Object-Oriented Software Engineering
Practical Software Development using UML and Java

Chapter 8:
Modelling Interactions and Behaviour

Lecture 11

8.1 Interaction Diagrams

Interaction diagrams are used to model the dynamic aspects of a software system

- They help you to visualize how the system runs.
- An interaction diagram is often built from a use case and a class diagram.
  - The objective is to show how a set of objects accomplish the required interactions with an actor.

Interactions and messages

- Interaction diagrams show how a set of actors and objects communicate with each other to perform:
  - The steps of a use case, or
  - The steps of some other piece of functionality.

- The set of steps, taken together, is called an *interaction*.

- Interaction diagrams can show several different types of communication.
  - E.g. method calls, messages send over the network
  - These are all referred to as *messages*.

Elements found in interaction diagrams

- Instances of classes
  - Shown as boxes with the class and object identifier underlined

- Actors
  - Use the stick-person symbol as in use case diagrams

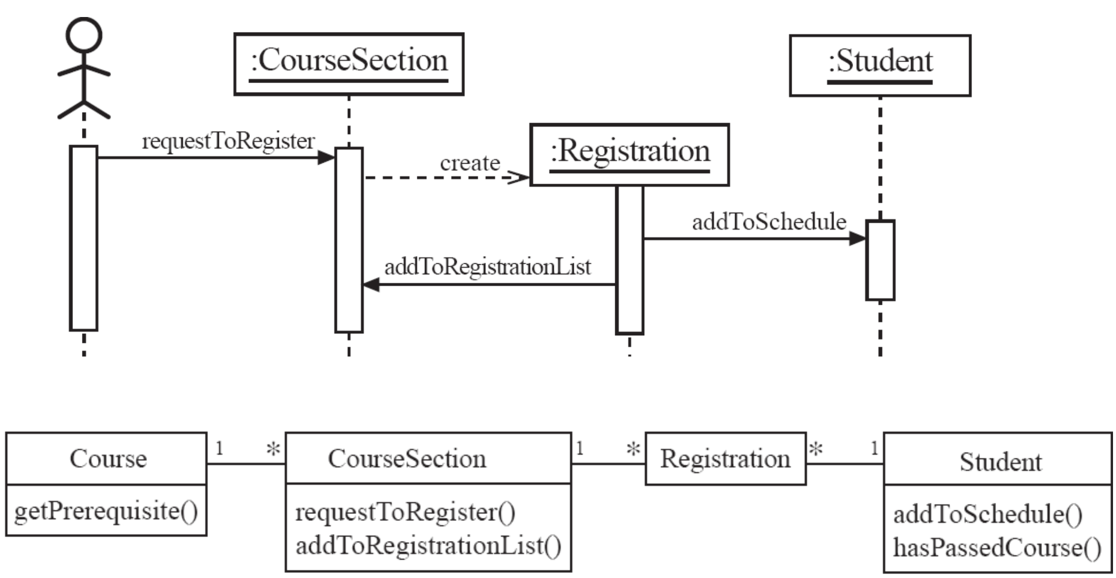
- Messages
  - Shown as arrows from actor to object, or from object to object

Creating interaction diagrams

You should develop a class diagram and a use case model before starting to create an interaction diagram.

- There are two kinds of interaction diagrams:
  - —*Sequence diagrams*
  - —*Communication diagrams*

Sequence diagrams – an example

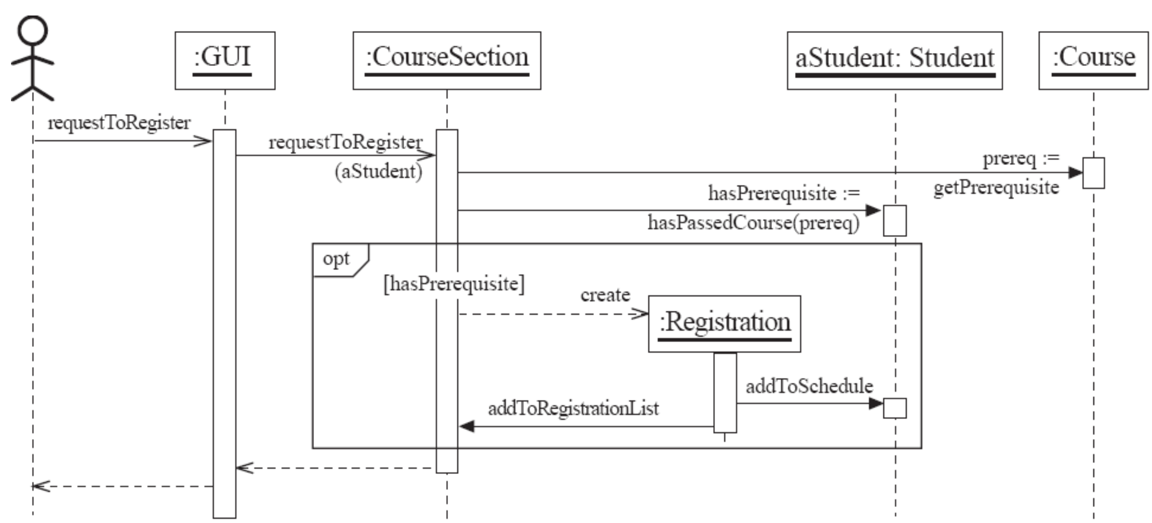


Sequence diagrams

A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task

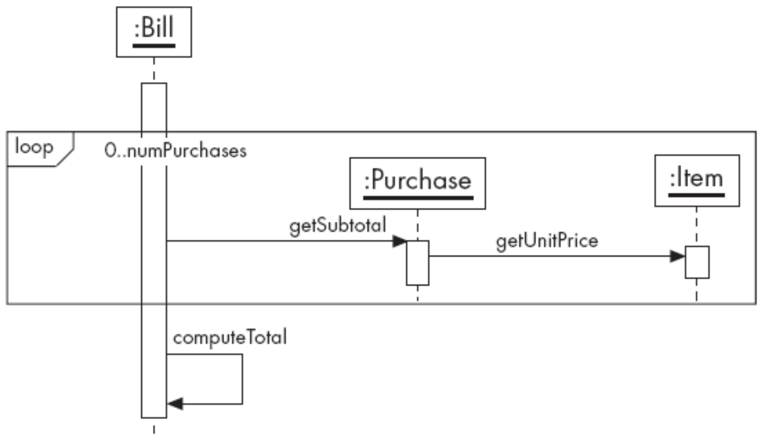
- The objects are arranged horizontally across the diagram.
- An actor that initiates the interaction is often shown on the left.
- The vertical dimension represents time.
- A vertical line, called a *lifeline*, is attached to each object or actor.
- The lifeline becomes a broad box, called an *activation box* during the *live activation* period.
- A message is represented as an arrow between activation boxes of the sender and receiver.
  - A message is labelled and can have an argument list and a return value.

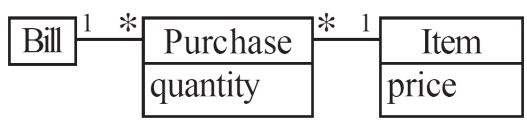
Sequence diagrams – same example, more details



Sequence diagrams – an example with replicated messages

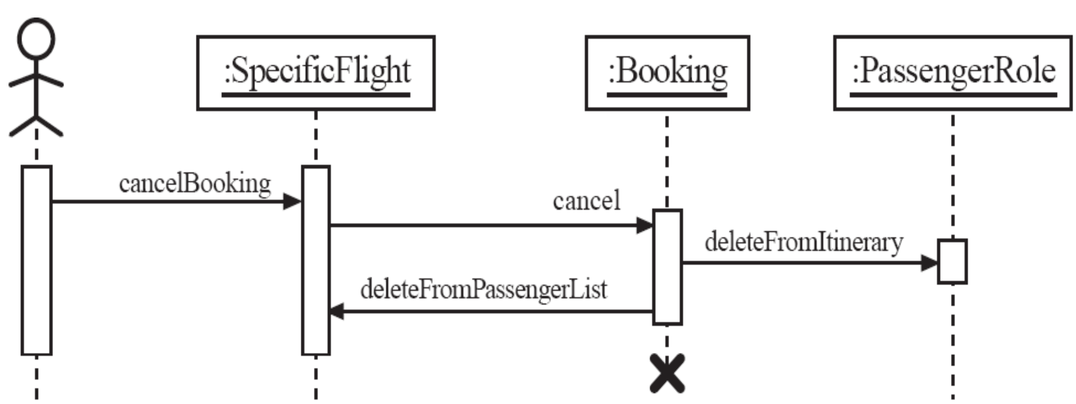
- An *iteration* over objects is indicated by an asterisk preceding the message name





Sequence diagrams – an example with object deletion

- If an object's life ends, this is shown with an X at the end of the lifeline



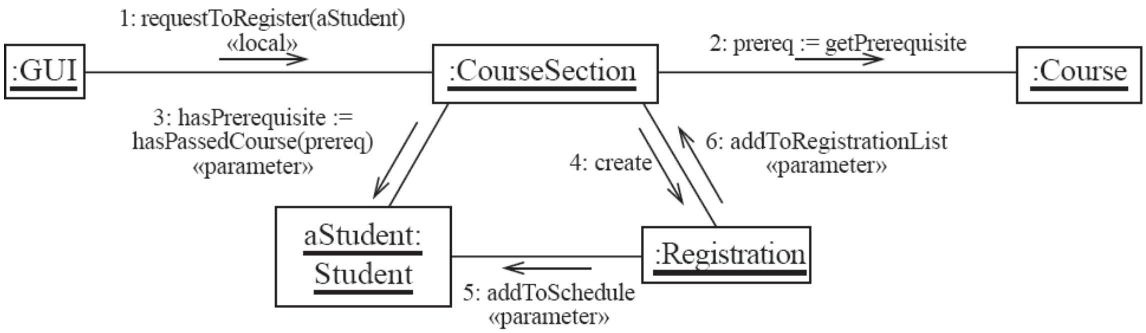
Communication diagrams – an example

Communication diagrams

Communication diagrams emphasize how the objects collaborate in order to realize an interaction

- A communication diagram is a graph with the objects as the vertices.
- Communication links are added between objects
- Messages are attached to these links.
  - Shown as arrows labelled with the message name
- Time ordering is indicated by prefixing the message with some numbering scheme.

Communication diagrams – same example, more details



Communication links

- A communication link can exist between two objects whenever it is possible for one object to send a message to the other one.

- Several situations can make this message exchange possible:

  1. The classes of the two objects have an *association* between them.
     - This is the most common case.
     - If all messages are sent in the same direction, then probably the association can be made unidirectional.

Other communication links

2. The receiving object is stored in a *local* variable of the sending method.
   - This often happens when the object is created in the sending method or when some computation returns an object .
   - The stereotype to be used is «local» or [L].

3. A reference to the receiving object has been received as a *parameter* of the sending method.
   - The stereotype is «parameter» or [P].

Other communication links

4. The receiving object is global.
   - This is the case when a reference to an object can be obtained using a static method.
   - The stereotype «global», or a [G] symbol is used in this case.

5. The objects communicate over a network.
   - We suggest to write «network».

How to choose between using a sequence or communication diagram

Sequence diagrams

- Make explicit the time ordering of the interaction.
  - —Use cases make time ordering explicit too
  - —So sequence diagrams are a natural choice when you build an interaction model from a use case.

- Make it easy to add details to messages.
  - —Communication diagrams have less space for this

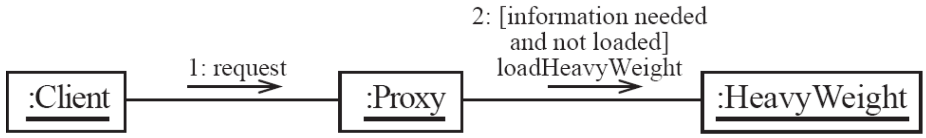
How to choose between using a sequence or communication diagram

Communication diagrams

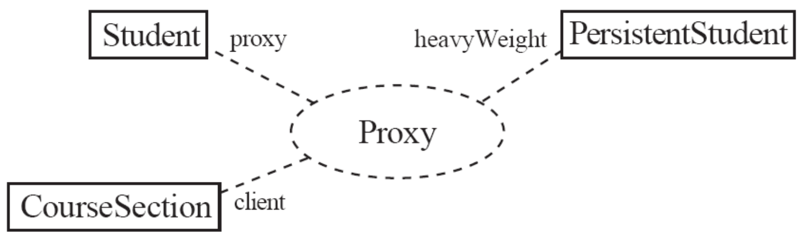
- Can be seen as a projection of the class diagram
  - —Might be preferred when you are *deriving* an interaction diagram from a class diagram.
  - —Are also useful for *validating* class diagrams.

Communication diagrams and patterns

A communication diagram can be used to represent aspects of a *design pattern*



(a)



(b)

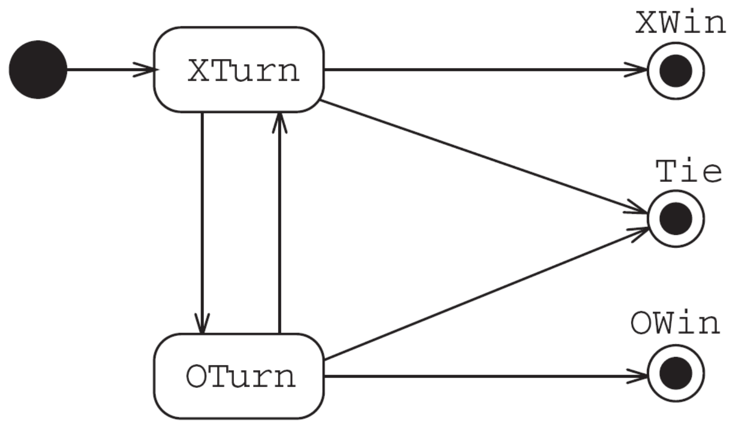
8.2 State Diagrams

A state diagram describes the behaviour of a *system*, some *part* of a system, or an *individual object*.

- At any given point in time, the system or object is in a certain state.
  - Being in a state means that it will behave in a *specific way* in response to any events that occur.
- Some events will cause the system to change state.
  - In the new state, the system will behave in a different way to events.
- A state diagram is a directed graph where the nodes are states and the arcs are transitions.

State diagrams – an example

- tic-tac-toe game (also called noughts and crosses)



States

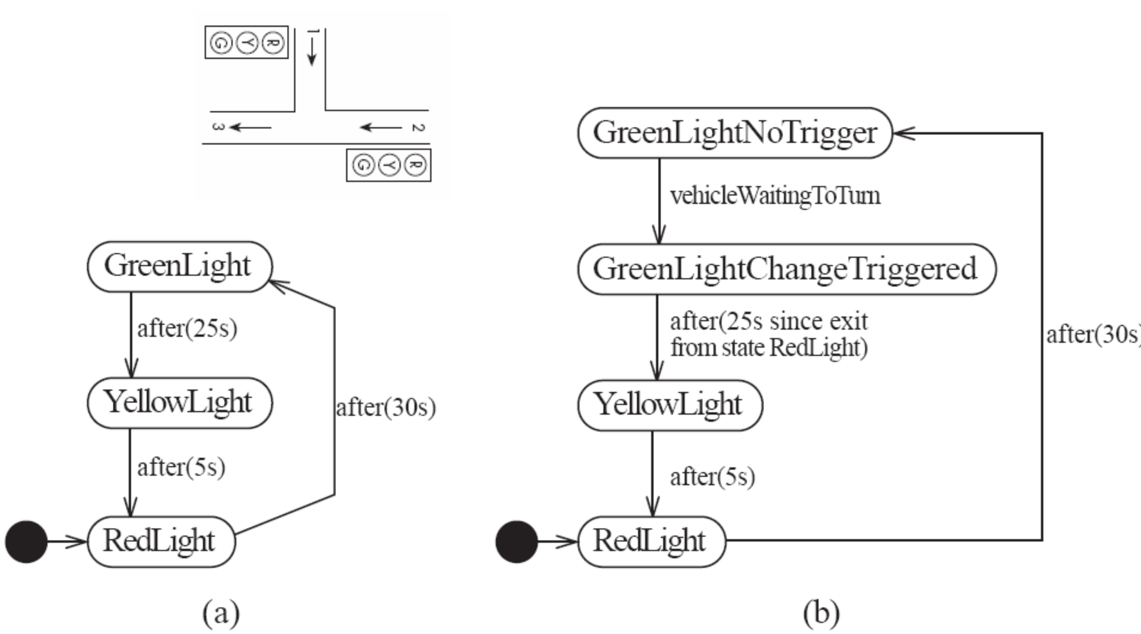
- At any given point in time, the system is in one state.
- It will remain in this state until an event occurs that causes it to change state.
- A state is represented by a rounded rectangle containing the name of the state.
- Special states:
  - — A black circle represents the *start state*
  - — A circle with a ring around it represents an *end state*

Transitions

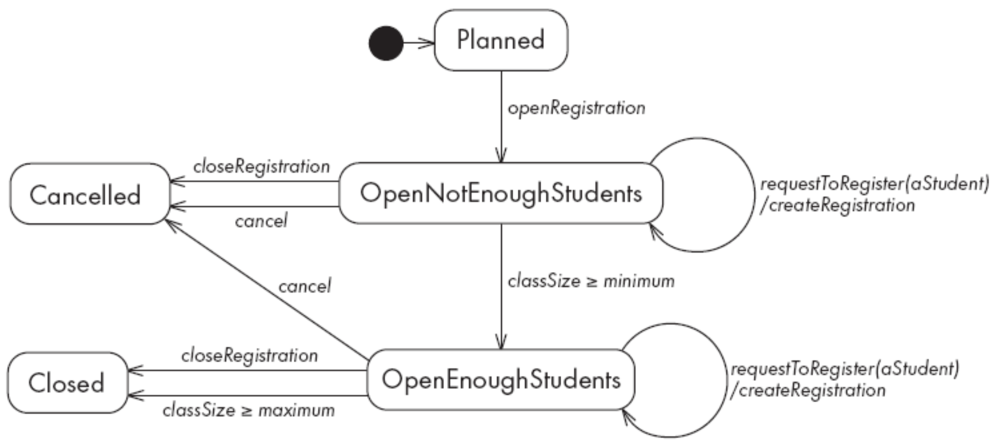
- A transition represents a change of state in response to an event.
  - It is considered to occur instantaneously.

- The label on each transition is the event that causes the change of state.

State diagrams – an example of transitions with time-outs and conditions



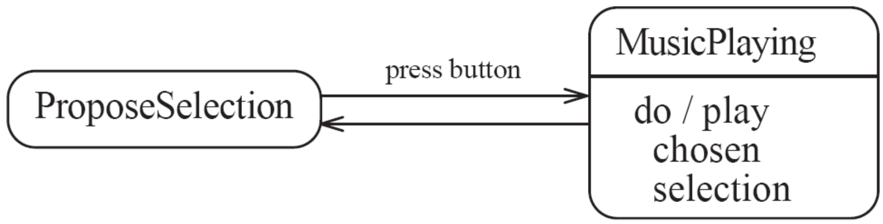
State diagrams – an example with conditional transitions



Activities in state diagrams

- An *activity* is something that takes place while the system is *in* a state.
  - It takes a period of time.
  - The system may take a transition out of the state in response to completion of the activity,
  - Some other outgoing transition may result in:
    - The interruption of the activity, and
    - An early exit from the state.

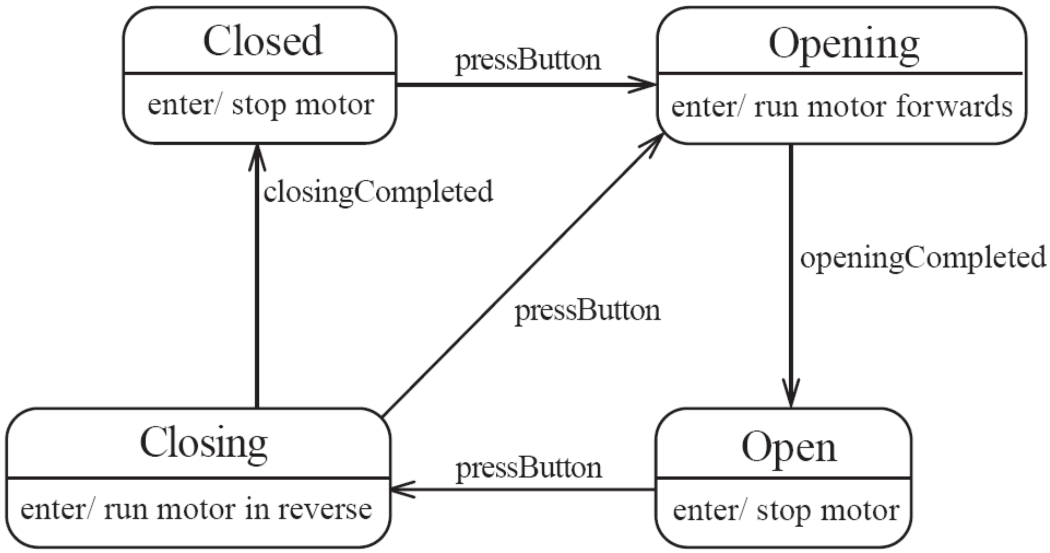
State diagram – an example with activity



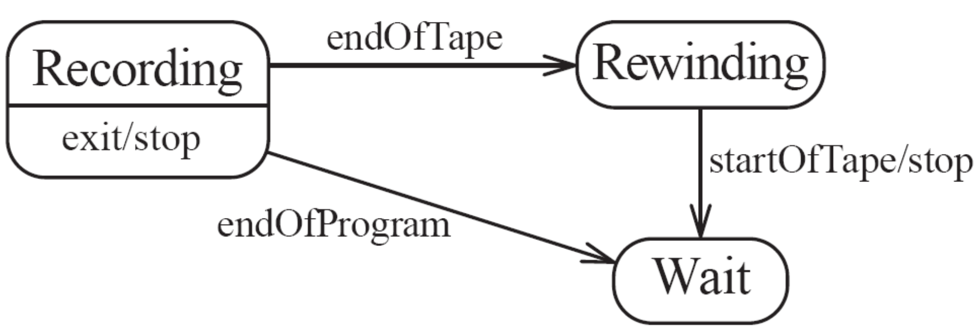
Actions in state diagrams

- An *action* is something that takes place effectively *instantaneously*
  - When a particular transition is taken,
  - Upon entry into a particular state, or
  - Upon exit from a particular state
- An action should consume no noticeable amount of time

State diagram – an example with actions



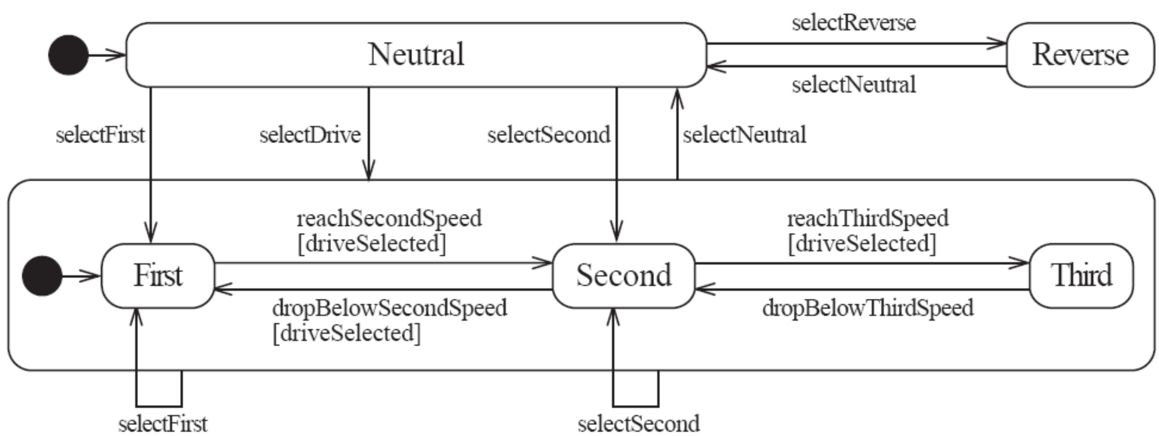
State diagrams – another example



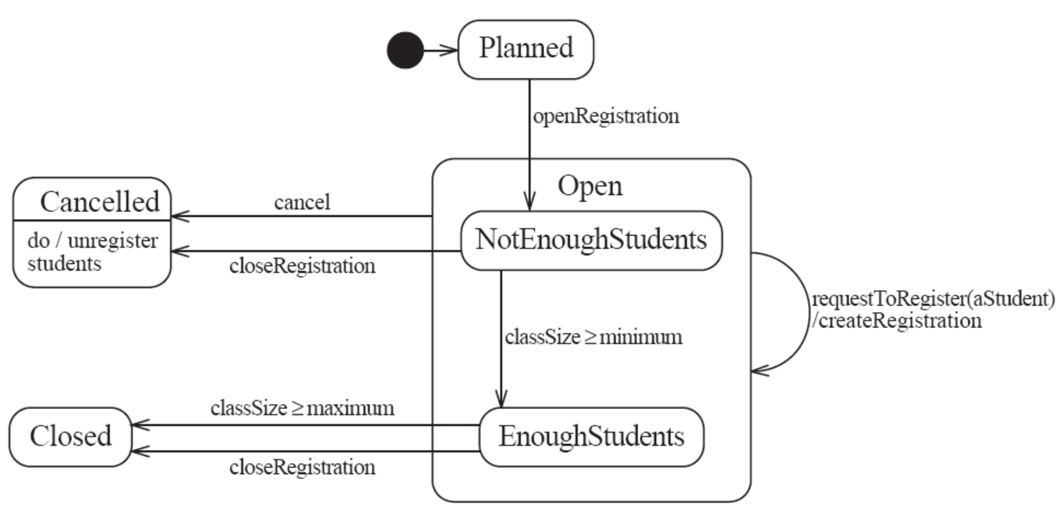
Nested substates and guard conditions

A state diagram can be nested inside a state.

- The states of the inner diagram are called *substates*.



State diagram – an example with substates



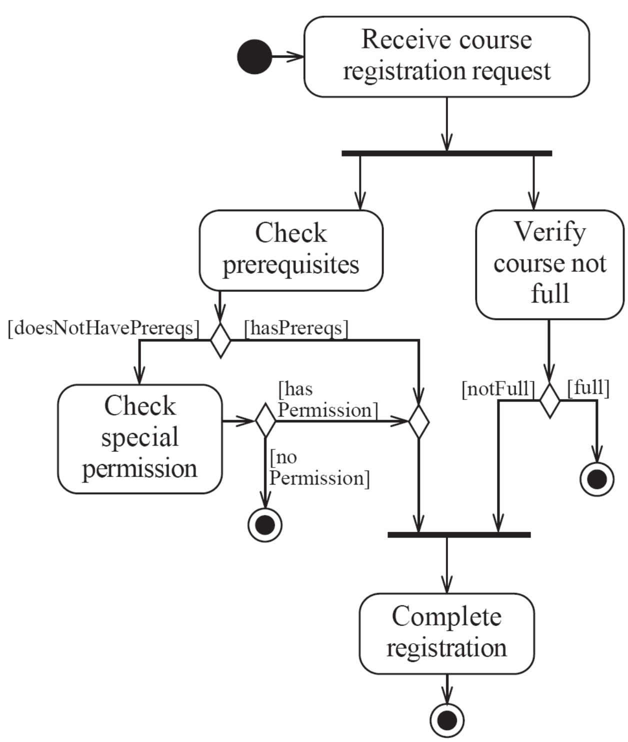
8.3 Activity Diagrams

- An *activity diagram* is like a state diagram.
  - —Except most transitions are caused by *internal* events, such as the completion of a computation.

- An activity diagram
  - —Can be used to understand the flow of work that an object or component performs.
  - —Can also be used to visualize the interrelation and interaction between different use cases.
  - —Is most often associated with several classes.

- One of the strengths of activity diagrams is the representation of *concurrent* activities.

Activity diagrams – an example



Representing concurrency

- Concurrency is shown using forks, joins and rendezvous.
  - A *fork* has one incoming transition and multiple outgoing transitions.
    - The execution splits into two concurrent threads.
  - A *rendezvous* has multiple incoming and multiple outgoing transitions.
    - Once all the incoming transitions occur all the outgoing transitions may occur.

Representing concurrency

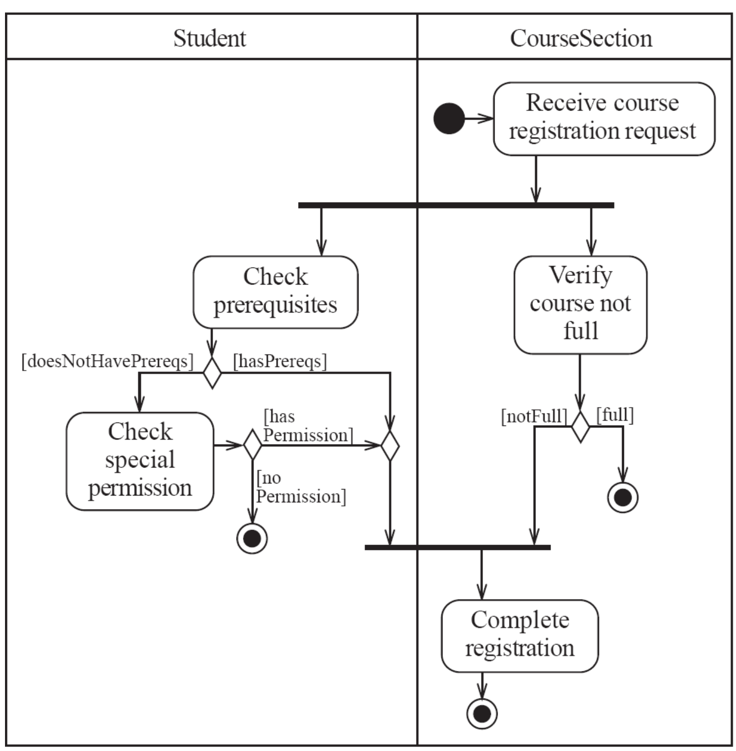
- A *join* has multiple incoming transitions and one outgoing transition.
  - The outgoing transition will be taken when all incoming transitions have occurred.
  - The incoming transitions must be triggered in separate threads.
  - If one incoming transition occurs, a wait condition occurs at the join until the other transitions occur.

Swimlanes

Activity diagrams are most often associated with several classes.

- The partition of activities among the existing classes can be explicitly shown using *swimlanes*.

Activity diagrams – an example with swimlanes



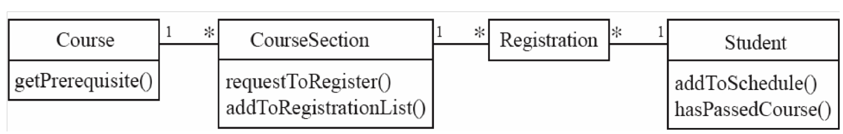
8.4 Implementing Classes Based on Interaction and State Diagrams

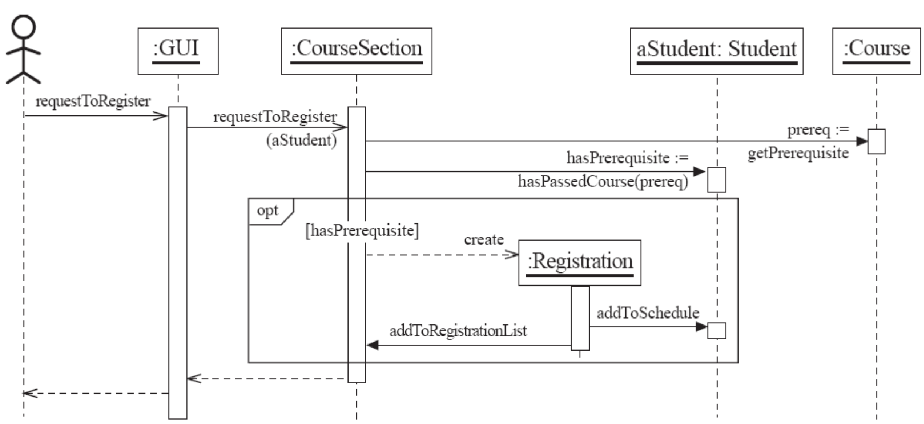
- You should use these diagrams for the parts of your system that you find most complex.
  - —I.e. not for every class

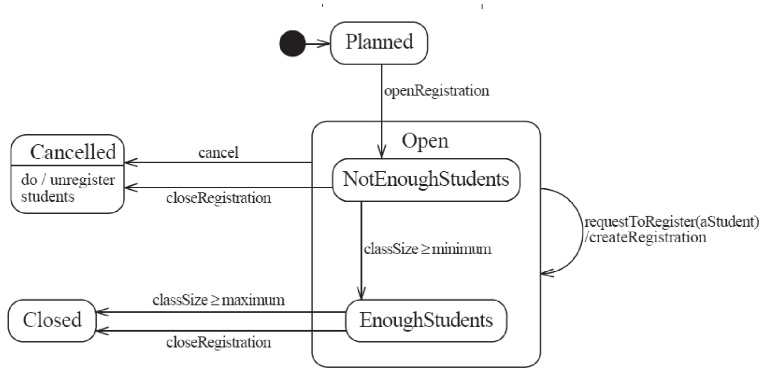
- Interaction, activity and state diagrams help you create a correct implementation.

- This is particularly true when behaviour is *distributed* across several use cases.
  - —E.g. a state diagram is useful when different conditions cause instances to respond differently to the same event.

Example







Example: The CourseSection class

States:

- 'Planned':
  closedOrCancelled == false && open == false

- 'Cancelled':
  closedOrCancelled == true &&
  registrationList.size() == 0

- 'Closed' (course section is too full, or being taught):
  closedOrCancelled == true &&
  registrationList.size() > 0

Example: The CourseSection class

States:

- 'Open' (accepting registrations):
  open == true

- 'NotEnoughStudents' (substate of 'Open'):
  open == true &&
  registrationList.size() < course.getMinimum()

- 'EnoughStudents' (substate of 'Open'):
  open == true &&
  registrationList.size() >= course.getMinimum()

Example: The CourseSection class

```
public class CourseSection
{
   // The many-1 abstraction-occurrence association (Figure 8.2)
   private Course course;

   // The 1-many association to class Registration (Figure 8.2)
   private List<Registration> registrationList;

   // The following are present only to determine the state
   // (as in Figure 8.19). The initial state is 'Planned‘
   private boolean open = false;
   private boolean closedOrCanceled = false;
```


Example: The CourseSection class

```
public CourseSection(Course course)
{
     this.course = course;
     registrationList = new LinkedList<Registration>();
}


public void openRegistration()
{
    if(!closedOrCanceled) // must be in 'Planned' state
   {
        open = true; // to 'OpenNotEnoughStudents' state
    }
}
```

Example: The CourseSection class

```
public void closeRegistration()
{
    // to 'Canceled' or 'Closed' state
    open = false;
    closedOrCanceled = true;
    if (registrationList.size() < course.getMinimum())
    {
        unregisterStudents(); // to 'Canceled' state
    }
}

public void cancel()
{
    // to 'Canceled' state
    open = false;
    closedOrCanceled = true;
    unregisterStudents();
}
```


Example: The CourseSection class

```
public void requestToRegister(Student student)
{
   if (open) // must be in one of the two 'Open' states
   {
      // The interaction specified in the sequence diagram of Figure 8.4
      Course prereq = course.getPrerequisite();
      if (student.hasPassedCourse(prereq))
      {
         // Indirectly calls addToRegistrationList
         new Registration(this, student);
      }
      // Check for automatic transition to 'Closed' state
      if (registrationList.size() >= course.getMaximum())
      {
         // to 'Closed' state
         open = false;
         closedOrCanceled = true;
      }
   }
}
```

Example: The CourseSection class

```
 // Private method to remove all registrations
 // Activity associated with 'Canceled' state.
 private void unregisterStudents()
{
  for(Registration next : registrationList)
  {
     next.unregisterStudent();
     registrationList.remove(next);
   }
}
// Called within this package only, by the constructor of
// Registration to ensure the link is bi-directional
void addToRegistrationList(Registration newRegistration)
{
   registrationList.add(newRegistration);
 }
}
```


8.5 Difficulties and Risks in Modelling Interactions and Behaviour

Dynamic modelling is a difficult skill

- In a large system there are a very large number of possible paths a system can take.
- It is hard to choose the classes to which to allocate each behaviour:
  - *Ensure that skilled developers lead the process, and ensure that all aspects of your models are properly reviewed.*
  - *Work iteratively:*
    - *Develop initial class diagrams, use cases, responsibilities, interaction diagrams and state diagrams;*
    - *Then go back and verify that all of these are consistent, modifying them as necessary.*
  - *Drawing different diagrams that capture related, but distinct, information will often highlight problems.*