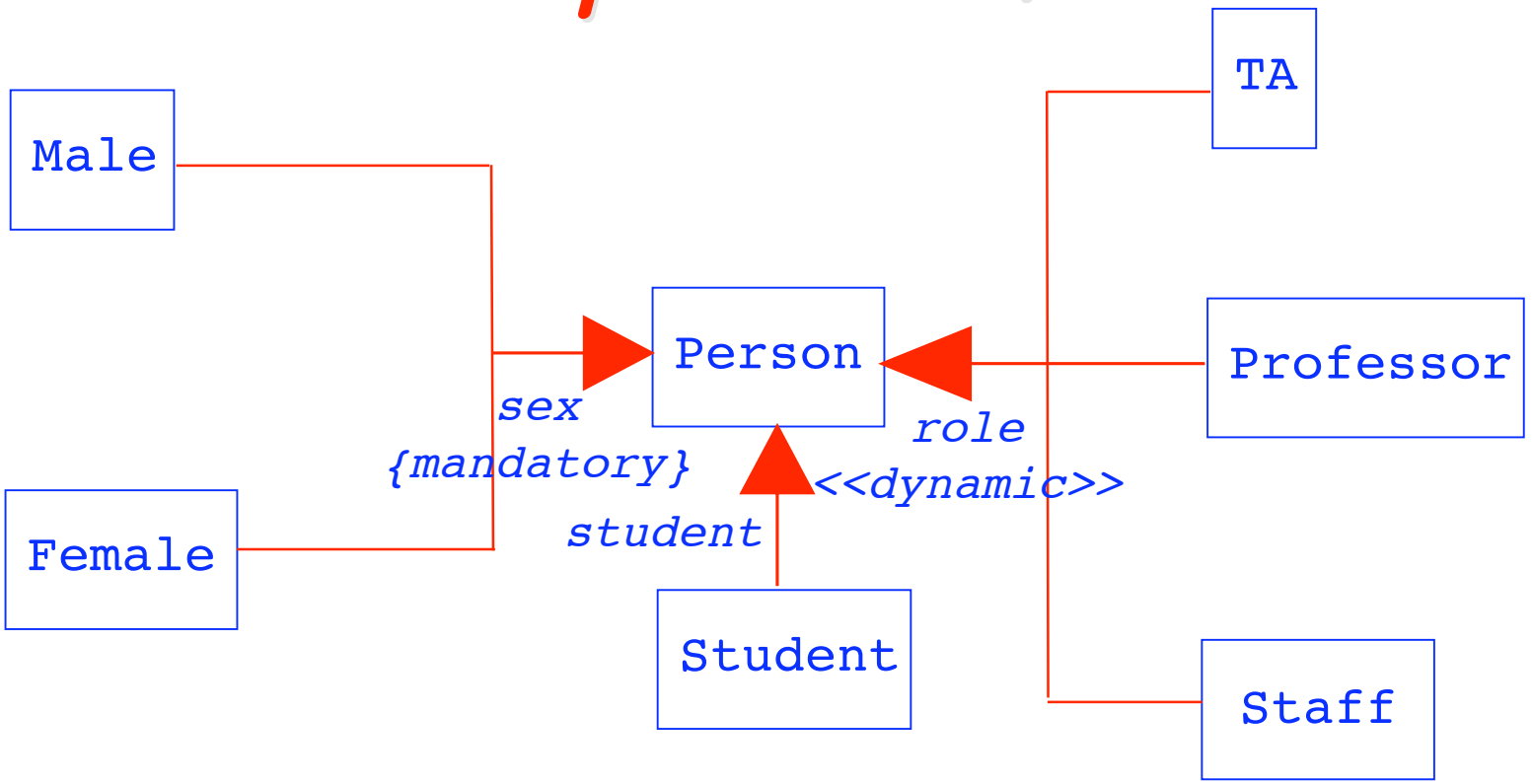
OCL Examples

OCL expression	Interpretation
Person self.age	In the context of a specific person, the value of the property 'age' of that person—i.e. a person's age.
Person self.income >= 5,000	The property 'income' of the person under consideration must be greater than or equal to 5,000.
Person self.wife->notEmpty **implies** self.wife.sex = female	If the set 'wife' associated with a person is not empty, then the value of the property 'sex' of the wife must be female. The boldface denotes an OCL keyword, but has no semantic import in itself.
Company self.employee->size <= 50	The size of the set of the property 'employee' of a company must be less than or equal to 50. That is, a company cannot have more than 50 employees.
Company self.employee->select (age > 50)	This specifies the set of employees of a company whose age is greater than 50.

Multiple and Dynamic Classification

- Classification refers to the relationship between an object and the classes it is an instance of.
- Traditional object models (e.g., Smalltalk, C++,…) assume that classification is *single* and *static*.
- Multiple classification allows an object to be an instance of several classes that are not is-a-related to each other; for example, `Maria` may be an instance of `GradStudent` and `Employee` at the same time.
- If you allow multiple classification, you want to be able to specify which combinations of instantiations are allowed. This is done through *discriminators*.
- *Dynamic* classifications allows an object to change its type during its lifetime.

Multiple Classification



- Mandatory means that every instance of Person must be an instance of Male or Female.
- <<Dynamic>> means that an object can cease to be a TA and may become a Professor.

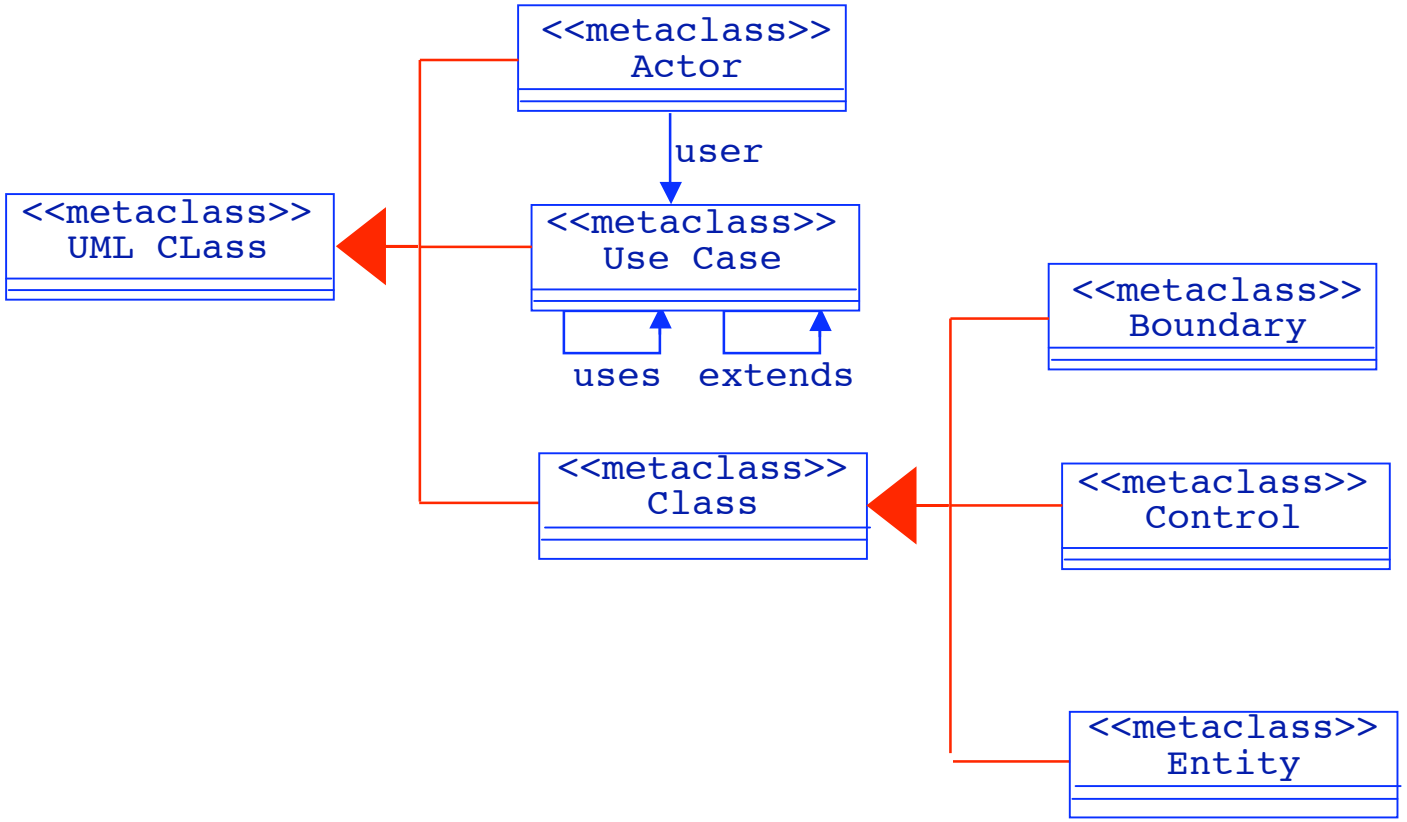
Generalization

- Multiple generalization involves a class which has two or more superclasses that are not is-a related. For example, TA is a specialization of Student and Employee.
- In UML, multiple generalization is allowed. Inheritance conflicts are resolved by predefined order of superclasses; renaming of attributes or operations is also allowed.
- For each discriminator, the associated collection of classes can be declared to be `complete/incomplete`, also `disjoint/overlapping`

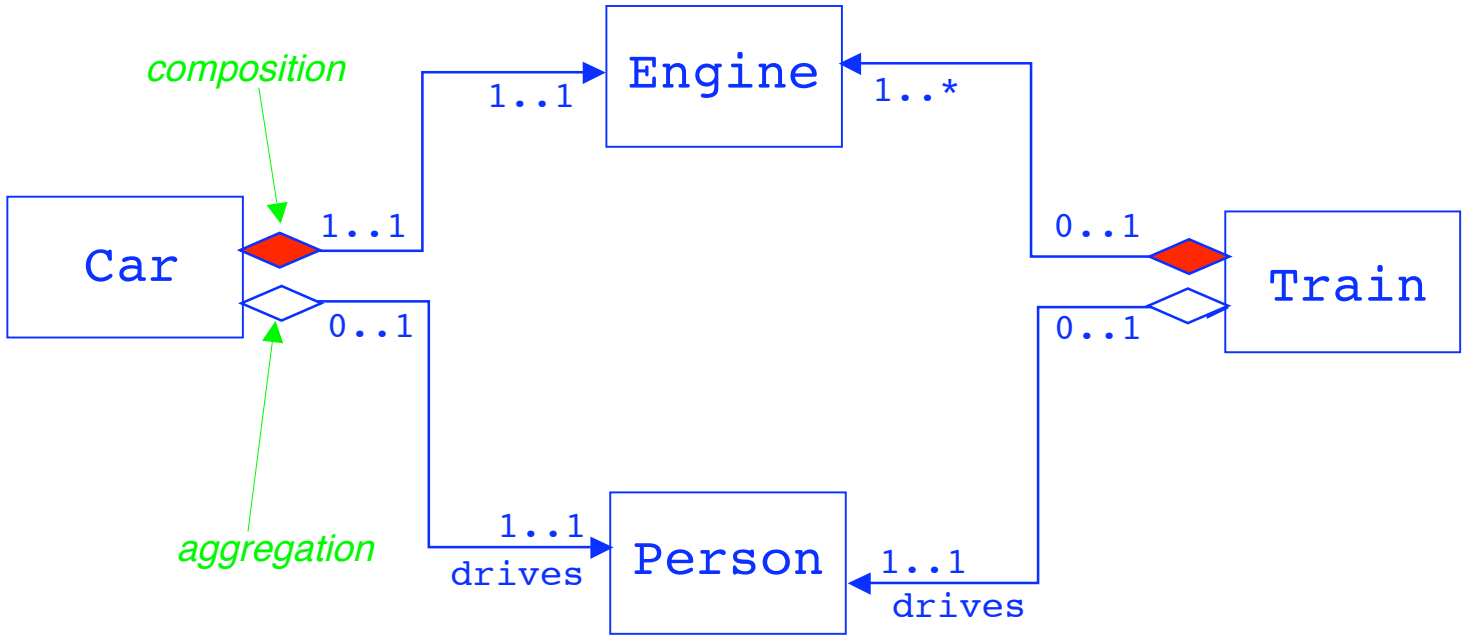
Stereotypes

- Stereotypes offer "a high level classification of an object…tell you the kind of object it is" [Fowler97].
- Stereotypes define the types of constructs that can be used in a UML diagram. You can think of them as offering a metamodel of UML diagrams, or as giving the graphical syntax of UML diagrams.
- In UML, stereotypes are shown delimited by <<…>>.
- Note that the stereotypes shown in class diagrams (such as `<<includes>>`, `<<extends>>`) are metaclasses which define the UML metamodel.
- One can extend UML by creating new stereotypes as specializations of built-in ones.

Stereotypes as Metaclasses



Aggregation



Notes on Aggregation

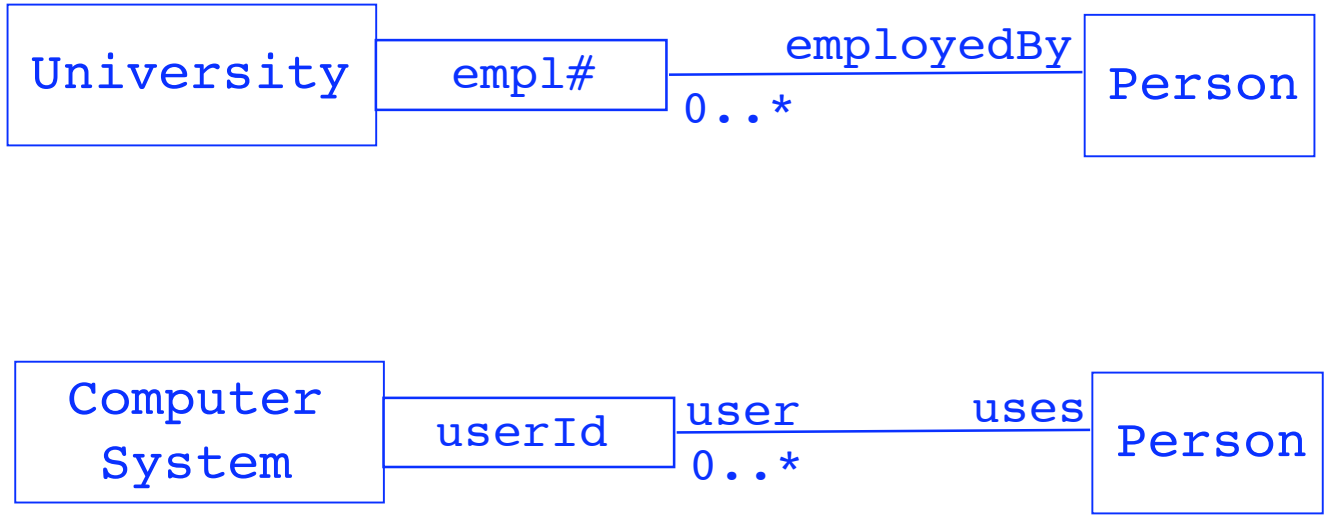
- Aggregation represents the `partOf` relationship.
- Composition is a strong form of aggregation, where a part can only participate in one composition relationship.
- Aggregation has been formalized in [Motschnig93], etc.; Every aggregation can be classified along two dimensions:
  - ✓ `Dependent/Independent` -- if an aggregation relation is dependent, then when you remove the whole you also remove the part;
  - ✓ `Shared/Exclusive` -- if an aggregation relationship is exclusive, a part can't be part of several wholes.
- So, composition amounts to a `dependent,exclusive` aggregation relationship.

Objects vs Values

- Values are mathematical objects, such as numbers, tuples, lists and sets. They come with their own equality predicate so that they can be compared. Values are immutable.
- (Reference) objects, on the other hand, have equality defined by their internal identifier. This means that two processes which have been running independently can never generate the same object, but may well be using the same values.
- Some conceptual models do make the distinction, [Fowler97] appears not to.
- The presence of values can influence (positively!) the semantics of attributes.

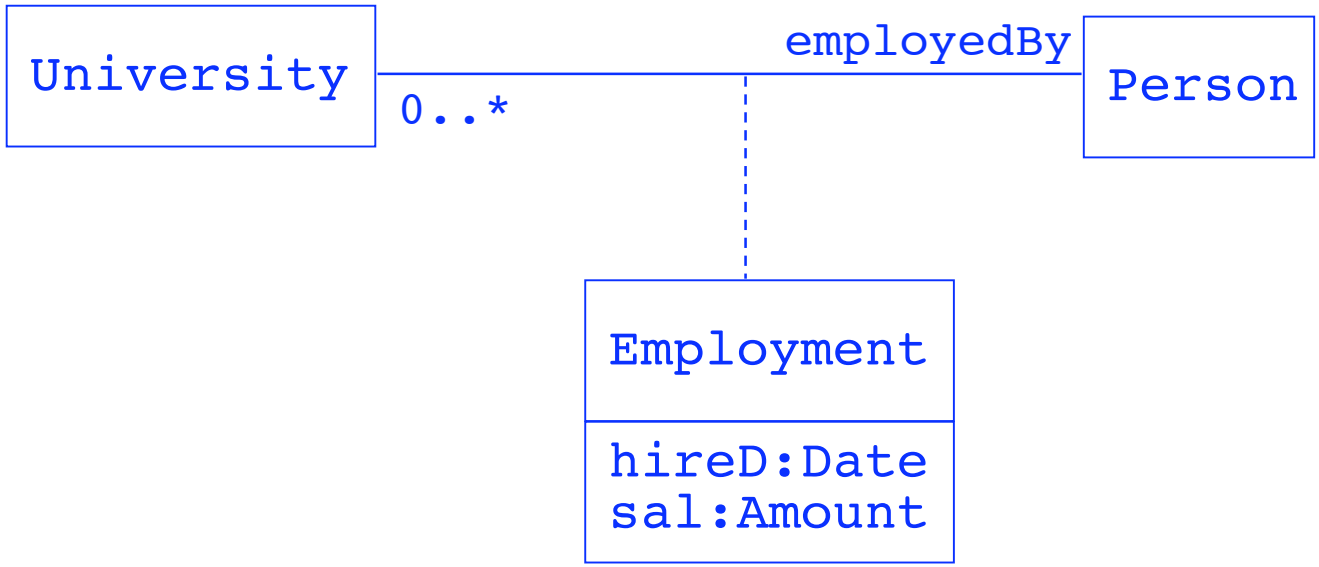
Qualified Associations

- Idea is that when you have a multi-valued association, you may have a key for all the values of that association.
- For instance:



Association Classes

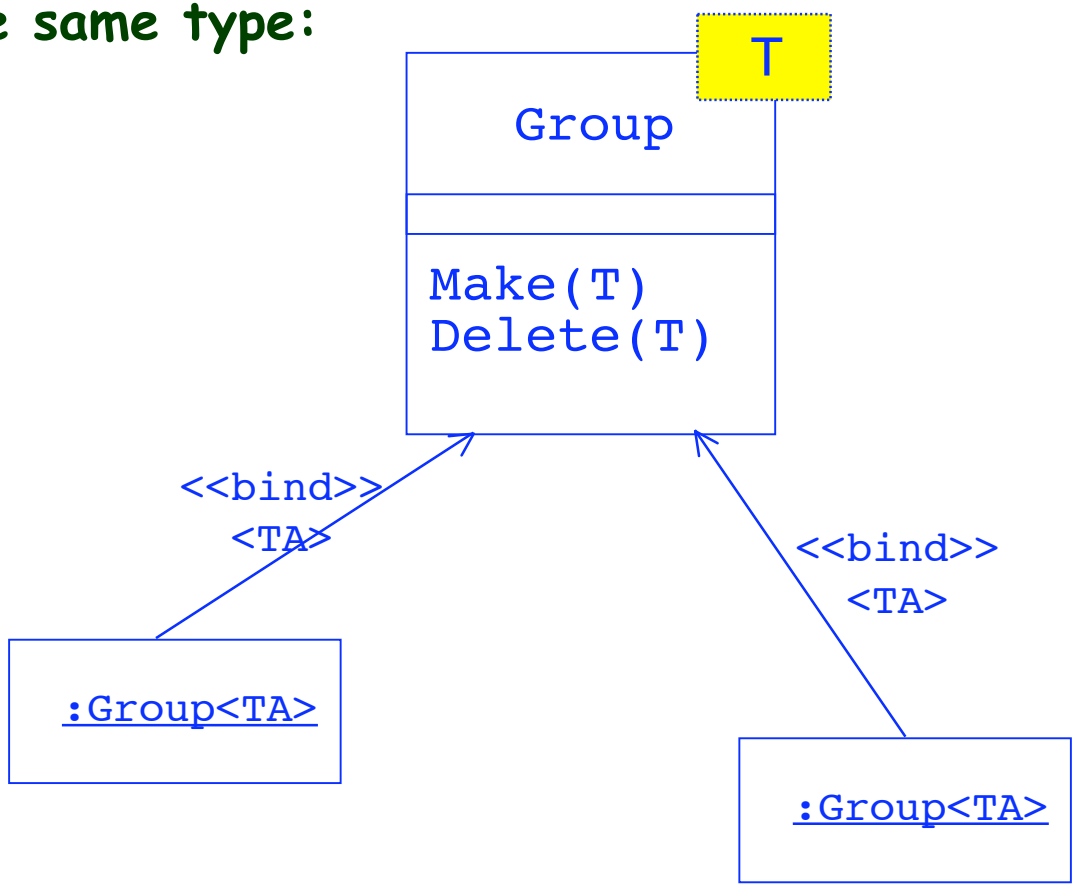
- Association classes allow you to treat associations as classes:



In UML you can only have a single instance of an association class for every pair of objects; this doesn't allow, for instance, several employments of the same person by the same employer.

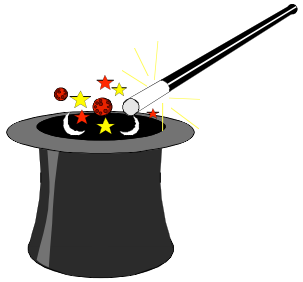
Template Classes

Template classes are parameterized classes; this construct is useful if you want to model groups or lists whose elements are all of the same type:



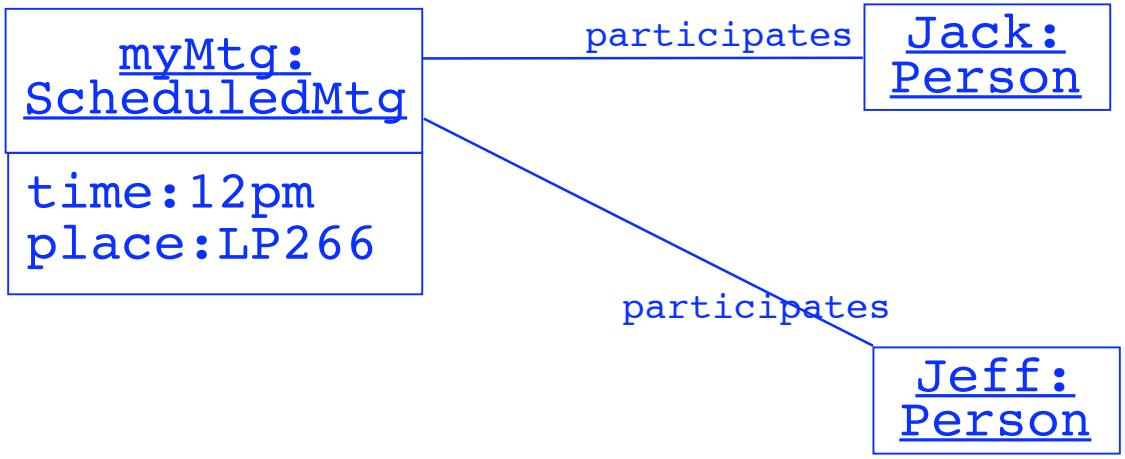
Visibility

- Private/public attributes and operations have obvious semantics.
- Protected attributes and operations can only be used by the owner class and its specializations.
- Question is: can one instance of a class see protected attributes of another instance of the same class?
- C++ allows this, Smalltalk does not. Smalltalk is obviously right...



Object Diagrams

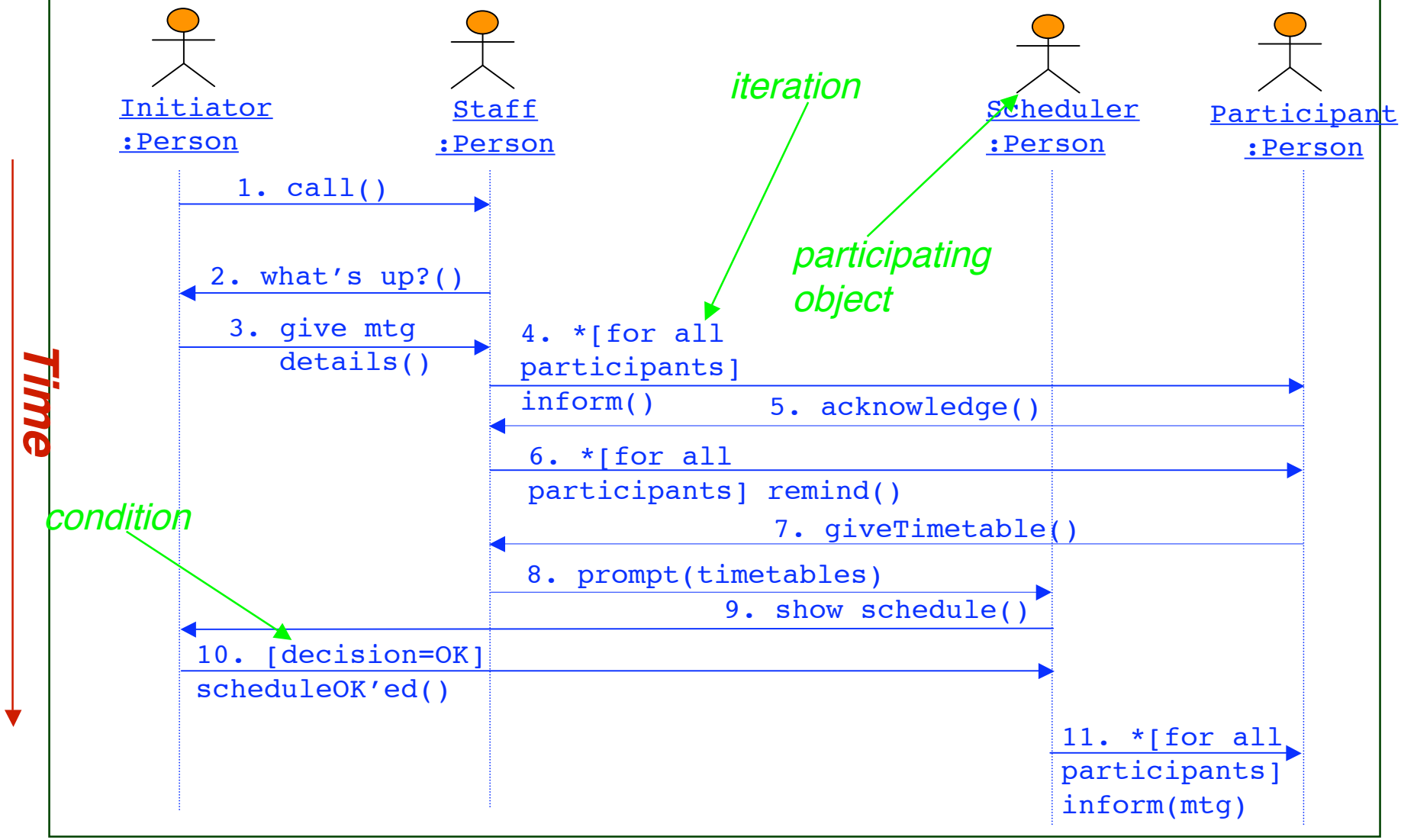
- These are like class diagrams, except now we model instances of the classes defined in class diagrams.
- Object diagrams are mentioned in [Gogolla98], but not in [Fowler97].



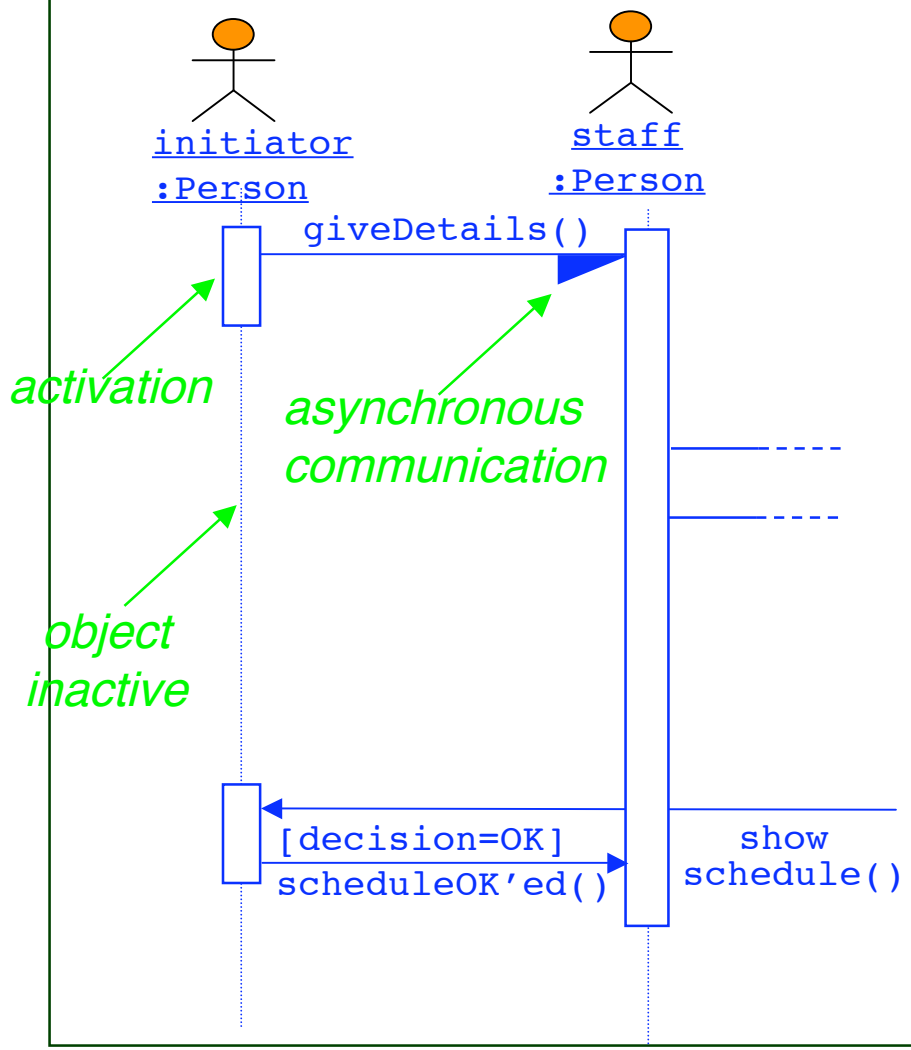
Interaction Diagrams

- Interaction diagrams capture interactions among objects.
- Typically, an interaction diagram models what happens for a single use case.
- An interaction diagram shows a number of example objects and the messages that are passed between them during the execution of the use case.
- There are two (comparable) types of interaction diagrams: sequence diagrams, and collaboration diagrams.
- Use icons to denote the objects participating in an interaction diagram (sequence or collaboration).

Sequence Diagram for ScheduleMtg

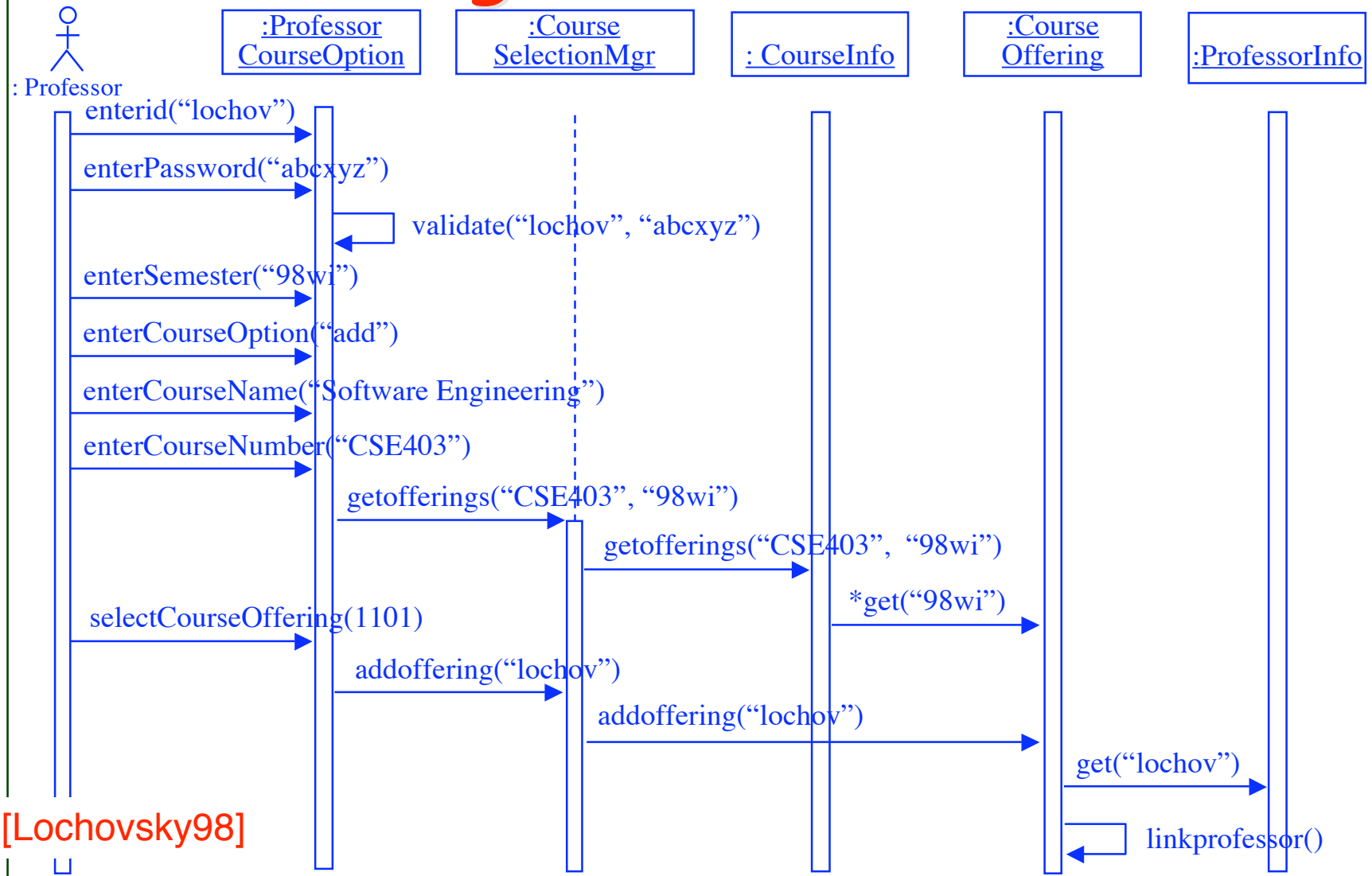


Concurrency and Synchronization



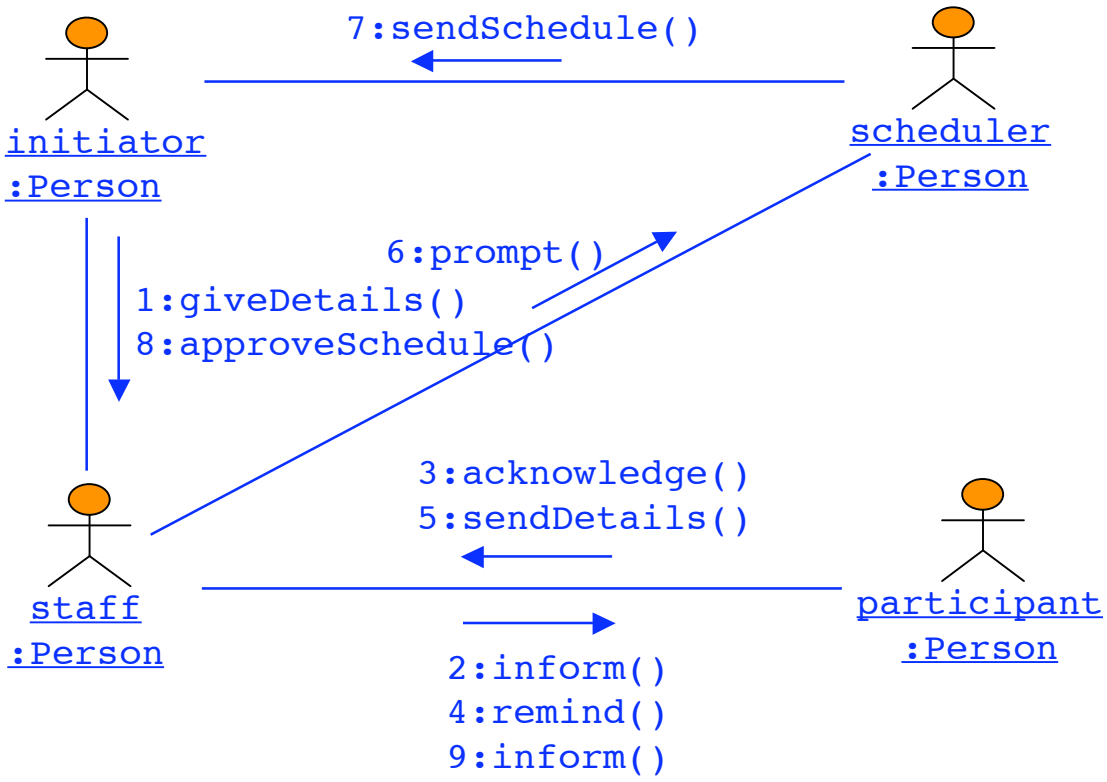
- Some of the features of sequence diagrams are useful for modeling concurrent computer processes, rather than for world modeling
- Statecharts are much more elegant for modeling concurrency.
- Numbering messages is optional for sequence diagrams.

Selecting a Course to Teach



[Lochovsky98]

Collaboration Diagrams



How to Use Collaboration Diagrams

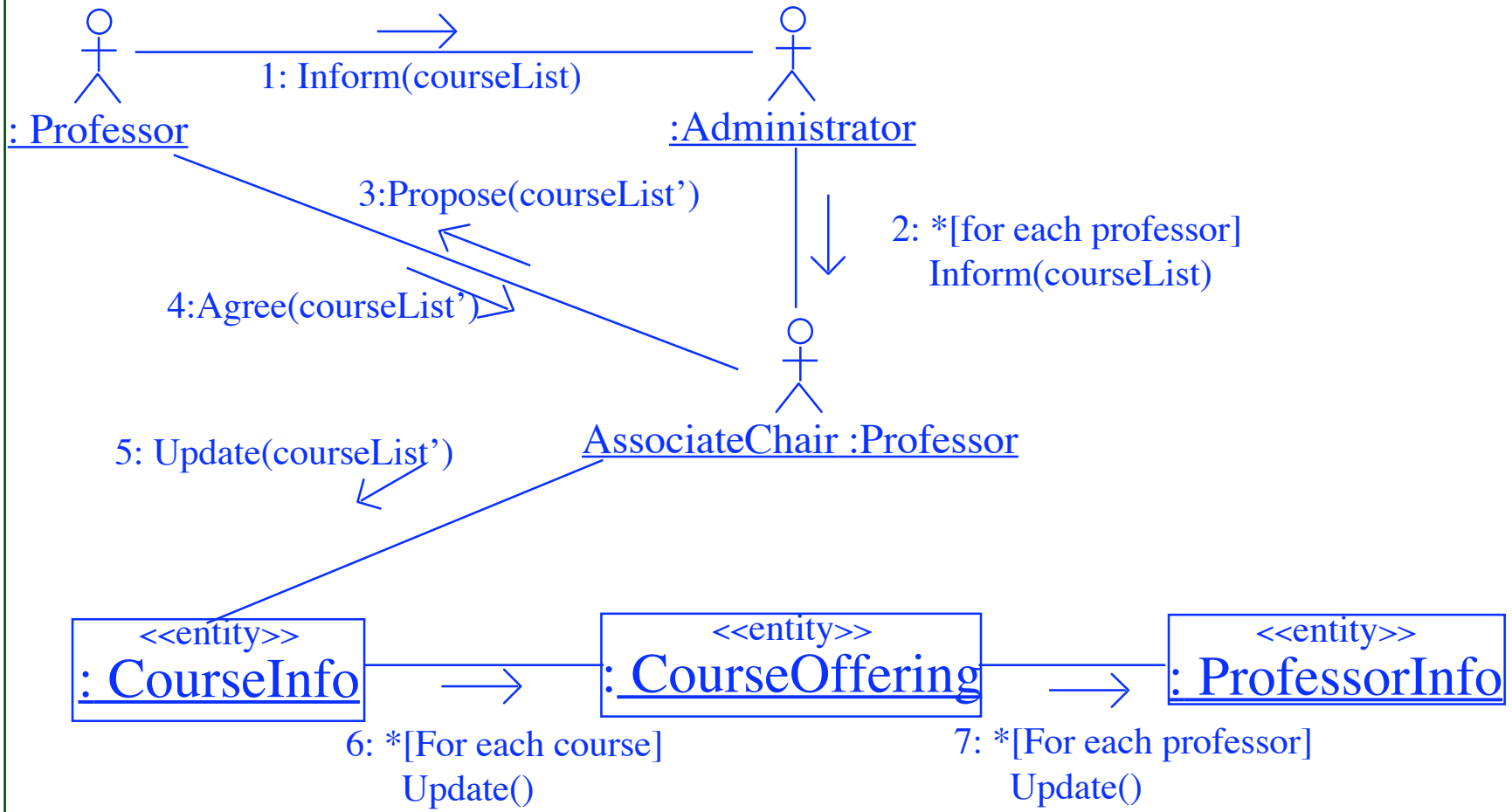
- Collaboration diagrams model scenaria; each scenario describes a possible sequence of events and actions.
- For a complex process, use several collaboration diagrams; make sure each collaboration diagram is simple and easy to understand.
- A special designation is available for objects which are created or destroyed during a collaboration:
  - ✓ Created during execution of the collaboration

    `:Employee{new}`
  - ✓ Destroyed during execution

    `:Employee{destroyed}`
  - ✓ Created and destroyed

    `:Employee{transient}`

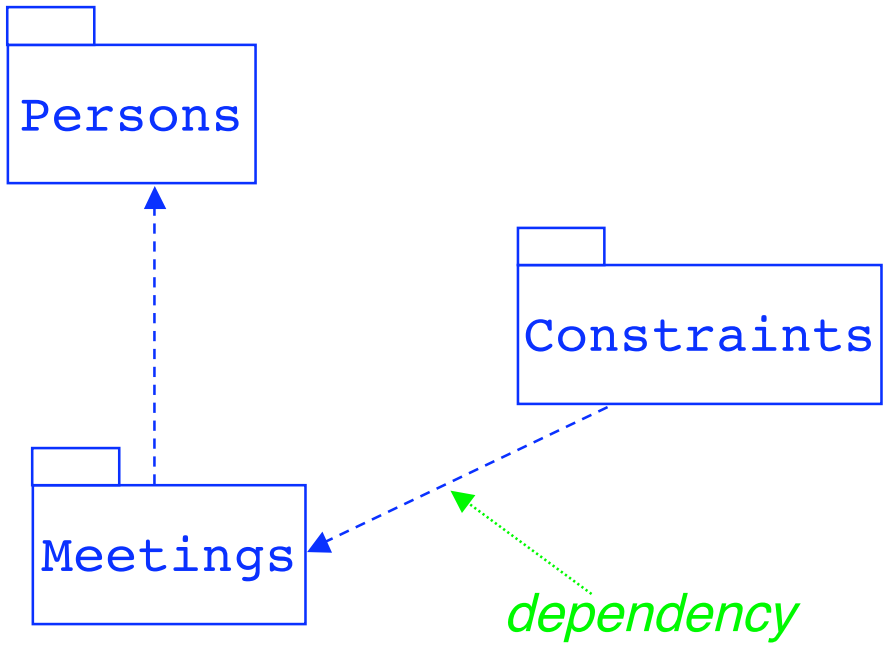
Select Courses to Teach



Packages

- Packages allow one to define useful subsets of a model to facilitate understanding and other modeling tasks.
- There are many criteria to use in defining the subsets:
  - ✓ Ownership -- who is responsible for which diagrams;
  - ✓ Application -- each application has its own obvious partitions; e.g., a university department model may be partitioned into staff, courses, degree programmes,…
  - ✓ Use -- clusters of classes used together, e.g.,

A Package Diagram

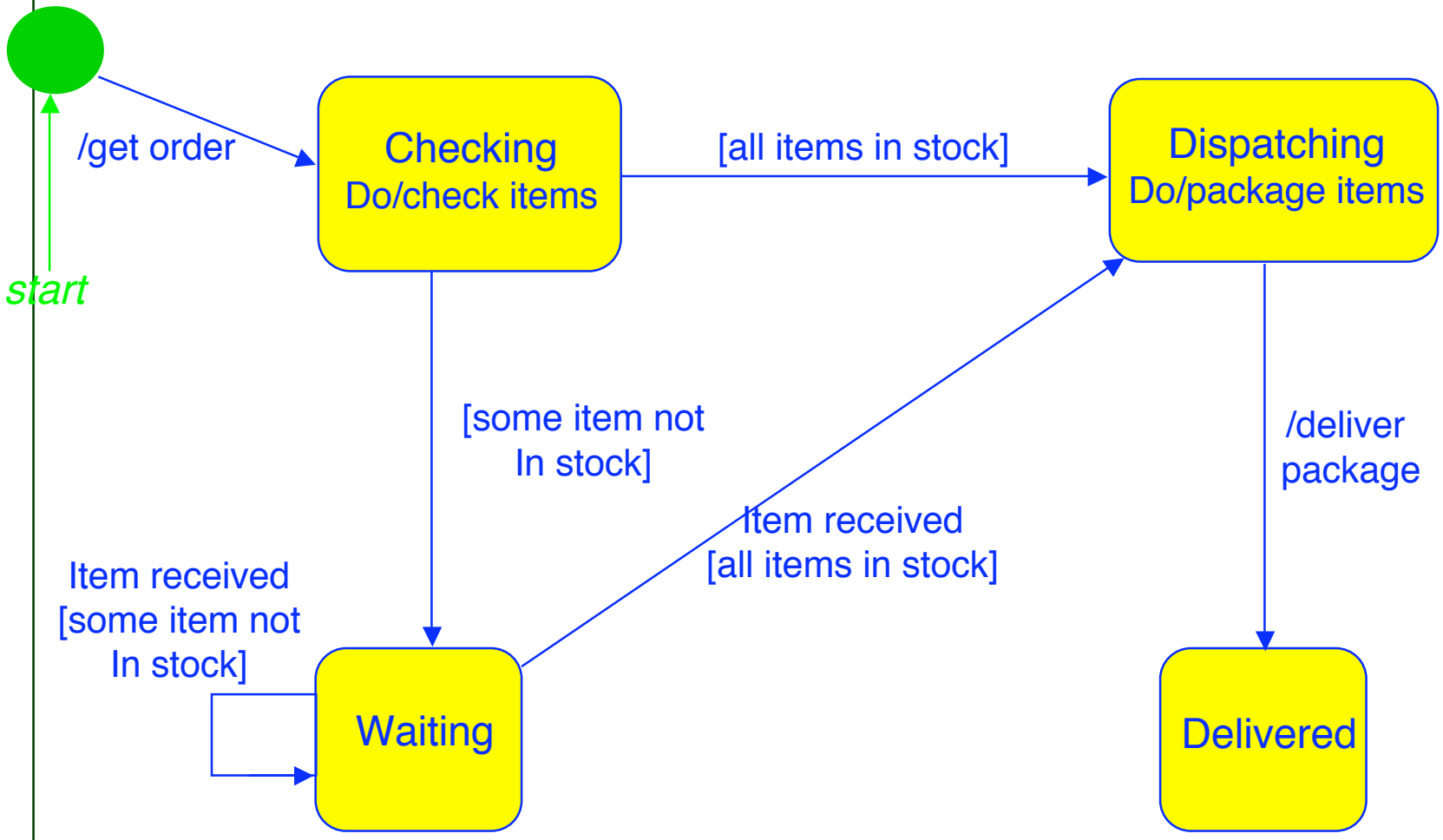


- A dependency means that if you change a class in one package, you may have to change something in the other.
- The concept is similar to compilation dependencies.
- It's desirable in UML to

State Diagrams

- These are state transition diagrams and describe the lifetime of some object (a person, a student,…)
- Transitions are supposed to represent actions which occur quickly and are not interruptible. States represent longer-running activities.
- A transition can have an associated `event [guard] action`, all of which are optional.
- When a transition has no associated event, it fires as soon as its source state activity is done.
- State diagrams can have superstates, consisting of several states. This is basically the statechart

State Diagram for Purchase Order

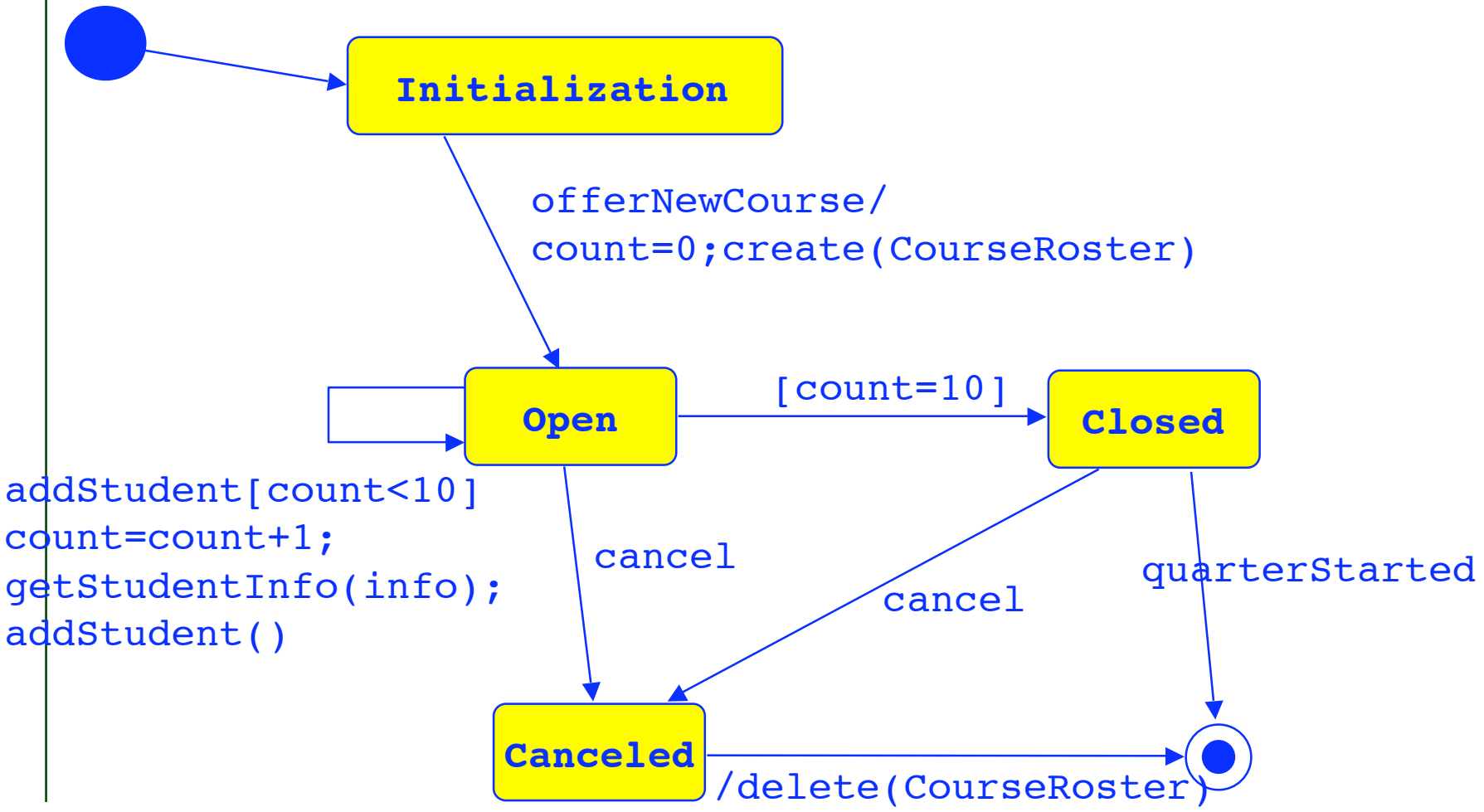


Events

UML distinguishes four different types of events:

- Change events designate when a condition becomes true
  E.g., `when(balance < 0)`
- Signal events designate the receipt of an explicit (real-time) signal from one object to another
- Call events indicate the receipt of a call for an operation by an object (request events would be more appropriate for non-software modeling)

Course Lifetimes



[Lochovsky98]

States

- A state represents a time period in the life of an object during which the object satisfies some condition, performs some action or waits for an event.
- In general a state can be characterized by a predicate which is true while the state is "on".
- Such a predicate may be defined in terms of:
  - ✓ The value(s) of one or more attributes of the class
    E.g., a person's address
  - ✓ The existence of a link to another object
- (In OO folklore...) The interval between two

Activities

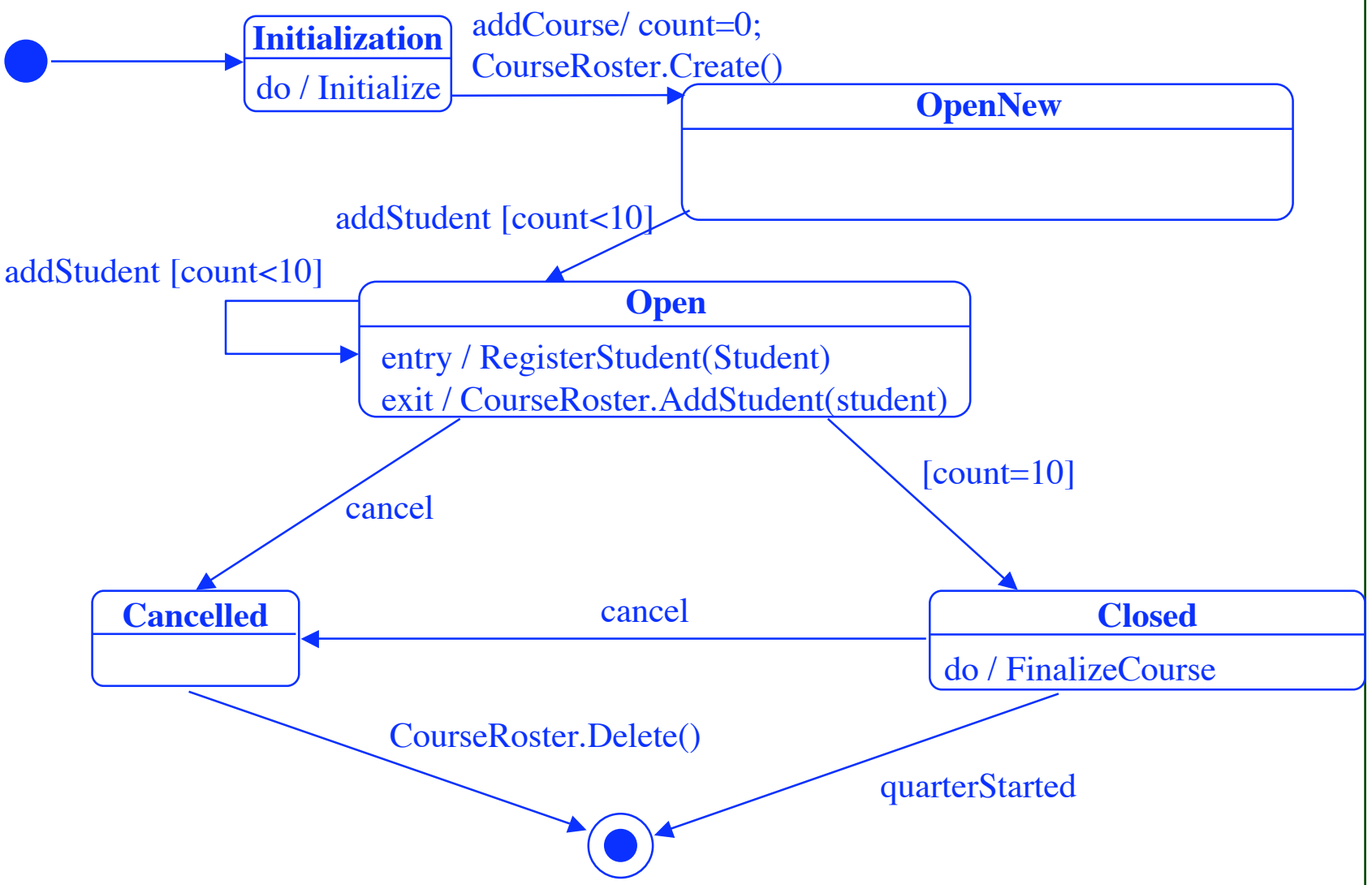
- Some states represent the lifetime of an activity that takes time to complete
  - ✓ starts when a state is entered
  - ✓ either completes or is interrupted by an event that causes an outgoing transition
- Special activity constructs:

  `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

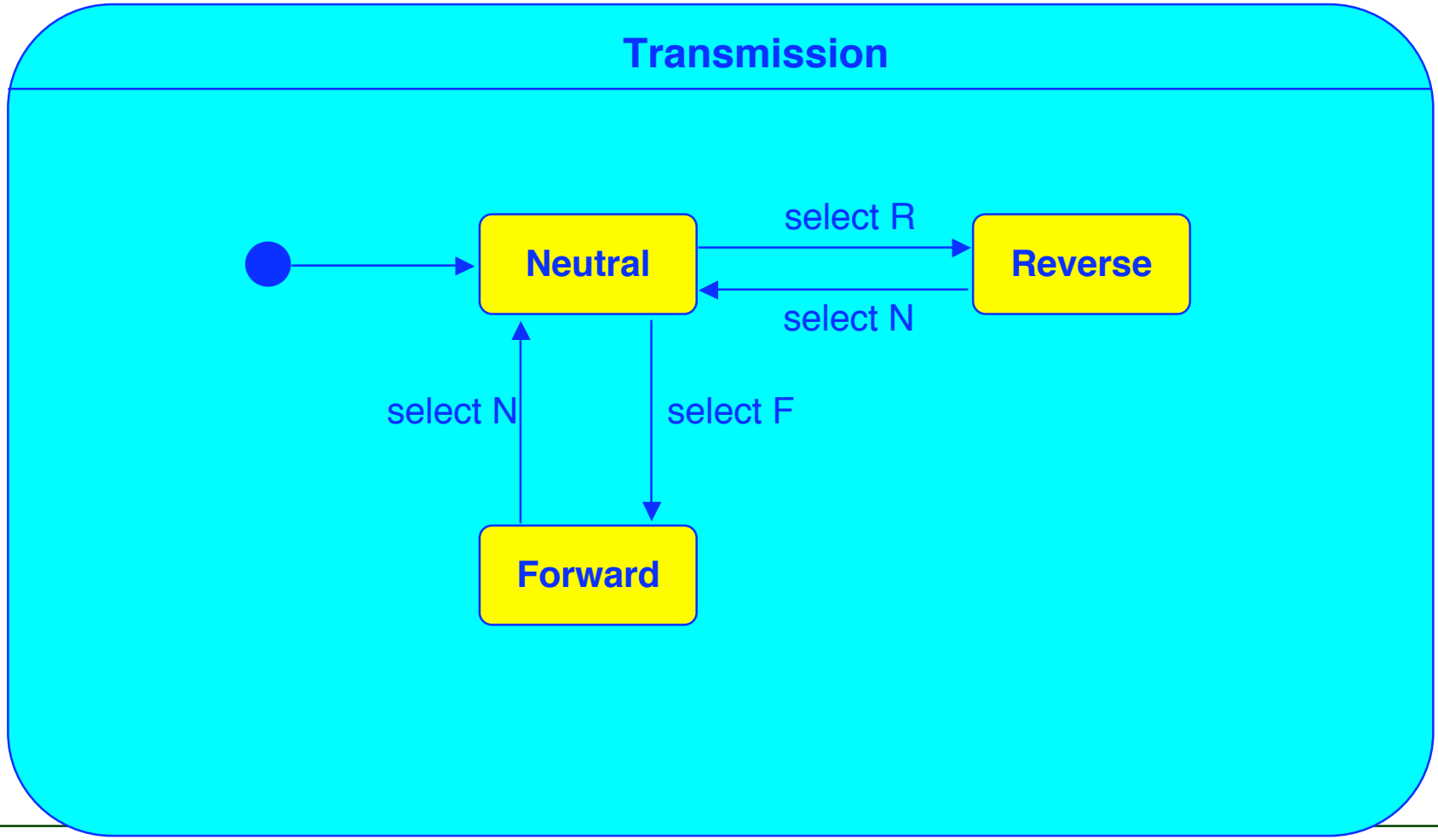
Course Lifetimes, Again



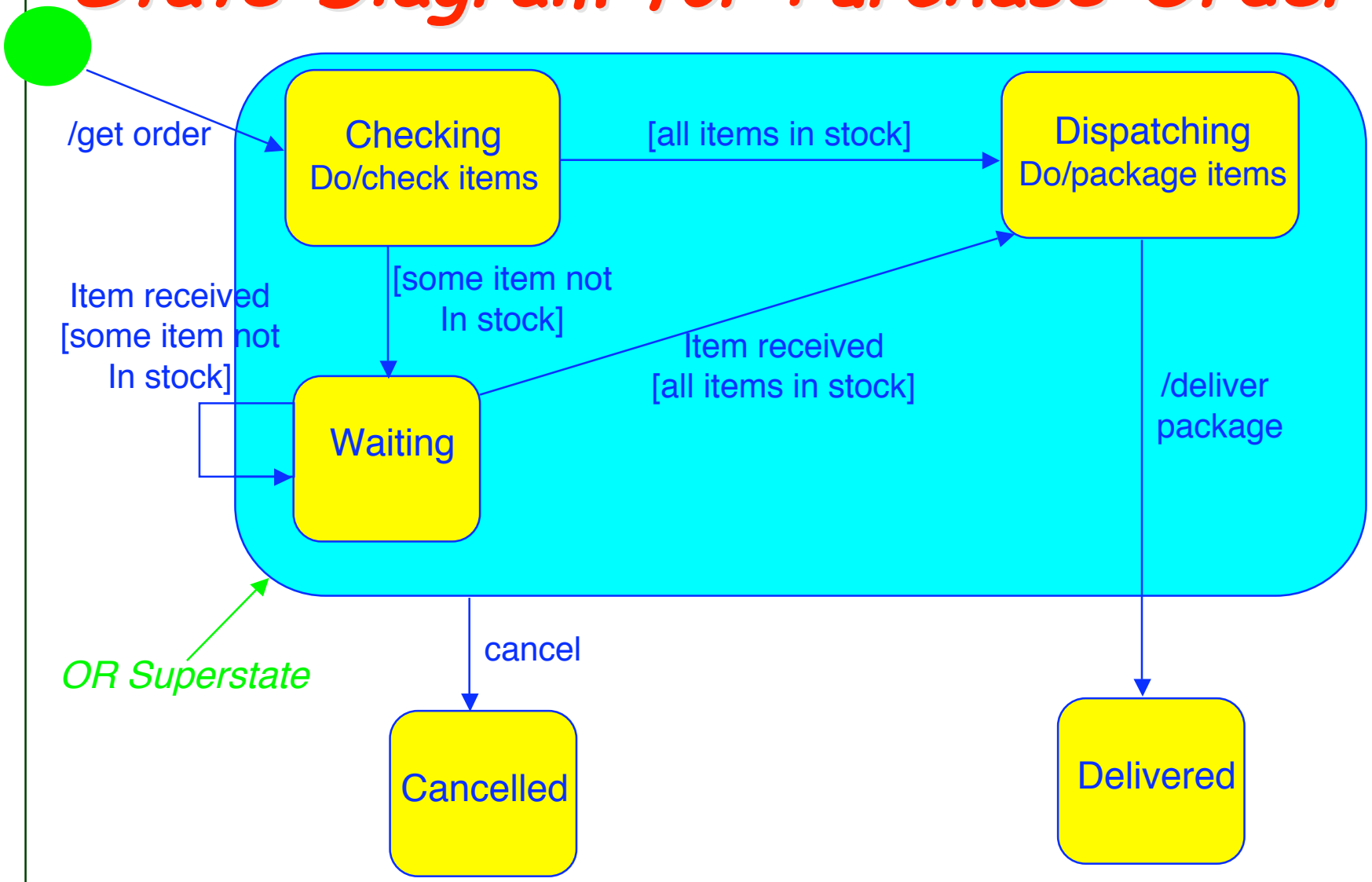
Superstates

- State transition diagrams can be very hard to read once they grow to more than a few dozen states.
- For UML activity diagrams, states can be composed into superstates. Such compositions make it possible to view an activity at different levels of abstraction.
- For example, the `Closed` state of the last activity diagram may have its own state transition diagram which describes what happens while this state is "on".
- There are two types of superstates:

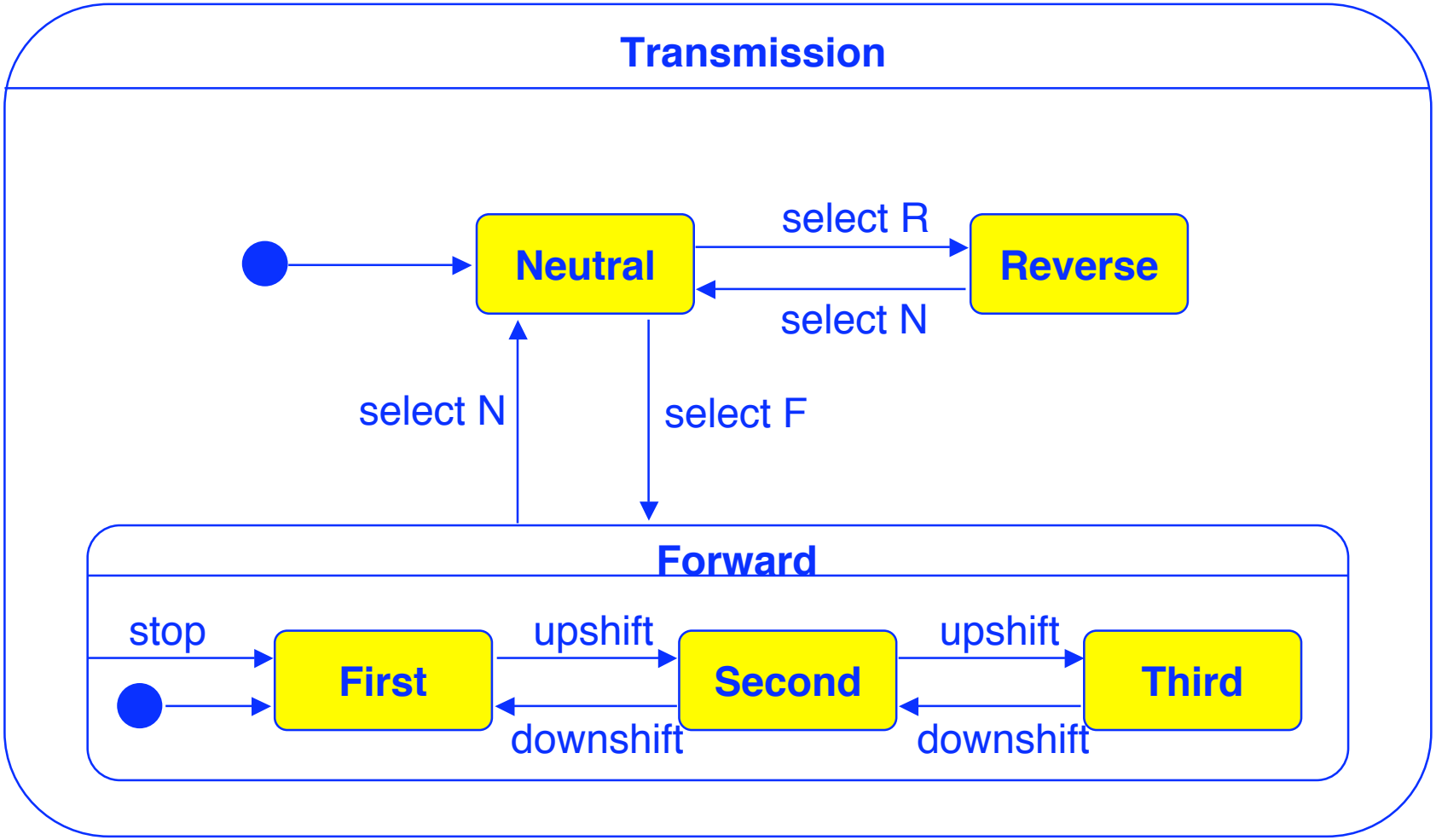
An OR Superstate



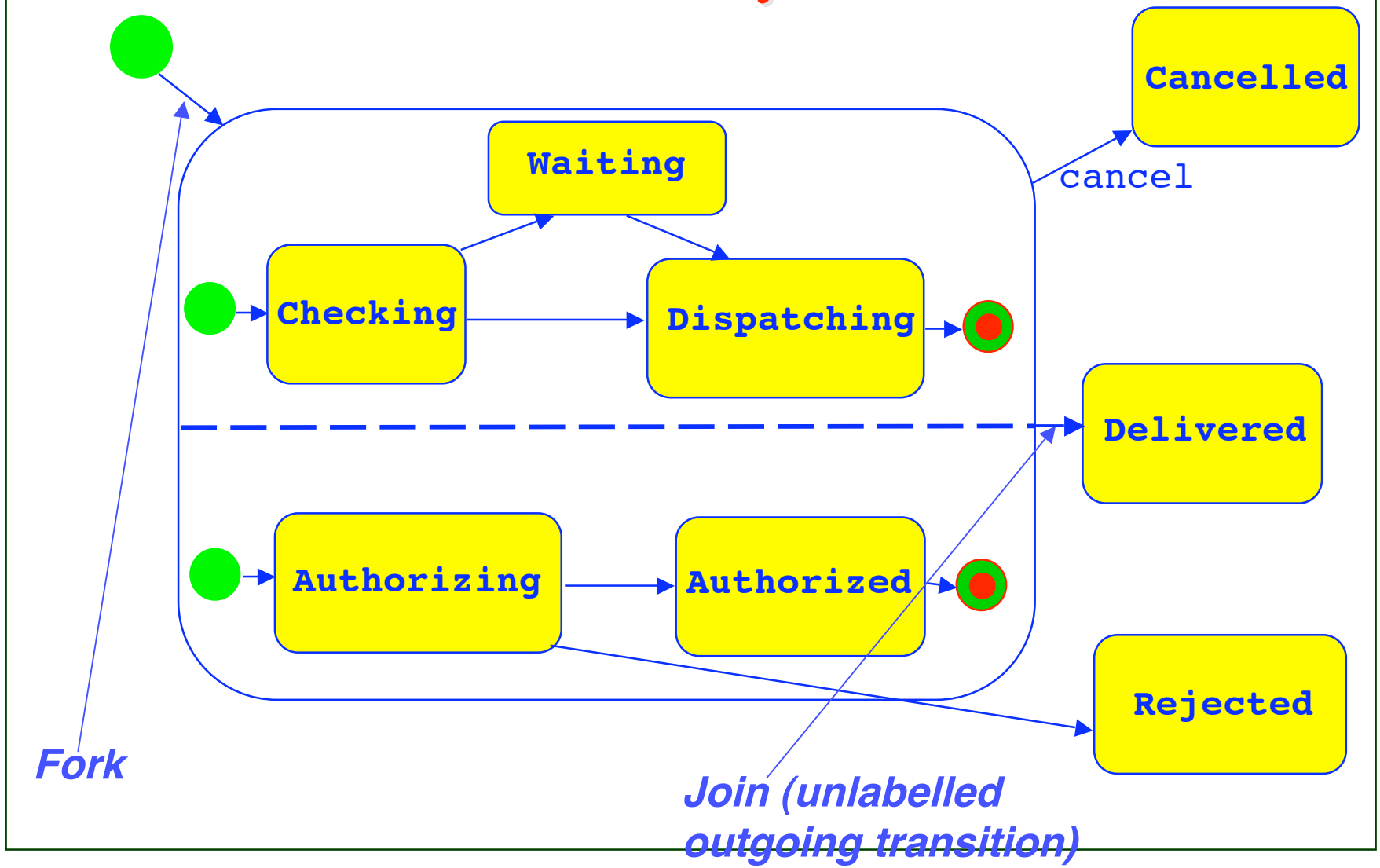
State Diagram for Purchase Order



State Diagram for (Auto) Transmission



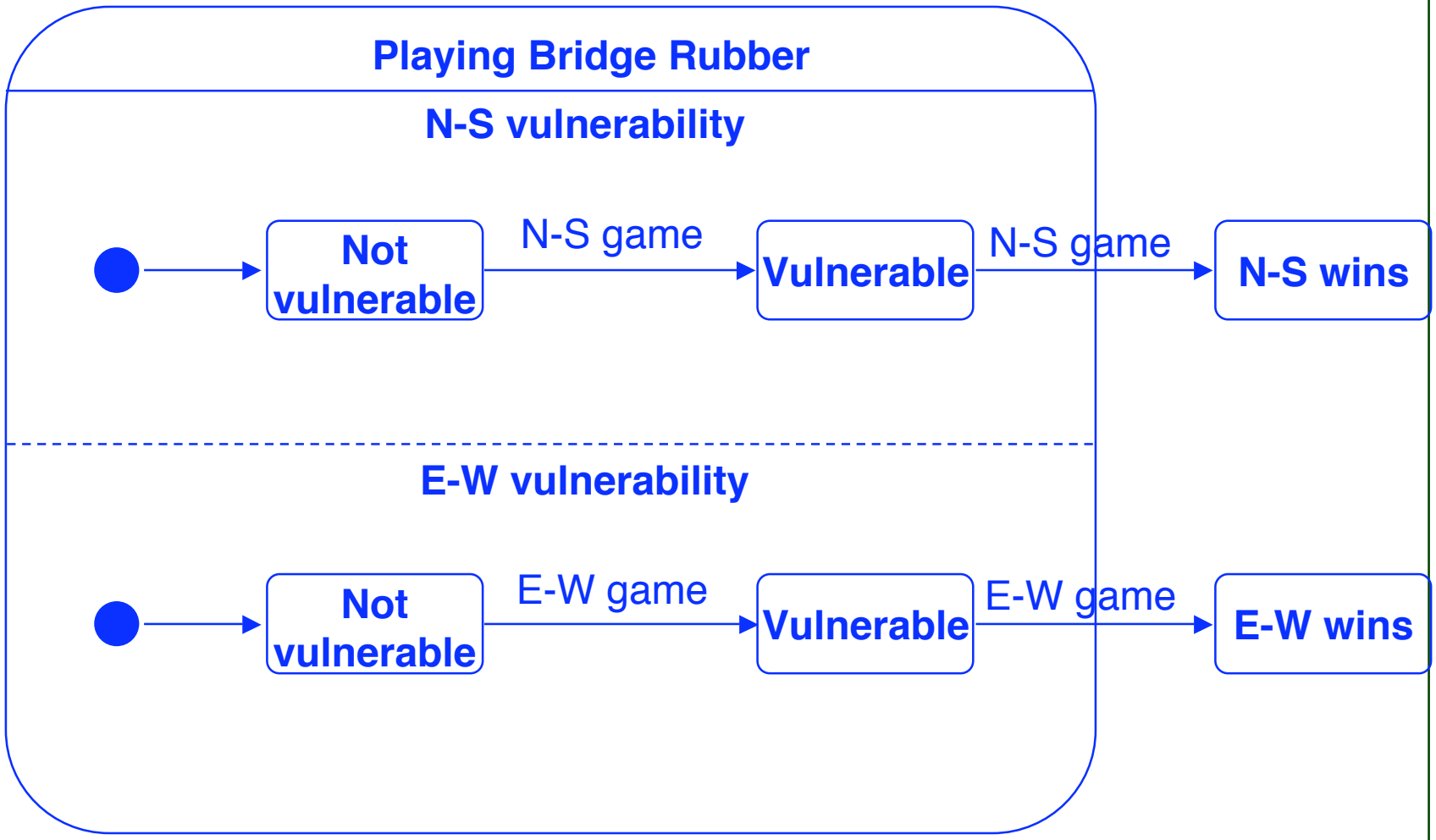
An AND Superstate



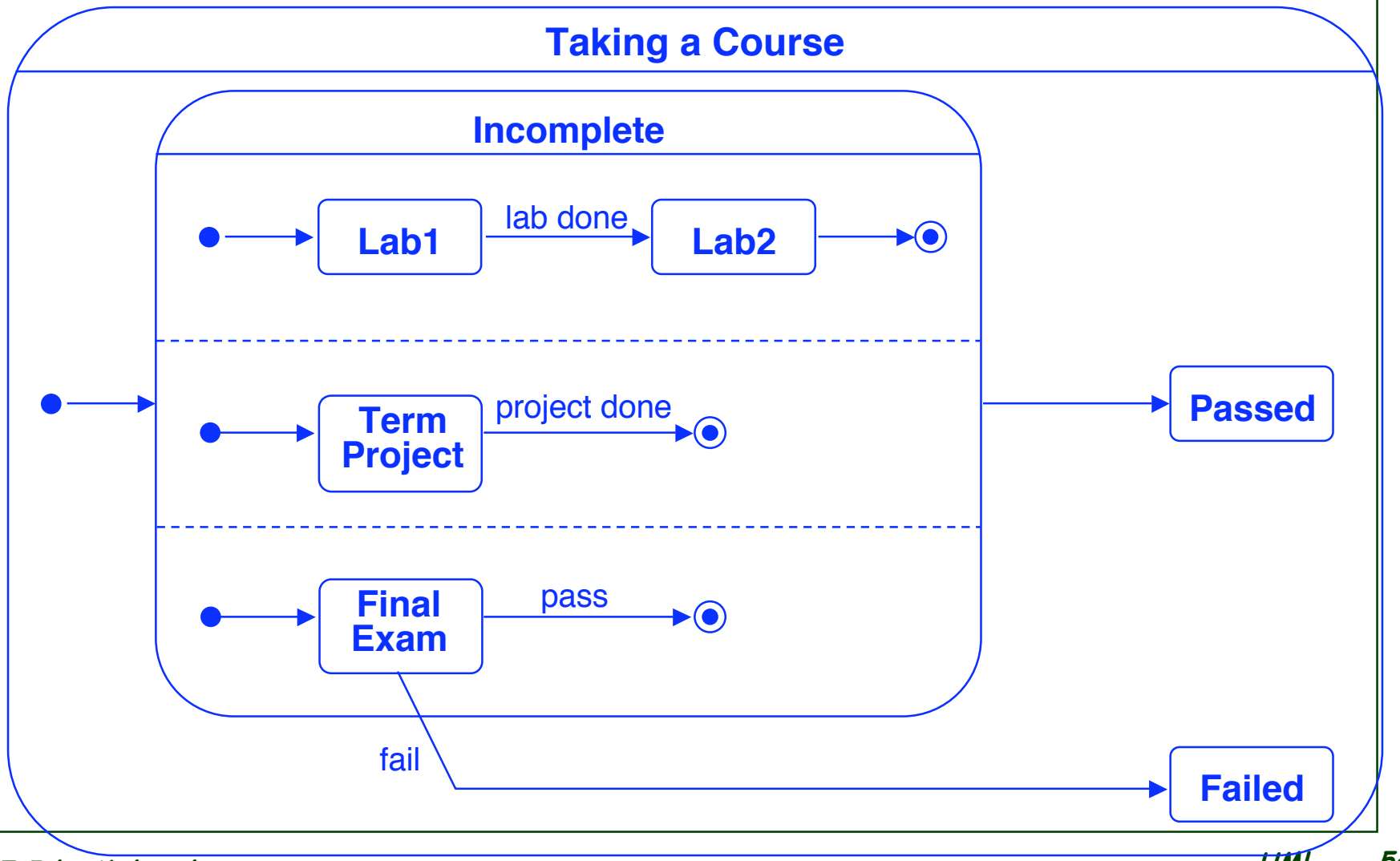
Complex State Diagram Transitions

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

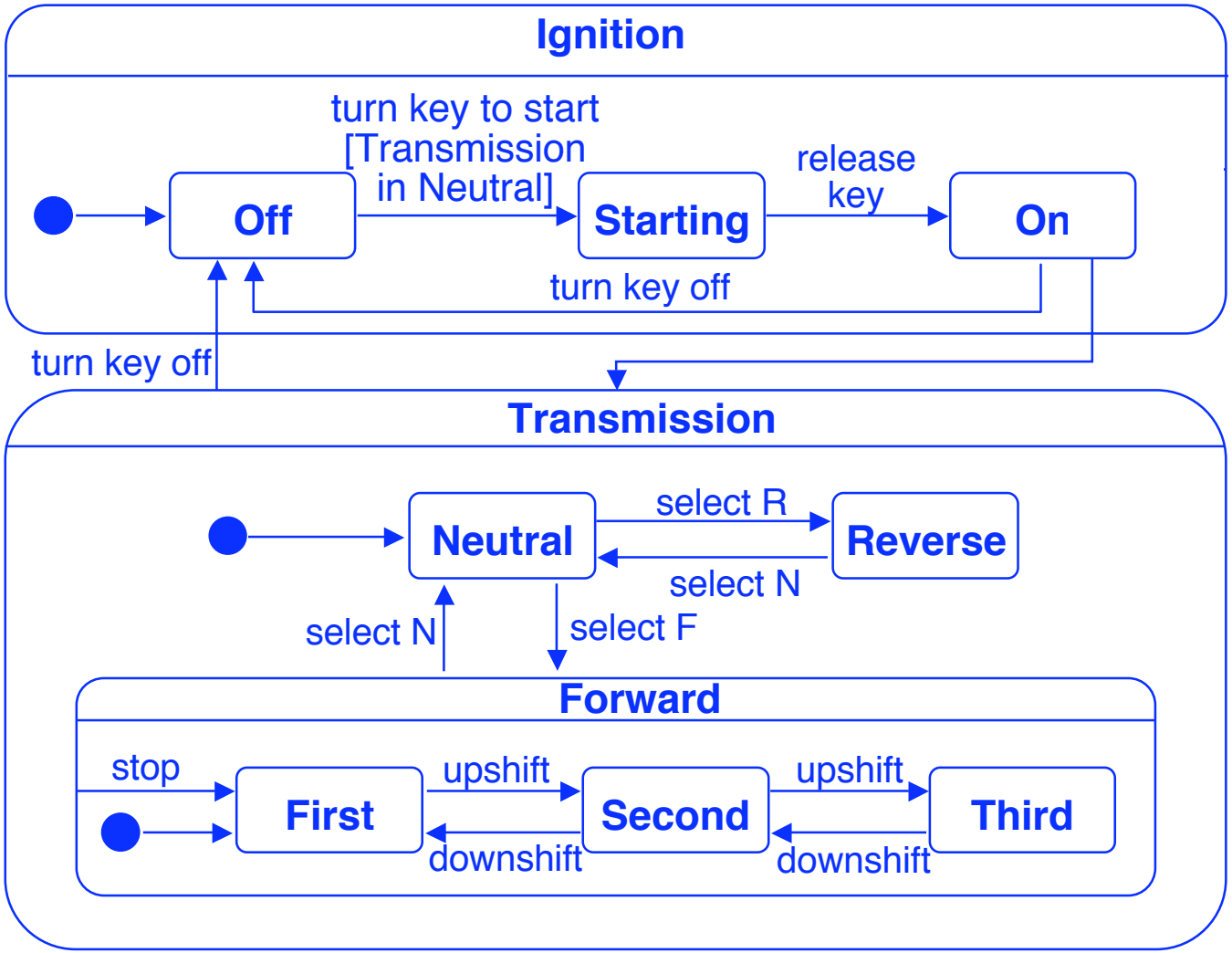
Bridge Vulnerability Rules



Taking a Course

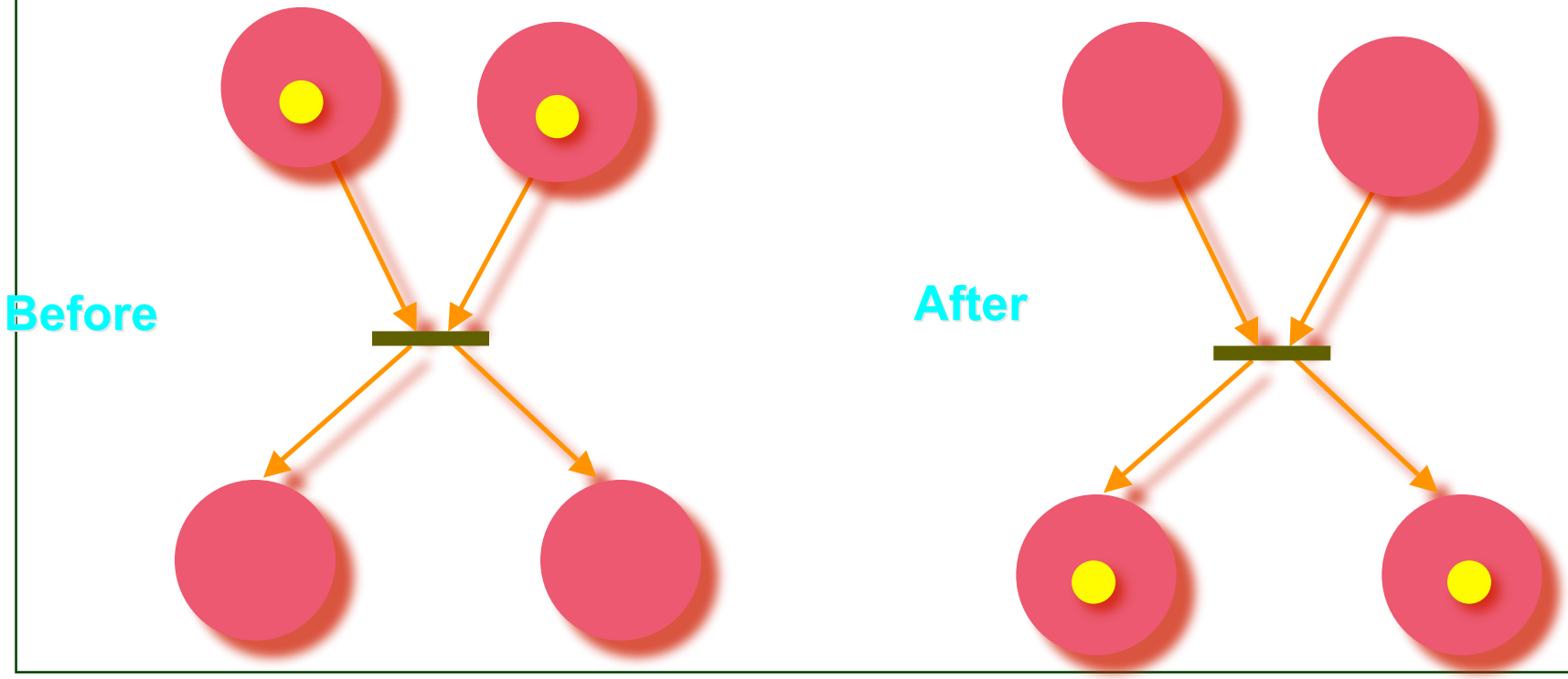


Auto Transmission

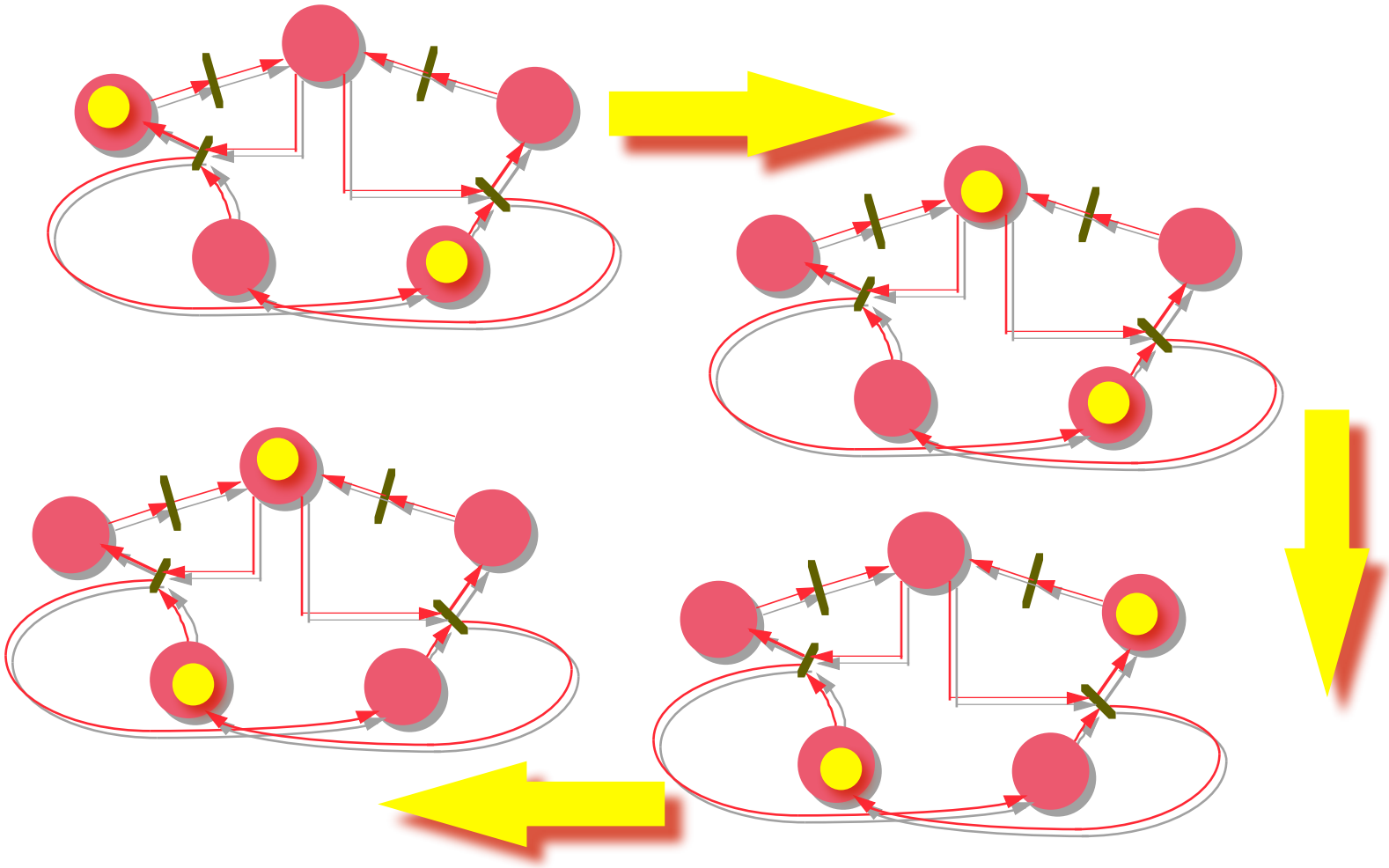


Activity Diagrams

- Like Perti nets, activity diagrams allow transitions with several input and output states:



An Example

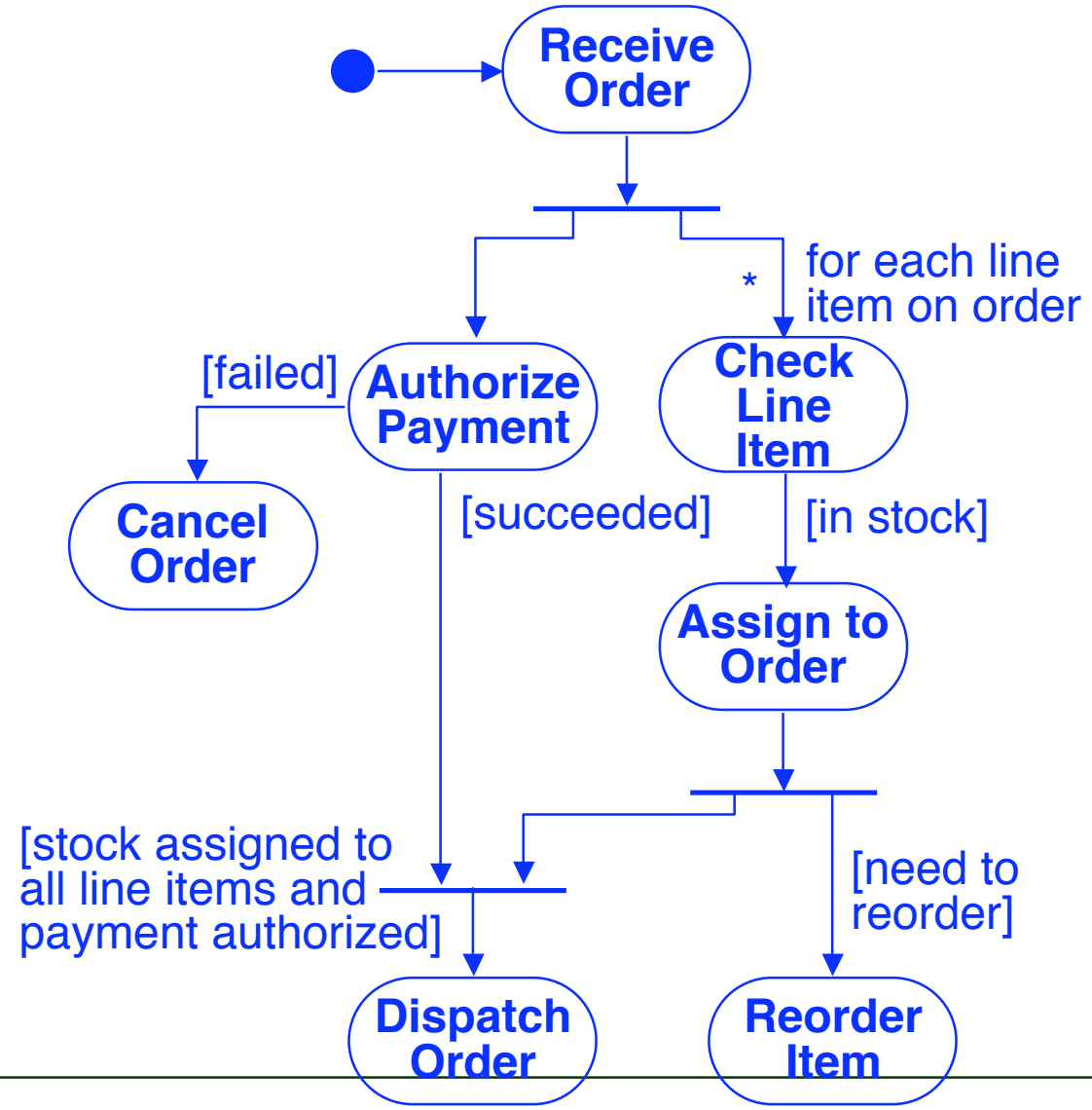


Another Example: Order Processing

- The Process Order use case:

"When we receive an order, we check each line item on the order to see if we have the goods in stock. If we do, we assign the goods to the order. If this assignment sends the quantity of those goods in stock below the reorder level, we reorder the goods. While we are doing this, we check to see if the payment is O.K. If the payment is O.K. and we have the goods in stock, we dispatch the order. If the payment is O.K. but we do not

Order Processing Activity Diagram

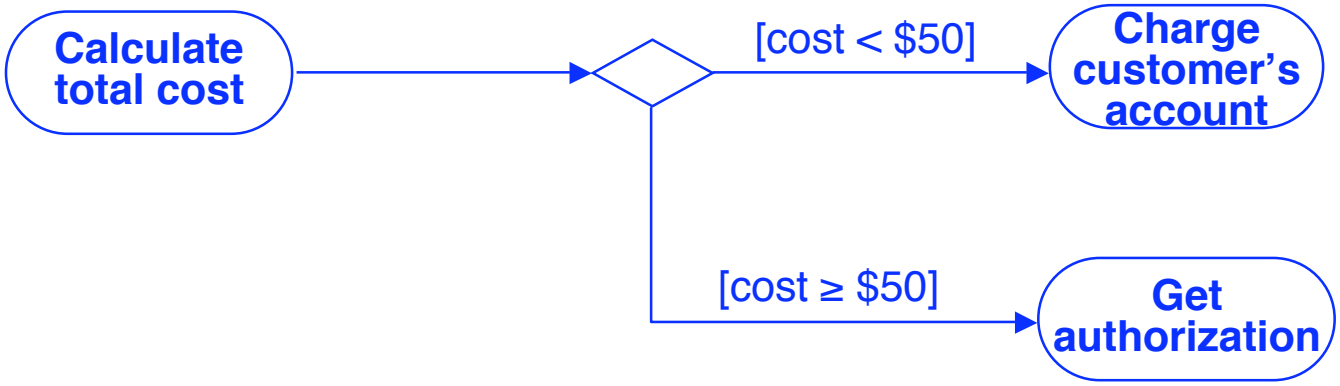


Activities

- An activity state represents an action in the execution of the activity. An activity state normally contains an action expression and usually has no associated name
- Actions may be described by:
  - ✓ Natural language
  - ✓ Structured English
  - ✓ Pseudo-code
  - ✓ Programming language
  - ✓ Another activity diagram
- An action expression may only use attributes and links of the owning object

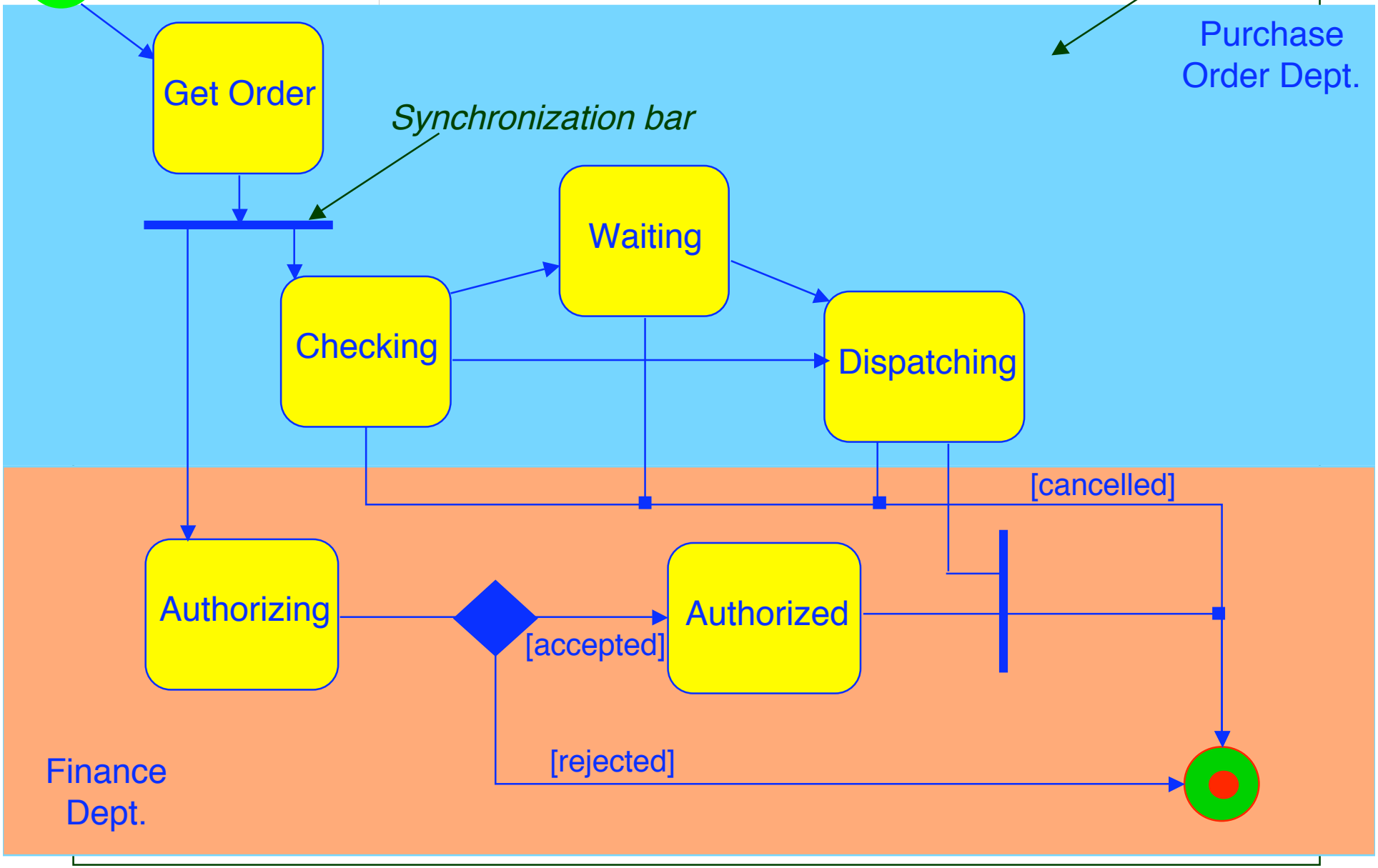
More About Activity Diagrams

- Decision points:

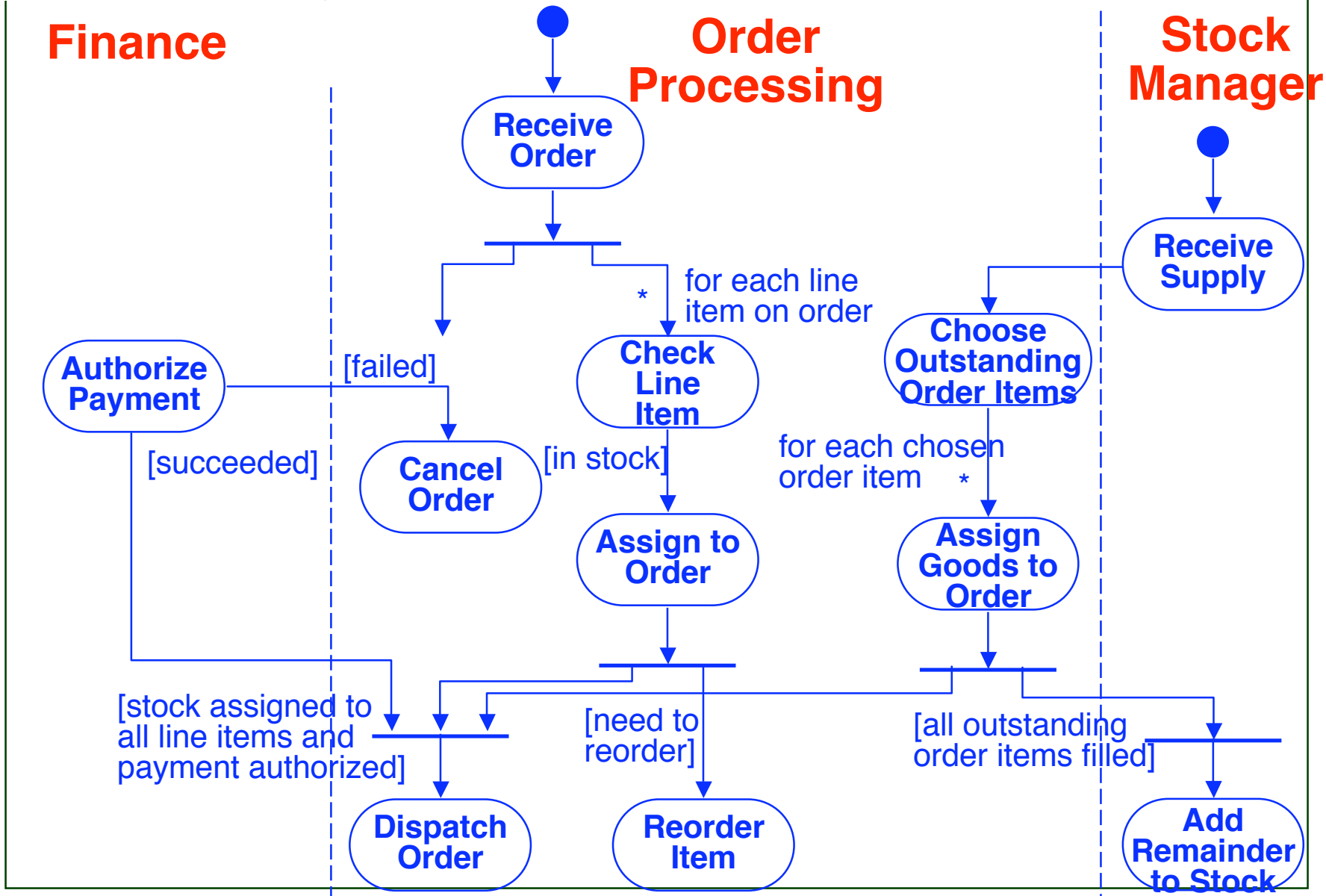


- Dead ends: there may be transitions in an activity diagram with no destination state; this can mean that:
  - ✓ Not all processing has been specified,
  - ✓ Or, that another activity diagram will take

Swimlanes



More Swimlanes



Modeling with UML

(For comparison purposes) To model with UML you need to answer the following kinds of questions:

- ✓ What are the users doing? -- use cases (Jacobson)
- ✓ What are the objects in the real world? (Rumbaugh)
- ✓ What objects are needed for each use case? (Jacobson)
- ✓ How do objects collaborate? (Jacobson, Booch)
- ✓ What are allowable sequences of actions and activities?
- ✓ What are allowable lifelines for objects?

Conclusions

- UML amounts to a combination of EER diagrams, statecharts and other diagrammatic notations.
- Much of this is useful for conceptual modeling. Some diagrams, however, are appropriate for *software* modeling only.
- The great contribution of UML is that for the first time ever, there is a modeling standard; this has led to compatibility and portability of conceptual models.
- The great weakness of UML is that it was designed by committee, and some design decisions were based on political, rather than technical

References

- [Booch97] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley, 1997.
- [Fowler97] Fowler, M., Scott, K., *UML Distilled*, Addison-Wesley, 1997.
- [Gogolla98] Gogolla, M., "UML for the Impatient", technical report 3-98, Fachberiech Mathematik und Informatik, Universitaet Bremen, 1998.
- [Harel87] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming 8*, 1987.
- [Jacobson92] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
- [Lochovsky98] Lochovsky, F., *Lecture Notes on Software Engineering*, Department of Computer Science, University of Washington, 1998, http://www.cs.washington.edu/education/courses/403/
- [Motschnig93] Motschnig-Pitrik, R., "The Semantics of Parts versus Aggregates in Data/Knowledge Modeling", Proceedings Fifth Conference on Advanced Information Systems Engineering (CAISE'93). Paris. June 1993.