Use Cases: Building the Functional Model

Functional Model of a System

- Recall that the *functional model* of a system is meant to describe the functionality from the user's point of view
- Usually built by identifying *use cases*. Can be represented in UML with Use Case Diagrams

Use Cases

- A **use case** is a description of a function of a system
    - Use cases are an analysis technique (done during requirements specification and analysis phases)
    - Intend to describe (sometimes formally) how a system should work
    - Use cases also describe the system's interaction with *actors*
- An **actor** is some external entity that interacts with the system
    - Could be a person performing a role (bank teller, system administrator)
    - Could be another system (a central database, a GPS satellite)
- Ideally, a use case should describe some single function of a system that is of value to (or yields some result for) a specific actor
    - More than one actor can be involved in a use case
    - Usually, a specific actor initiates a use case
- Examples:
    - In a voice mail phone system, a caller (actor) invokes the use case "Leave a Message". A mailbox owner (another actor) invokes a use case "Retrieve Messages". Each use case describes the steps needed to accomplish the task.
    - For an online shopping site, a shopper (actor) invokes use case "Place Order" (i.e. checkout). A shopper could also invoke use case "Contact Customer Service" -- this use case would result in providing information to a CS rep (another actor).
- A *use case model* is the set of all use cases of a system
    - This would be the *functional model* (a complete description of the system's functionality and environment
- A ***scenario*** is an instance of a use case

Specifying Use Cases

- Use cases could be specified formally or informally
- At a minimum, a use case should consist of:
    - a name that identifies it uniquely (across the system)
    - a main sequence of actions to be performed
    - any variations that might occur in the main sequence
- The Bruegge/Dutoit textbook uses a more formal description of a use case, which includes 6 fields:

  - use case name
  - participating actors
  - flow of events -- usually best to number them)
  - Entry conditions -- things that must be true for the use case to begin
  - Exit conditions -- things that will always hold when the use case ends
  - Quality requirements -- requirements not related to the functionality (performance constraints, hardware platform to be used, etc). i.e. these might relate to how the job is done, but not *what* is done
- Example of a use case (for a subway ticket machine):

Name	Purchase Ticket
Participating Actors	`Passenger`
Event flow	1. `Passenger` selects the destination 2. Distributor machine displays amount due 3. `Passenger` inserts money, of at least the amount due 4. Distributor returns change 5. Distributor issues ticket
Entry Conditions	o `Passenger` standing in front of ticket distributor o `Passenger` has sufficient money to purchase ticket
Exit Conditions	Passenger has ticket and any change given
Special Requirements	Once money inserted, ticket and change issued within 10 seconds

- Question: Why is *Distributor* **not** considered an actor?
- Regardless of the level of formality, use cases are written in natural language. Remember, this is the user's point of view being described!

UML: Use Case Diagrams

Use case diagrams graphically represent sets of use cases, communications with actors, and relationships between use cases

Basic diagram elements

- A use case is represented by an oval, labelled with the use case name

- An actor is represented with a stick figure (person), labelled with a name. The name is usually the *role* played by that actor
- A line between actor and use case represents communication.
  - Typically bidirectional (e.g. actor initiates a use case, a use case sends information to an actor)

Relationships (associations) between use cases

- **Extend** relationship
  - Typically represents exceptional or seldom invoked cases
  - Good to use when the original use case is getting cluttered with too many exceptional event flows
  - Can factor these out of the main event flow for clarity -- put in separate use cases
  - When main use case is invoked, some of the extended ones *might* be invoked, but they don't have to
  - Example: Use case `Buy Soda` might have a less-frequently occurring situation, like the machine being out of change. Use case `NoChange` could handle this, without always occurring.
  - Represent on a use case diagram with a dashed arrow between cases, labelled with `<<extend>>`, pointing towards the main use case
- **Include** relationship
  - When use cases contain some common behavior, the common steps can be factored out into a separate use case.
  - The original use cases would now *include* the new use case
  - This is like functional decomposition (factoring out common behavior when writing functions -- into a new function that the others can call)
  - Example: Use cases `Check Grades` and `Register For Classes` on the FSU system would both first involve the process of logging in. So we could create the `Log In` use case. The other two use cases would now include `Log In`
  - Represent on a use case diagram with a dashed arrow, labelled with `<<include>>`, pointing towards the included use case
- **Generalization** relationship
  - Similar to the class notion of inheritance
  - Used when there are categories and subcategories of use cases
  - Example: A high-level use case `Authenticate` is split into two more specialized use cases, `Authenticate With Password` and `Authenticate With Retinal Scan`. The original use case is the *generalization* of the two specific ones
  - Represent on a use case diagram just like in a class diagram. A line with a closed arrow, pointing from the specific case to the general case