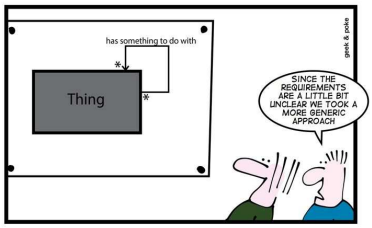


HOW TO CREATE A STABLE DATA MODEL

Software Modeling – UML

www.cs.uoi.gr/~zarras/http://www.cs.uoi.gr/~zarras/se.htm

Slides material sources:

Lethbridge and Laganiere, Object-Oriented Software Engineering Practical Software Development using UML and Java, 2005

Use case diagrams

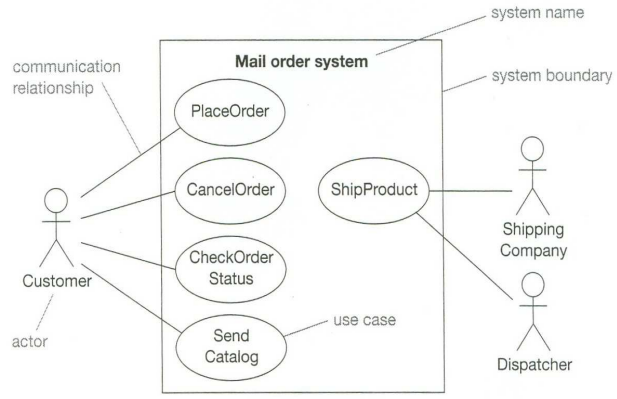


Figure 4.6

Use case name	**Use case: PayVAT**
Unique identifier	**ID: UC1**
The actors involved in the use case	**Actors:** Company Government
The system state before the use case can begin	**Preconditions:** 1. It is the end of a business quarter.
The actual steps of the use case	**Flow of events:** 1. Starts by the company at the end of a business quarter 2. The system determines the amount of Value Added Tax (VAT) owed to the Government. 3. The system sends an electronic payment to the Government.
The system state when the use case is over	**Postconditions:** 1. The Government receives the correct amount of VAT.

Figure 4.8

Branching in two ways

Use case: ManageBasket
ID: UC10
Actors: Customer
Preconditions: 1. The shopping basket contents are visible.
Flow of events: 1. The use case starts when the Customer selects an item in the basket. 2. If the Customer selects "delete item" 2.1 The system removes the item from the basket. 3. If the Customer types in a new quantity 3.1 The system updates the quantity of the item in the basket.
Postconditions: 1. The basket contents have been updated.

Branch in the normal process

Use case: DisplayBasket
ID: UC11
Actors: Customer
Preconditions: 1. The Customer is logged on the system.
Flow of events: 1. The use case starts when the Customer selects "display basket". 2. If there are no items in the basket 2.1 The system informs the Customer that there are no items in the basket yet. 2.2 The use case terminates. 3. The system displays a list of all items in the Customer's shopping basket including product ID, name, quantity and item price.
Postconditions: .....
Alternative flow 1: 1. At any time the Customer may leave the shopping basket screen.
Postconditions: .....
Alternative flow 2: 1. At any time the Customer may leave the system.
Postconditions: .....

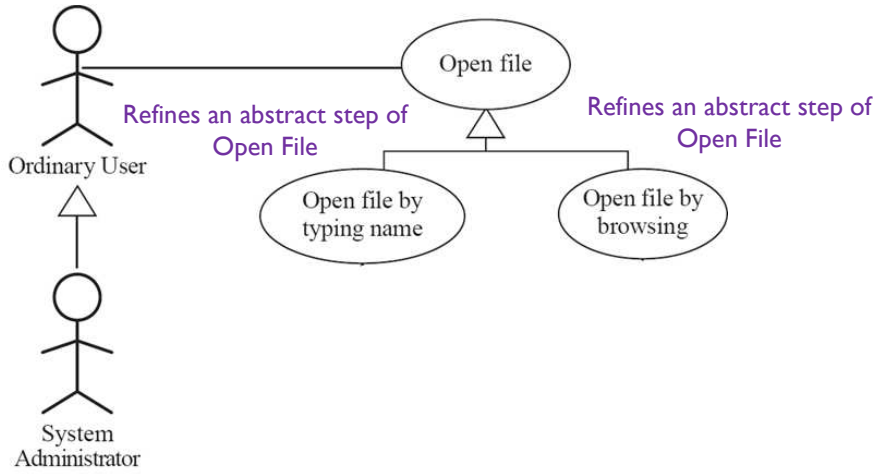
Branch that can happen sometime during the normal process – could be exceptional cases

For and While within use cases (rarely, however)

Use case: FindProduct
ID: UC12
Actors: Customer
Preconditions:
Flow of events: 1. The Customer selects "find product". 2. The system asks the Customer for search criteria. 3. The Customer enters the requested criteria. 4. The system searches for products that match the Customer's criteria. 5. If the system finds some matching products then 5.1. For each product found 5.1.1. The system displays a thumbnail sketch of the product. 5.1.2. The system displays a summary of the product details. 5.1.3. The system displays the product price. 6. Else 6.1. The system tells the Customer that no matching products could be found.
Postconditions:
Alternative flow: 1. At any point the Customer may move to different page.
Postconditions:

Use case: ShowCompanyDetails
ID: UC13
Actors: Customer
Preconditions:
Flow of events: 1. The use case starts when the Customer selects "show company details". 2. The system displays a web page showing the company details. 3. While the Customer is browsing the company details 3.1. The system plays some background music. 3.2. The system displays special offers in a banner ad.
Postconditions:

Use case and Actor specializations



Use case: **Open file**

ID:UC1

Actors:
Ordinary User

Related use cases:
Generalization of:
- Open file by typing name
- Open file by browsing

Pre conditions: The application is up and running

Basic Flow:
1. The use case starts when the user chooses the "Open" command
2. The file open dialog appears.
3. **The user specifies a file name**
4. The user confirms the selection
5. The file open dialog disappears

Post conditions: The contents of the file are available to the user for further processing

Use case: Open file by typing name

ID:UC2

Actors:
Ordinary User

Related use cases: Specialization of: Open file

Pre conditions: The application is up and running

Basic Flow:
1. The use case starts when the user chooses the "Open" command
2. The file open dialog appears.
3. **The user specifies a file name**
    1. **The user selects the text field**
    2. **The user types the file name**
4. The user confirms the selection
5. The file open dialog disappears

Post conditions: The contents of the file are available to the user for further processing

Alternative flow:
1. A file not found exception is raised by the system
2. The application indicates the file the user specified does not exist
3. The user corrects the file name in the text field

Post conditions: The use case continues from step 4 of the main flow

Use case: Open file by browsing

ID:UC4

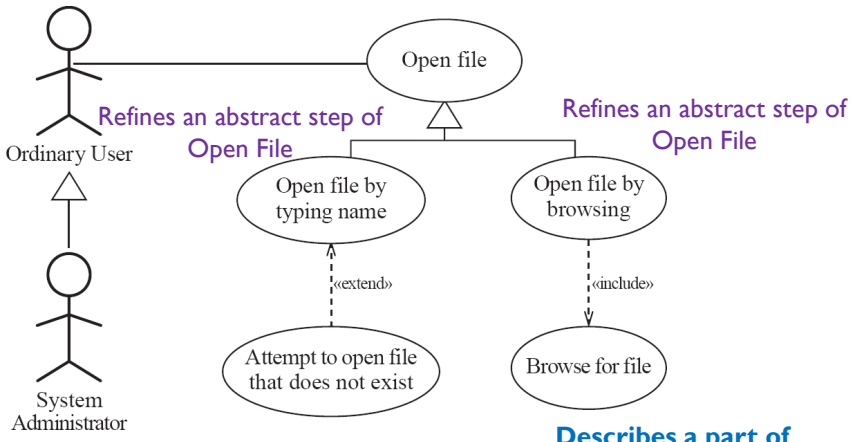
Actors:
Ordinary User

Related use cases: Specialization of: Open file

Pre conditions: The application is up and running

Basic Flow:
1. The use case starts when the user chooses the "Open" command
2. The file open dialog appears.
3. The user specifies a file name
    1. While the desired file is not displayed in the files list of the file browser
        1. The user selects a directory from the list
        2. The application displays the contents of the directory in the files list
    2. The user selects the desired file
4. The user confirms the selection
5. The file open dialog disappears

Post conditions: The contents of the file are available to the user for further processing



Describes a branching flow that is complex OR used in >1 use cases and can stand as a separate use case

Describes a part of the normal flow that is complex OR used in >1 use cases and can stand as a separate use case

Use case: Open file by typing name

ID:UC2

Actors:
Ordinary User

Related use cases:
Specialization of: Open file
Extended by: Attempt to open file that does not exist

Pre conditions: The application is up and running

Basic Flow:
1. The use case starts when the user chooses the "Open" command
2. The file open dialog appears.
3. **The user specifies a file name**
    1. **The user selects the text field**
    2. **The user types the file name**
4. The user confirms the selection
5. The file open dialog disappears

Post conditions: The contents of the file are available to the user for further processing

Alternative flow:
Attempt to open file that does not exist

Use case: Attempt to open file that does not exist

ID:UC3

Actors:
Ordinary User

Related use cases:
Extension of: Open file by typing name

Pre conditions: The user executes UC2

Basic Flow:
1. A file not found exception is raised
2. The application indicates the file the user specified does not exist
3. The user corrects the file name in the text field

Post conditions: The main flow of UC2 continues at step 4

Use case: **Open file by browsing**

ID:UC4

Actors:
Ordinary User

Related use cases:
Specialization of: Open file
Includes: Browse for file

Pre conditions: The application is up and running

Basic Flow:
1. The use case starts when the user chooses the "Open" command
2. The file open dialog appears.
3. The user specifies a file name
    1. **Browse for file**
4. The user confirms the selection
5. The file open dialog disappears

Post conditions: The contents of the file are available to the user for further processing

Use case: **Browse for file**

ID:UC5

Actors:
Ordinary User

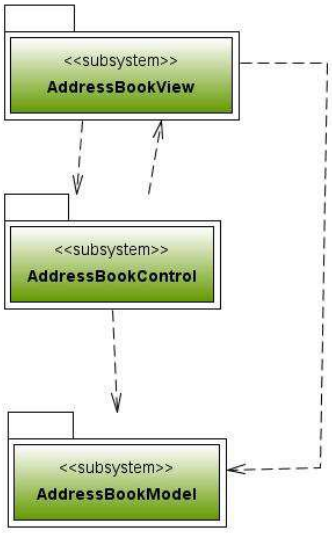
Related use cases:
Included by: Open file by browsing

Pre conditions: The application is up and running

Basic Flow:
1. The use case starts at step 3 of UC4
2. While the desired file is not displayed in the files list of the file browser
    1. The user selects a directory from the list
    2. The application displays the contents of the directory in the files list
3. The user selects the desired file

Post conditions: UC 4 continues at step 4

Package diagrams



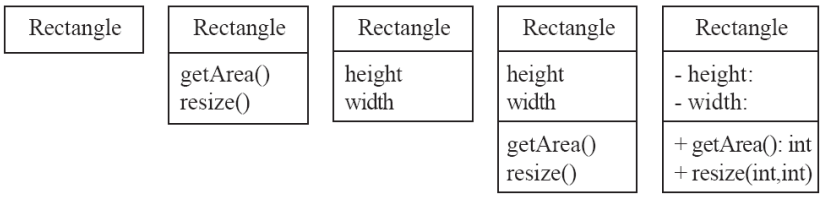
Packages logically group other elements (packages, classes, etc.)

A **dependency** from A to B means that A is based on functionality provided by B.

I also means that changing B may affect A.

Class diagrams

Classes



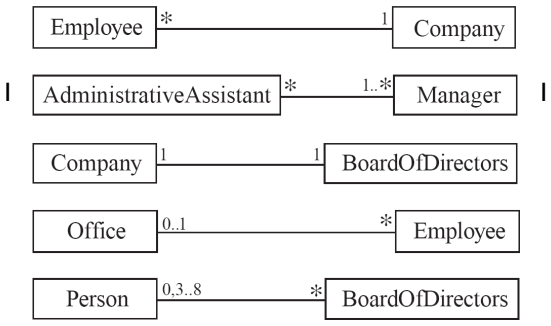
operationName(parameterName: parameterType …): returnType

Associations & Multiplicity

An ***association*** is used to show how **instances** (objects) of two classes are related/collaborate to/with each other

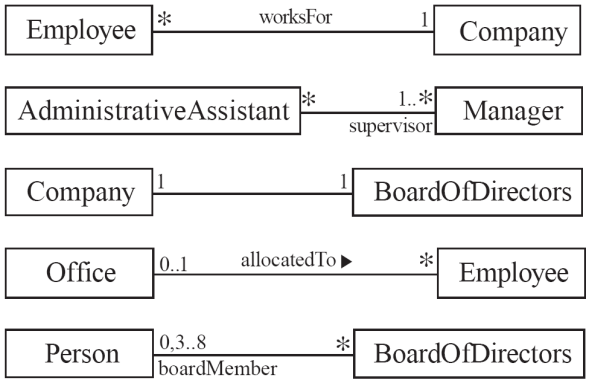
- Symbols indicating *multiplicity* are shown at each end of the association

***** in the implementation this relation could represent any form of interaction between objects that are created by other objects, passed as parameters to other objects etc.**



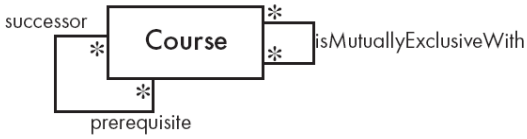
Labelling associations

Each association can be labelled, to make explicit the nature of the association



Reflexive associations

It is possible for an association to connect a class to itself

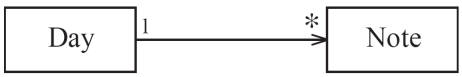


Directionality in associations

Associations are by default *bi-directional*

It is possible to limit the direction of an association by adding an arrow at one end

It determines in more detail how instances refer to each other, Day objects refer to Note objects

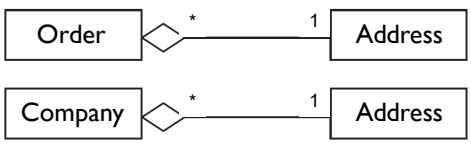


Aggregation

Aggregations are special associations that represent '**part-whole**' relationships.

***** in the implementation the "parts" are typically class attributes of the "whole" class**

- The 'whole' side is often called the ***assembly*** or the ***aggregate***
- This symbol is a shorthand notation association named isPartOf
- In general, the parts may be **shared** among different assemblies/aggregates
- Technically the aggregate class has **fields/attributes** of the part class



```
class Order {
        private Address addr;
        private ……
        ……..

        public Order(Address a) {
                addr = a;
        }
…..
}
    main() {
            Address a = new Address();
            Order o1 = new Order(a);
            Order o2 = new Order(a);

    }
```

```
class Address {
        …….
        ……
}
```

Composition

A *composition* is a strong kind of aggregation

- The parts cannot be shared among different assemblies/aggregates

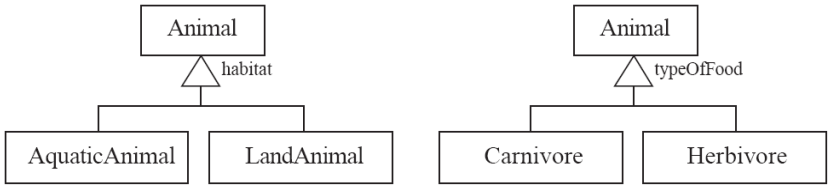
***** in the implementation the "parts" are typically class attributes of the "whole" class**

How can we tell if not shared?
The "parts" objects are created by the "whole"
The "whole" does not provide ways (e.g. getters) for getting references to the "parts"

Generalization

Specializing a superclass into two or more subclasses

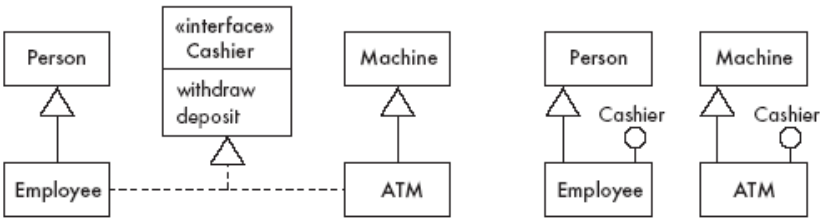
- A *generalization set* is a labelled group of generalizations with a common superclass
- The label (sometimes called the *discriminator*) describes the criteria used in the specialization



***** in the implementation this is done by extension/inheritance**

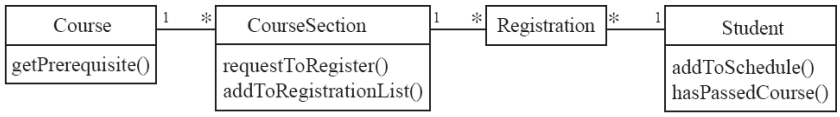
Interfaces

An interface describes a *portion of the visible behaviour* of a set of objects.



Note that conceptually ATM and Employee are not related somehow we have machines vs people … However with the interface we specify that some of the functionalities they provide are the "same"

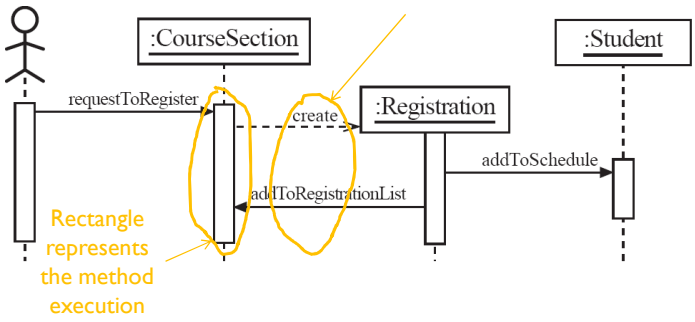
Sequence diagrams

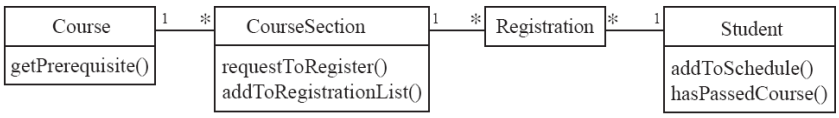


A class diagram does not tell us how do the class objects collaborate to realize a particular functionality

Basic interactions

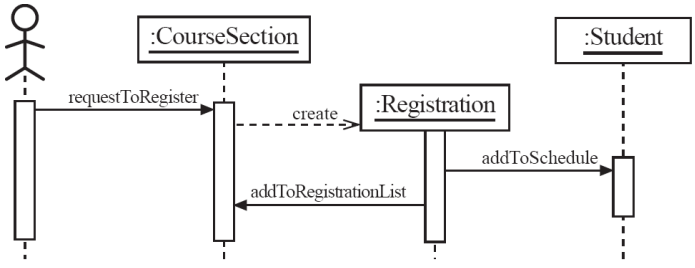
Happen during execution of requestToRegister()

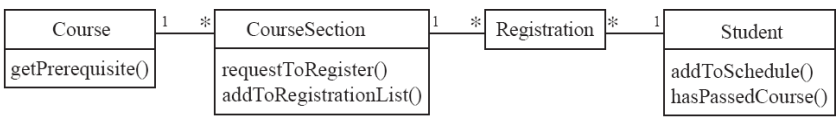




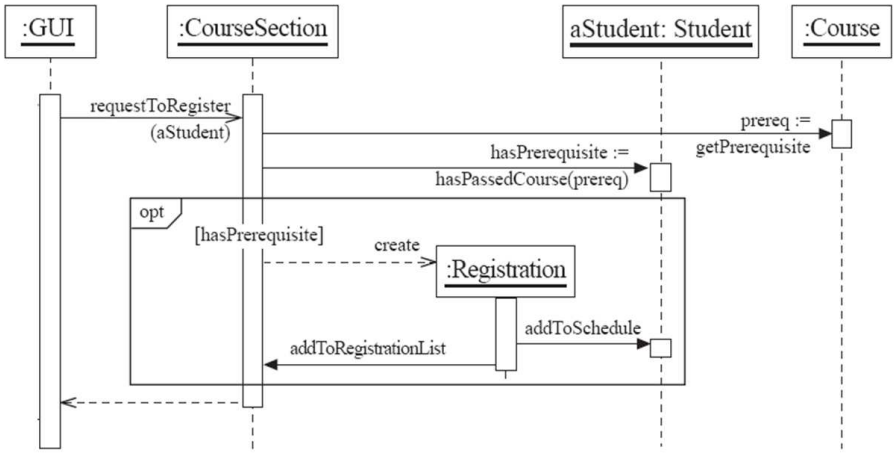
Basic interactions

Is it correct with respect to the class diagram???

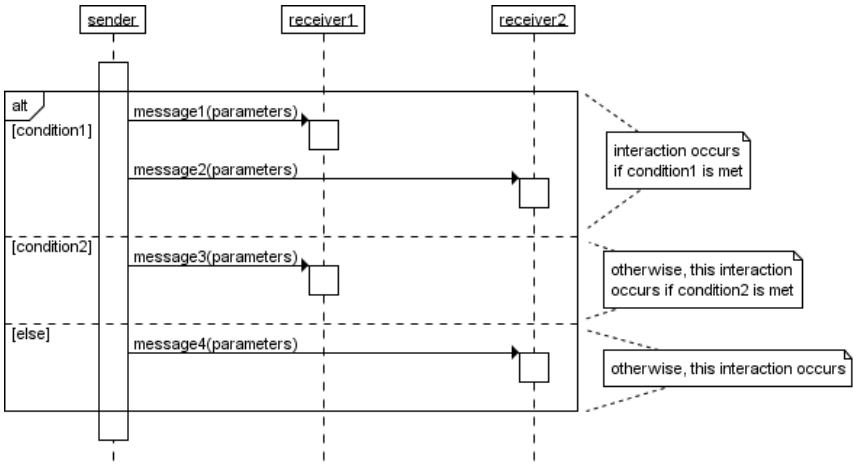




Alternative, optional interactions



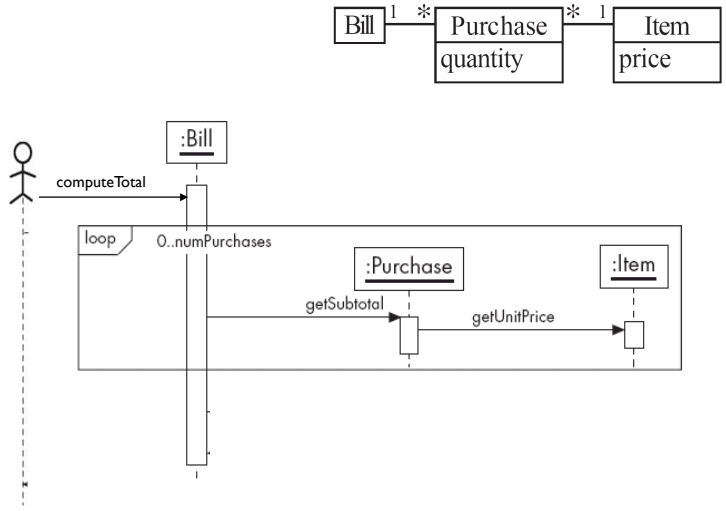
Alternative, optional interactions



Similarly can use an **[alt]** construct to show alternative flows

http://www.tracemodeler.com/articles/a_quick_introduction_to_uml_sequence_diagrams/

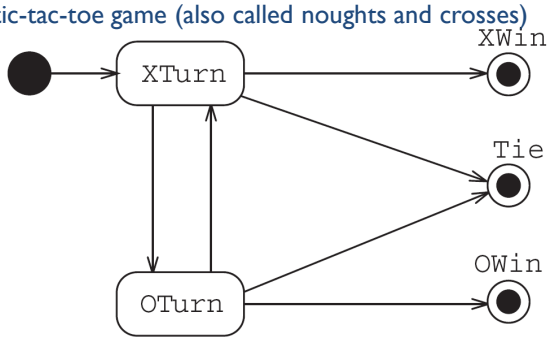
Iterative interactions



State diagrams

State diagrams

- Can be used to describe an automaton at the requirements level or at design/implementation level (how does a class works)
- tic-tac-toe game (also called noughts and crosses)



Requirements specification for the game

States

- At any given point in time, the system/object is in one 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*

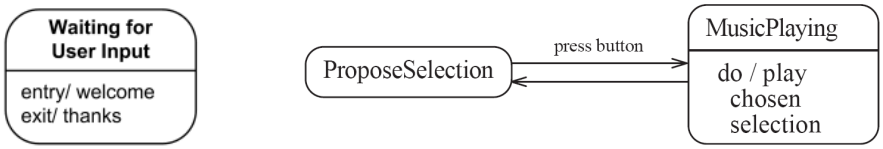
Transitions, actions, activities

- A transition represents a change of state in response to an event.
  - It is considered to occur instantaneously.
- The transition may be **labeled** :
  - trigger is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time.
  - guard is a condition which must be true in order for the trigger to cause the transition.
  - action is an action which will be invoked directly on the object that is described by the state machine as a result of the transition.

[trigger] [guard] / [action]

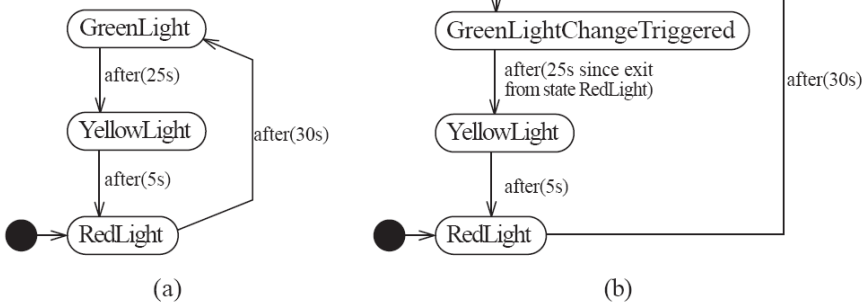
Transitions, actions, activities

- A state may have an associated entry behavior. This behavior, if defined, is executed whenever the State is entered.
- A state may also have an associated exit behavior, which, if defined, is executed whenever the state is exited.
- A State may also have an associated doActivity behavior. This behavior commences execution when the state is entered (but only after the state entry behavior has completed) and executes until it completes or the state is exited.



Time-outs and conditions

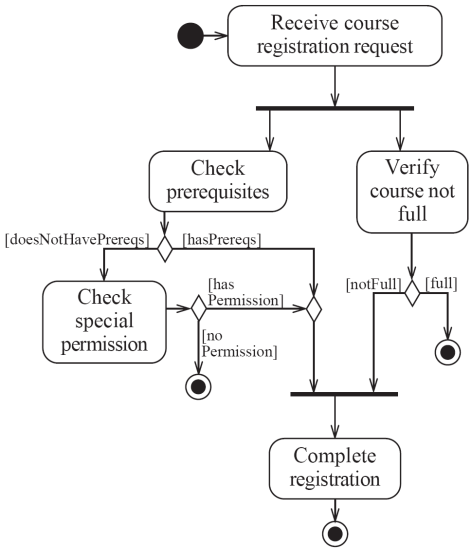
e.g. traffic light control logic



Activity diagrams

Activity Diagrams

- An activity diagram is typically used for process/algorithm modelling
- One of the strengths of activity diagrams is the representation of ***concurrent* activities**.



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

Swimlanes

- Activity diagrams are most often associated with objects of several classes.
  - The partition of activities among the existing classes can be explicitly shown using *swimlanes*.