Interaction Diagrams

Interaction diagrams are used to model the dynamic aspects of a software system. They help in visualizing how the system runs.

- An interaction diagram is often built from a use case and a class diagram with the objective of showing how a set of objects accomplish the required interactions with an actor.
- Interaction diagrams show how a set of actors and objects communicate with each other to perform a set of steps, which taken together are called an *interaction*:
  - The steps of a use case, or
  - The steps of some other piece of functionality.
- Interaction diagrams show several different types of communication referred to as messages. E.g. method calls, messages send over the network.
- There are two kinds of interaction diagrams:
  - Sequence diagrams
  - Communication/Collaboration diagrams

Elements in Interaction diagrams

- Instances of classes: These are shown as boxes with the class and object identifier underlined.
- Actors: These are represented using the stick-person symbol as in use case diagrams.
- Messages: Shown as arrows from actor to object, or from object to object.

Interaction diagrams Control Logic Representation

- Conditional Message
  - [variable = value] : message()
  - Message is sent only if clause evaluates to true
    e.g. [colour=red] calculate()
- Iteration (Looping)
  - * [i := 1..N]: message()
  - "*" is required; [...] clause is optional
    e.g.[i:=1..5]: st = gettotal()*

It is recommended that before an interaction diagram is created, a use case diagram and class diagram be developed.

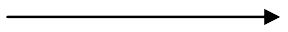
1. Sequence Diagrams

A sequence diagram is an *interaction diagram* that emphasizes the time-ordering of messages Sequence diagrams focus on the order in which the messages are sent and provide a sequential map of message passing between objects performing a certain task over time.

Sequence Diagrams are about deciding and modelling "how" the system will achieve "what" we described in the Use Case.

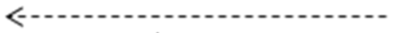
Sequence diagram Notation

- The objects are arranged horizontally across the diagram.
- The 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. The activation period is the time period during which an object performs an operation either directly or through a call to some subordinate operation.
- 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.
  - Messages can be of different types:
    - ☞ Synchronous Message: A type of message in which a caller has to wait for the receiving object to finish executing the called operation before it can resume execution itself.

      E.g. checkIfOpen in Figure 2.
      When a Registration Entry object sends this messgage to a Course Offering object, the latter responds by xecuting an operation called checkIfOpen. After the execution of this operation is completed control os handed back to the calling operation within Registration Entry with a return value of "true" or "false.
    - ☞ Asynchronous message: This is shown as a half arrow head or line arrow head in a sequence diagram and is one where the sender does not have to wait for the recipient to handle the message. The sender can continue executing immediately after sending the message.

    - ☞ Return (optional): Shows that the receiver has finished processing the message and returns control to the sender, draw a dashed arrow from receiver to sender.

Sequence Diagram Example

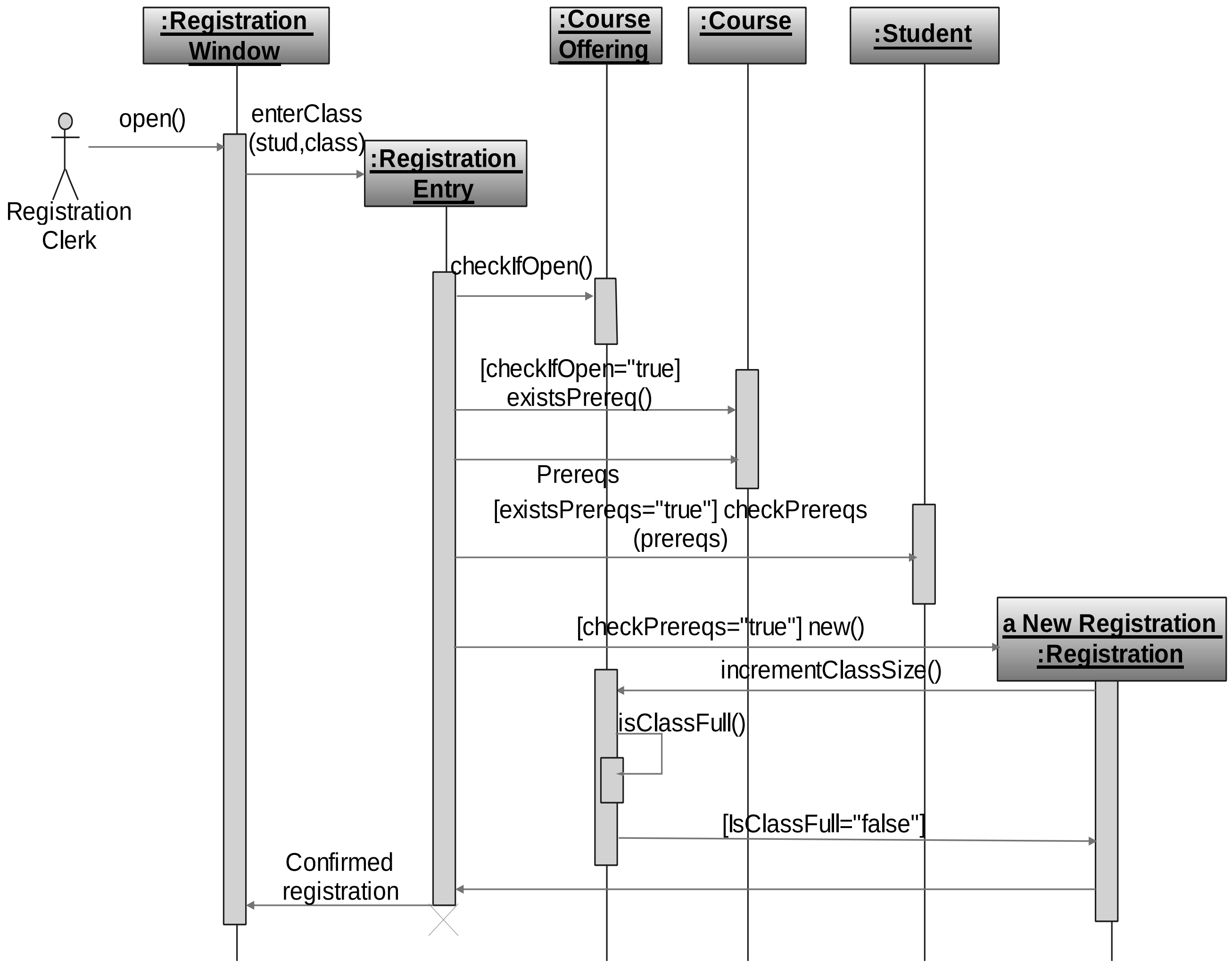


Figure 1: Sequence diagram for course registration system

Developing Sequence Diagrams

- Take the Use Case description and turn it into simple pseudo code running down the right hand side of the State Diagram.
- Guess which classes you think might be involved - based on the content of the Use Case description. Simple noun analysis is as good a way to start as any.
- For each of the steps in you pseudo code, decide which of the classes should have the responsibility for doing that task.
- For each of those tasks you may want to go back and decide to break them down into a number of simpler tasks.
- Add in probes that correspond to the "uses (includes)" and "extends" relationships in the Use Case diagram.
- Consider any important errors you might have to handle that perhaps weren't covered in the Use Case model.
- Consider whether anything you have discovered needs to be fed-back into the Use Case model.

2. Collaboration Diagrams

A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

They are very useful for visualizing the way several objects collaborate to get a job done and for comparing a dynamic model with a static model.

When creating collaboration diagrams, patterns are used to justify relationships

Examples of Collaboration Diagrams

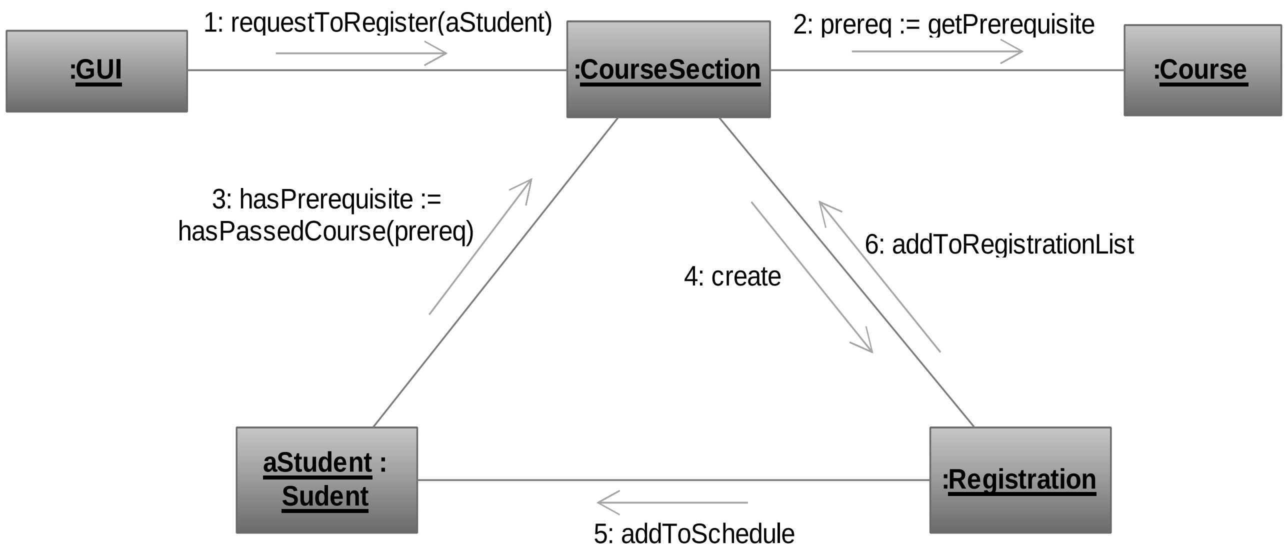


Figure 2: Collaboration diagram of course registration system

Note: Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other. Therefore collaboration and sequence diagrams describe the same information, and can be transformed into one another without difficulty.

The choice between the two depends upon what the designer wants to make visually apparent.

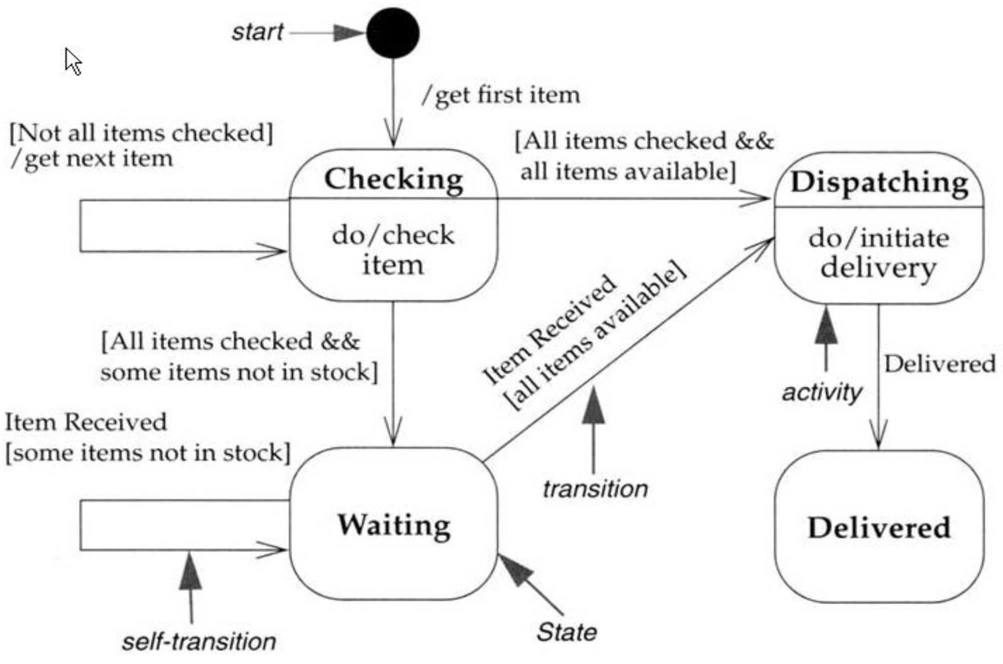
3. State Chart Diagram

A state chart diagram or simply a state diagram describes the behaviour of a *system*, some *part* of a system, or an *individual object* i.e. *the dynamic view of a system.*

- At any given point in time, the system or object is in a certain state.
  - ☞ Being in a state calls for a *specific behaviour* in response to any events that occur.
  - ☞ A state is represented by a rounded rectangle containing the name of the state
  - ☞ A black circle represents the *start state* while a circle with a ring around it represents an *end state*
  - ☞ Some events will cause the system to change state and 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.
  - ☞ A transition represents a change of state in response to an event; the label on each transition is the event that causes the change of state

Note: UML State charts are not normally needed. They are needed when an object has a different reaction dependent on its state

Notation



☞ Transitions labels have three optional parts: **Event [Guard] / Action**
e.g. **Item Received** is an event, **/get first item** is an action, **[Not all items checked]** is a guard

☞ State may also label activities, *e.g.* **do/check item**

- **Actions**: are associated with transitions, occur quickly and aren't interruptible
- **Activities**: are associated with states, can take longer and are interruptible

Example of State Chart diagram

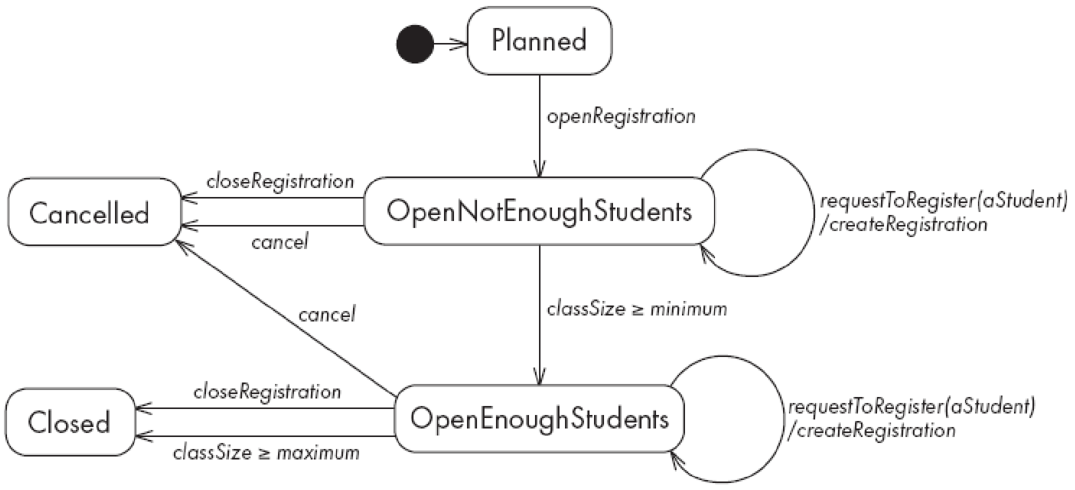


Figure 3: State diagram for class registration system

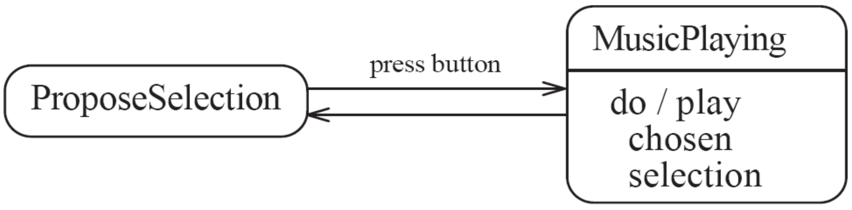
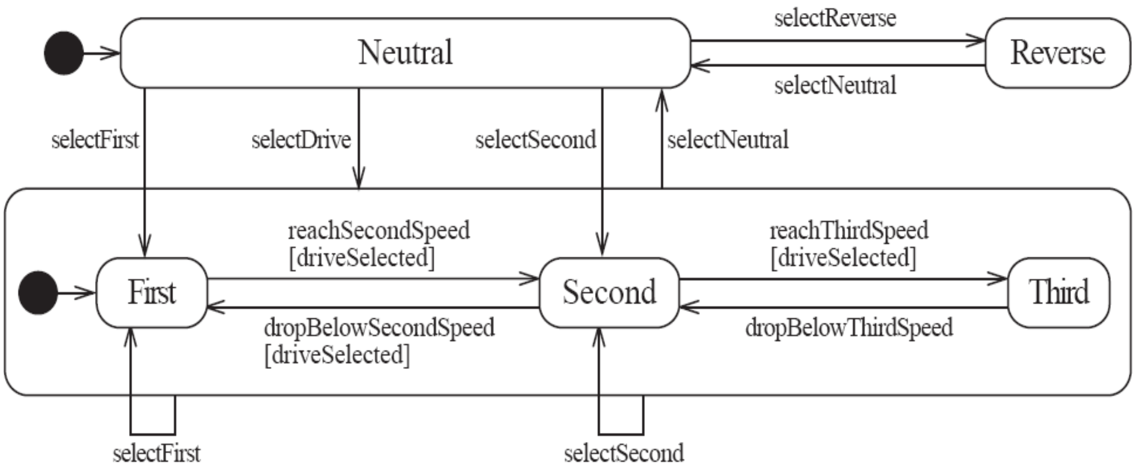


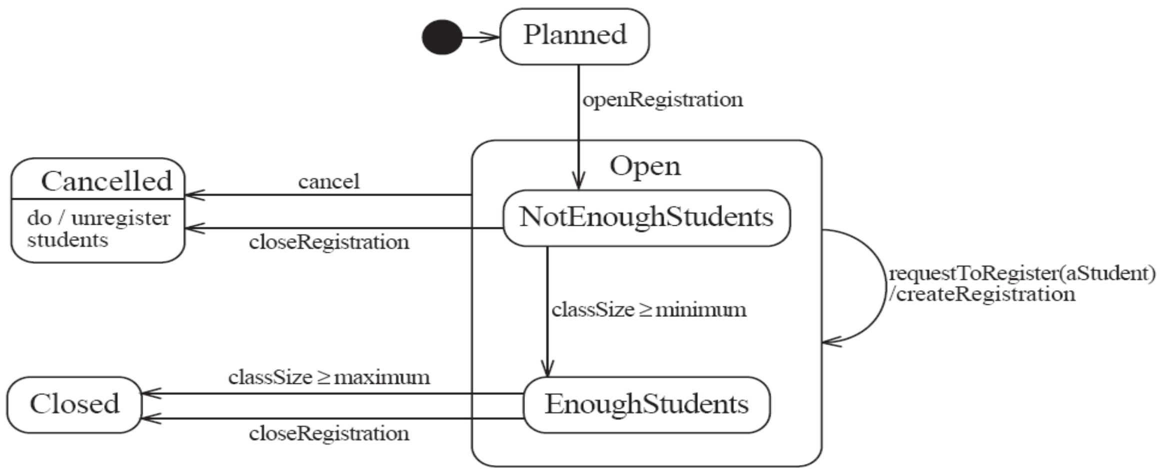
Figure 4: State diagram for Music Player (showing activities)

Nested sub states and guard conditions in State Diagrams

Gear Selection example



Class registration example

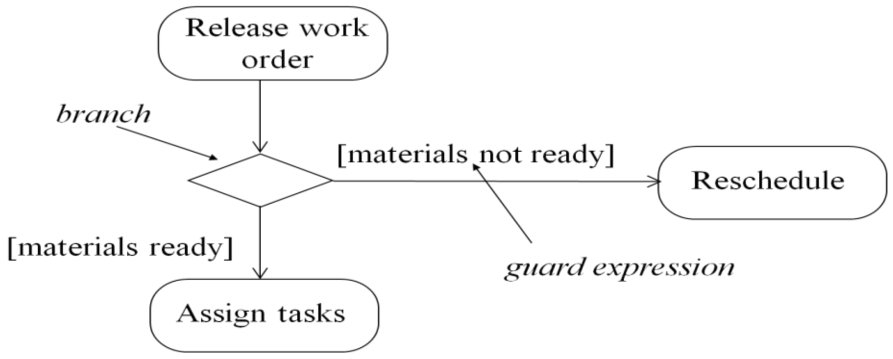


4. Activity Diagrams

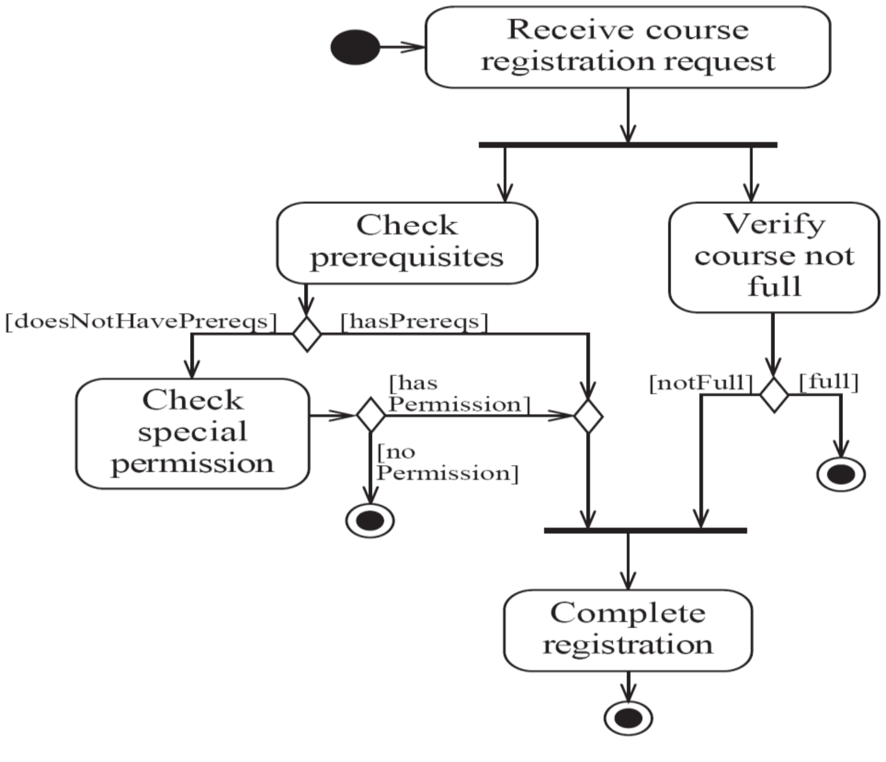
- An *activity diagram* is like a state diagram that shows the flow from activity to activity within a system.
  - ☞ Except most transitions are caused by *internal* events, such as the completion of a computation.
- An activity diagram
  - ☞ Can be used to understand the flow of work that an object or component performs.
  - ☞ Can be used to visualize the interrelation and interaction between different use cases.
  - ☞ Can show how different work flows or processes in a system are constructed, how they start, the many decision paths that can be taken from start to finish and where parallel processing may occur during execution.
  - ☞ Is most often associated with several classes and the partition of activities among the existing classes can be explicitly shown using *swimlanes*.

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

Notation



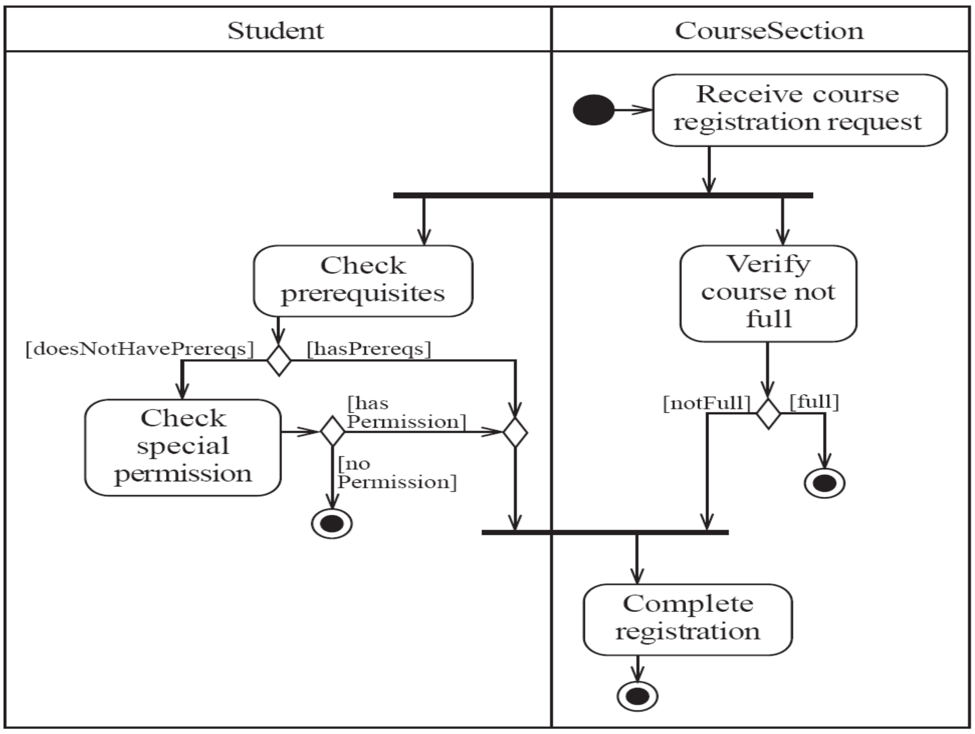
Activity Diagram Example



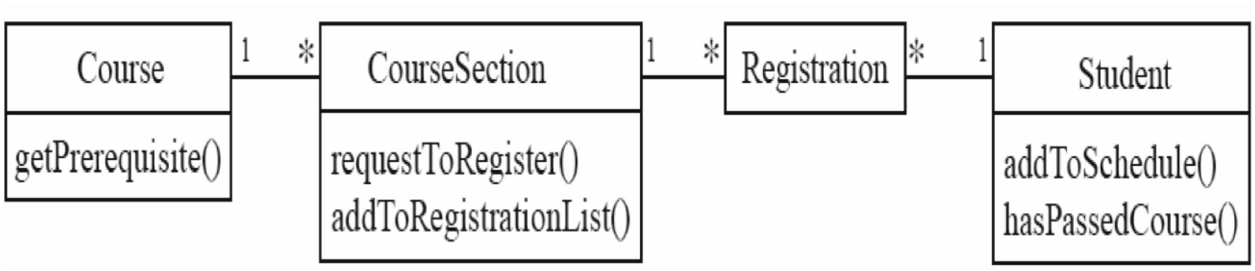
Representing concurrency

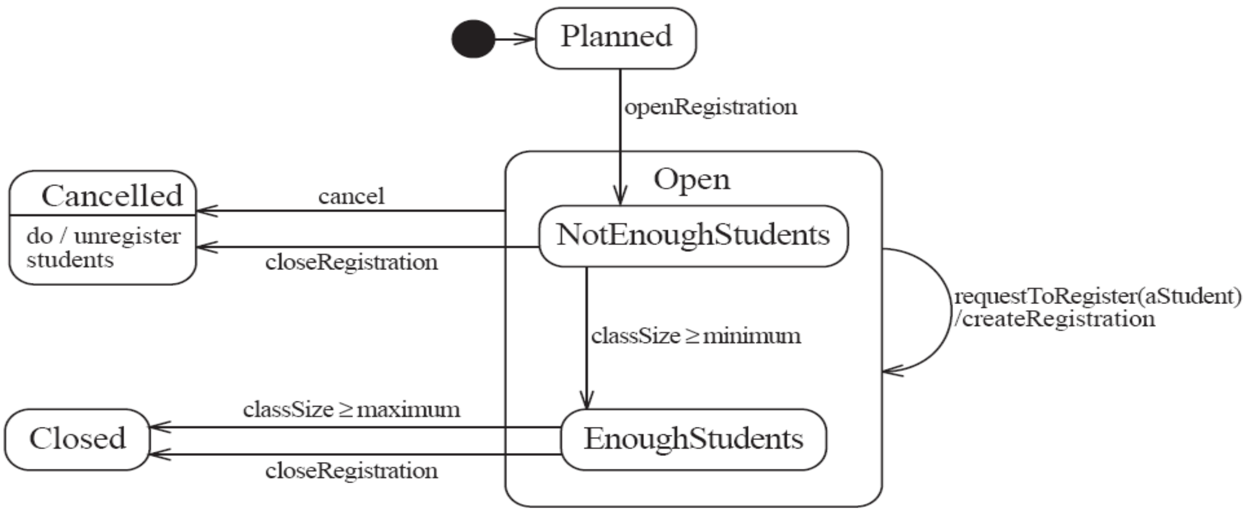
Concurrency is shown using forks, joins and rendezvous.

- 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.
- 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.



Analysis to Design: 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

- '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()

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

   // The 1-many association to class Registration
   private List registrationList;
   // The following are present only to determine the state
   // The initial state is 'Planned‘
   private boolean open = false;
   private boolean closedOrCanceled = false;

    public CourseSection(Course course)
    {
         this.course = course;
         registrationList = new LinkedList();
    }
    public void openRegistration()
    {
        if(!closedOrCanceled) // must be in 'Planned' state
       {
            open = true; // to 'OpenNotEnoughStudents' state
        }
    }

    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();
    }
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;
       }
    }
}

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

Implementation Diagrams

These are static structural diagrams that show aspects of model implementation, including source code structure and run-time implementation structure. There are two kinds;

- component diagram
- deployment diagram

5. Component Diagrams

A component diagram shows the organizations and dependencies among a set of components. They emphasize the physical software entity e.g. files headers, executables, link-libraries etc, rather than the logical partitioning of the package diagram.

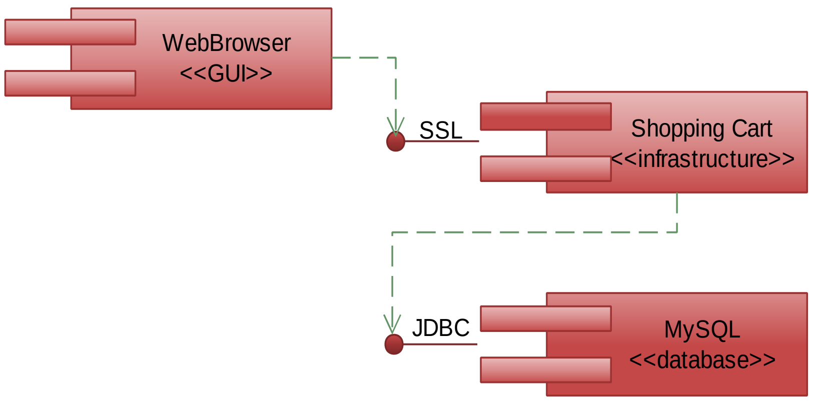
A component:

– Is a modular, replaceable and significant part of a system that packages implementation and exposes a set of interfaces

– May be implemented by artifacts (e.g., binary, executable, or script files)

Note: Though not heavily used, they can be helpful in mapping the physical, real life software code and dependencies between them.

Example

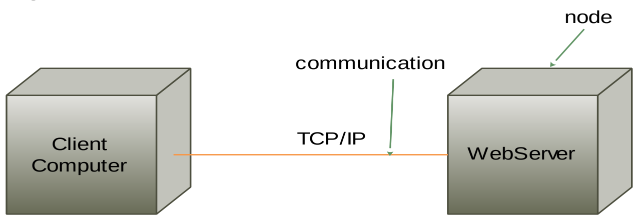


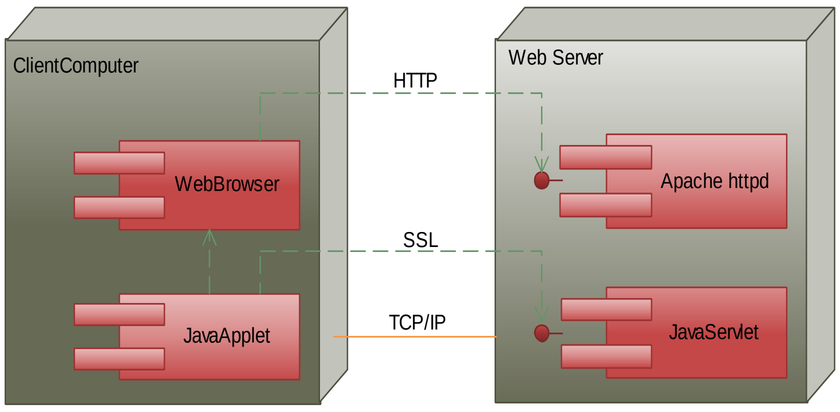
The Web browser communicates customer orders to a shopping cart which accesses a database for persistence and transactions

6. Deployment Diagrams

These show the configuration of run-time processing nodes and the software components, processes and objects that live on them i.e they show which components may run on which nodes. They address the static deployment view of architecture and are related to component diagrams in that a node typically encloses one or more components.

E.g.




ClientComputer
Web Server
HTTP
WebBrowser
Apache httpd
SSL
TCP/IP
JavaApplet
JavaServlet

UML Diagrams Summary

- Use Cases – How will our system interact with the outside world?
- Class Diagram – What objects do we need? How will they be related?
- Collaboration Diagram – How will the objects interact?
- Sequence Diagram – How will the objects interact?
- Statechart (or state) Diagram – What states should our objects be in?
- Component Diagram – How will our software components be related?
- Deployment Diagram – How will the software be deployed

The UML Model

When using models to describe a current or proposed system, at least the following viewpoints are needed:

1. **Data or structural view**
   This is a view of the system structure which contains the receptacles for the persistent data and the relationships of data artifacts with each other.
   In UML, this is the class diagram

2. **Functional view**
   This describes a model which clearly identifies, at least at a high level, the functions that the system must perform.
   In UML, this is the use-case diagram.

3. **Behavioural view**
   This view is a description of how the system components must react to specific external or internal events.
   In UML, the interaction diagrams, both sequence and collaboration diagrams and state diagrams, describe behaviour.

** *The functional view is usually generally regarded as part of the behavioural view.*

Structure Analysis and Design (SAD) vs. Object Oriented Analysis Design (OOAD)

Similarities

- Both SAD and OOAD had started off from programming techniques
- Both techniques use graphical design and graphical tools to analyze and model the requirements
- Both techniques provide a systematic step-by-step process for developers
- Both techniques focus on documentation of the requirements

Differences

- SAD is Process-oriented
- OOAD is Data-oriented
- Another difference is that OOAD encapsulates as much of the systems' data and processes into objects

The Design Model

- Data Design:
  - ✓ Transforms the information domain model created during the analysis into the data structures that will be required to implement SW.
  - ✓ Data objects and relationships in ERD and the detailed data content depicted in the data dictionary provide the basis.
- Architectural Design:
  - ✓ Defines the relationship among major structural elements of the program.
- Interface Design:
  - ✓ Descibes how SW communicates within itself, to systems that interoperate with it with humans who use it
  - ✓ Interface implies a flow of information.
- Procedural Design:
  - ✓ Transforms structural elements of the program architecture into a procedural description of SW components.

Design Specification

Design documents are an aid to making better designs

- They force you to be explicit and consider the important issues before starting implementation.
- They allow a group of people to review the design and therefore to improve it.
- Design documents as a means of communication.
  - ✓ To those who will be *implementing* the design.
  - ✓ To those who will need, in the future, to *modify* the design.
  - ✓ To those who need to create systems or subsystems that *interface* with the system being designed.

When writing the design document:

- Avoid documenting information that would be readily obvious to a skilled programmer or designer.
- Avoid writing details in a design document that would be better placed as comments in the code.
- Avoid writing details that can be extracted automatically from the code, such as the list of public methods.

Ref: DESIGN DOCUMENTATION – *pg 358* Software Engineering- A Practitioner's Approach

1. Scope

Information presented here may be derived from the System Specification and the analysis model (SRS).

   A. System objectives

   B. Major software requirements

   C. Design constraints, limitations

2. Data Design

   A. Data objects and resultant data structures

   B. File and database structures

3. Architectural Design

   A. Review of data and control flow

   B. Derived program structure

4. Interface Design

   A. Human-machine interface specification

   B. Human-machine interface design rules

   C. External interface design

D. Internal interface design rules

5. Procedural Design

For each module :

A. Processing narrative

B. Interface description

C. Design language (or other) description

D. Modules Used

E. Internal data structures

F. Comments/restrictions/limitations

6. Requirements Cross Reference

7. Test Provisions

8. Special Notes

9. Appendices

http://www.smartdraw.com/resources/tutorials/uml-diagrams/

http://www.sparxsystems.com/uml-tutorial.html

http://www.vtc.com/products/UML_tutorials.htm

VTC UML Video Tutorials