Lecture 17:
Modelling System Interactions

- Interactions with the new system
  - How will people interact with the system?
  - When/Why will they interact with the system?
- Use Cases
  - Introduction to use cases
  - Identifying actors
  - Identifying cases
  - Advanced features
- Sequence Diagrams
  - Temporal ordering of events involved in a use case

Moving towards specification

→ What functions will the new system provide?
  - How will people interact with it?
  - Describe functions from a user's perspective
→ UML Use Cases
  - Used to show:
    - the functions to be provided by the system
    - which actors will use which functions
  - Each Use Case is:
    - a pattern of behavior that the new system is required to exhibit
    - a sequence of related actions performed by an actor and the system via a dialogue.
→ An actor is:
  - anything that needs to interact with the system:
    - a person
    - a role that different people may play
    - another (external) system.

Use Case Diagrams

→ Capture the relationships between actors and Use Cases

Notation for Use Cases

- Use case
- Actor
- Communication association
- System boundary
- Staff contact
- Accountant
- Change client contact
- Add a new client
- Record client payment
Example

Add new staff member
Add new staff grade
Change rate for staff grade
Change grade for staff member
Calculate staff bonuses

<<extends>> and <<uses>>

→ <<extends>> when 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.

→ <<uses>>: one use case invokes another (like a procedure call);
   % used to avoid describing the same flow of events several times
   % puts the common behavior in a use case of its own.

Sample use cases for a car

Driver GasAttendant Mechanic

Drive Fill Up <<uses>> Check Oil <<uses>> Fix Car

Fix car on the road

Meeting Scheduler Example

Initiator Participant

Generate Schedule <<uses>> Schedule meeting
Withdraw Edit Constraints Provide constraints
Validate User

<<extends>> <<uses>>
Identifying Actors

→ Ask the following questions:
  1. Who will be a primary user of the system? (primary actor)
  2. Who will need support from the system to do her daily tasks?
  3. Who will maintain, administrate, keep the system working? (secondary actor)
  4. Which hardware devices does the system need?
  5. With which other systems does the system need to interact with?
  6. Who or what has an interest in the results that the system produces?

→ Look for:
  1. the users who directly use the system
  2. also others who need services from the system

Finding Use Cases

→ For each actor, ask the following questions:
  1. Which functions does the actor require from the system?
  2. What does the actor need to do?
  3. Does the actor need to read, create, destroy, modify, or store some kinds of information in the system?
  4. Does the actor have to be notified about events in the system?
  5. Does the actor need to notify the system about something?
  6. What do these events require in terms of system functionality?
  7. Could the actor’s daily work be simplified or made more efficient through new functions provided by the system?

Documenting Use Cases

→ For each use case:
  1. prepare a “flow of events” document, written from an actor’s point of view.
  2. describe what the system must provide to the actor when the use case is executed.

→ Typical contents
  1. How the use case starts and ends;
  2. Normal flow of events;
  3. Alternate flow of events;
  4. Exceptional flow of events;

→ Documentation style:
  1. Choice of how to represent the use case:
     > English language description
     > Collaboration Diagrams
     > Sequence Diagrams

Generalizations

→ Actor classes
  1. It's sometimes useful to identify classes of actor
     > E.g. where several actors belong to a single class
     > Some use cases are needed by all members in the class
     > Other use cases are only needed by some members of the class
  2. Actors inherit use cases from the class

→ Use Case classes
  1. Sometimes useful to identify a generalization of several use cases
Modelling Sequences of Events

→ Objects "own" information and behaviour
- They have attributes and operations relevant to their responsibilities.
- They don’t "know" about other objects’ information, but can ask for it.
- To carry out business processes, objects have to collaborate.
  - by sending messages to one another to invoke each others’ operations
- Objects can only send messages to one another if they “know” each other
  - i.e. if there is an association between them.

→ Describe a Use Case using Sequence Diagrams
- Sequence diagrams show step-by-step what’s involved in a use case
  - Which objects are relevant to the use case
  - How those objects participate in the function
- You may need several sequence diagrams to describe a single use case.
  - Each sequence diagram describes one possible scenario for the use case
- Sequence diagrams...
  - should remain easy to read and understand.
  - do not include complex control logic

Example Sequence Diagram

Initiate Person

Respond

Iteration

Time

condition

[decision=OK] ScheduleOK

Initiator: Person

Participant: Person

Staff: Person

Scheduler: Person

Acknowledge()

Acknowledge()

branching messages, etc

CustomerP

PrinterP

Printer

Queue

Lifeline

PrintFile(file)

GetStatus()

[Ready] Print()

[Busy] PutInQueue(file)

[OutOfService] CallRepair

[Ready] GetNext()

Active

Inactive

Asynchronous

Branching

GetNext()

PrintUser(file)

GetFile(file)

[Ready] PutInQueue(file)

[Busy] CallRepair

[Ready] GetNext()
Don't forget what we're modelling

→ During analysis
  δ we want to know about the application domain and the requirements
  δ so we develop a course-grained model to show where responsibilities are, and how objects interact
    > Our models show a message being passed, but we don't worry too much about the contents of each message
    > To keep things clear, use icons to represent external objects and actors, and boxes to represent system objects.

→ During design
  δ we want to say how the software should work
  δ ... so we develop fine-grained models to show exactly what will happen when the system runs
    > E.g. show the precise details of each method call.