Requirements
Use Cases
Scenarios

About Models

- Generally, a model captures *implicitly* lots of design decisions
- A model typically abstracts away from implementation details:
  - advantage: the design should be reusable whatever the implementation is
  - disadvantage: the level of abstraction may be too high, leaving the door open for more design decisions during implementation
  - inevitability: no design model captures all details and subtleties of coding. Consequently:
    - » there may be some semantic gap between design and code
    - » coding decisions must be motivated (if not traced back to requirements) and explained.
- In this course, we are concerned with modeling decisions.

The Problem

- Given some problem statement:
  - Where do we start?
  - What models are we to use?
  - How do we capture design and coding decisions?
  - How do we capture traceability links?
  - How do we capture alternative designs, if at all?
- A 'recipe' (a s/w dev. method) should provide answers to all these questions:
  - We will adopting a general recipe called "scenario-driven development" (next slide)

Scenario Driven Development

- Capture/re-express requirements in the form of *use-cases* (UCs)
- Get a set of UCs: a use-case is a set of *scenarios*
- From use-cases, which view the system as a black-box, extract, a set of *representative* (high yield or essential) scenarios, that is, interactions between the components of the system
- From these *interaction diagrams*, or concurrently with their design, establish the *architecture* of the system (e.g., using a *class diagram*)
- Capture the internal workings of each class using a *state diagram* called a *statechart* or code the class.

Learning Objectives

- distinguish the words: requirements, use-cases, and scenarios
- present a general format for capturing assumptions, requirements and use cases
- introduce UML 's use-case and use-case diagrams

About Requirements

(Software) Requirements Engineering

- "The process of discovering [the] purpose [of a software system] by identifying stakeholders and their needs and documenting these in a form that is amenable to analysis, communication, and subsequent implementation" (B. Nuseibeh, S. Easterbrook)
- Requirements elicitation:
  - What problems need to be solved?
  - Where is the system boundary?
  - Who are the stakeholders and what are their goals?
- What is captured needs to be *communicated*, *validated*, and *agreed on* with stakeholders
  - Modeling is one common technique in support of these objectives (but not the only one)

Common Approaches to Specifying Requirements

- "Ad hoc" analysis
  - Mixture of different kinds of requirements (what, how, how well, etc.)
  - Unsystematic, informal, and error prone (omissions, conflicts, inconsistencies, ambiguities)
  - However, usually, the most common starting point
- Use case analysis
  - System viewed as a set of functional capabilities that realize desired stakeholder goals
  - An operational view of a system
- Feature analysis
  - System viewed as a collection of desired properties (features)
  - A capability-based view of a system

Why Isn't The Informal Spec Sufficient?

- The informal nature of the text makes it difficult
  - To identify potential conflicts and inconsistencies between requirements
  - To determine any interdependencies between individual requirements
  - To ensure all requirements are covered
  - To understand precisely what needs to be implemented
  - To trace requirements to implementation and vice versa
  - To provide a basis for comprehensive testing
- Need a more systematic and more precise approach to requirements specification
- Also, need requirements in electronic form for easier manipulation (searching, linking, tracing)

Why are Requirements so Important?

- Attention to requirements is a basis of building quality products
- Because satisfying requirements is so fundamental, requirements form a basis for managing a development project

  - requirements model the problem
  - requirements form the basis of agreements
  - requirements form the basis for analysis
  - requirements form the basis for testing
  - getting requirements right saves money

- It is crucial to acknowledge the fact that requirements will change over time
- We do not focus in this course on requirement gathering techniques, nor on standards (eg from IEEE) for capturing requirements.

Types of System Requirements

- **Functional requirements**

  Define what the system must do
- **Nonfunctional requirements (quality of service)**

  Define the constraints that apply to the system and/or to the development of the system
  - **Performance requirements** (time-constraints)
  - **Interface requirements** (usability)
  - **Resource requirements** (e.g., what 3rd party s/w can you use)
  - **Verification requirements** (i.e., how much testing?)
  - **Documentation requirements** (captured in templates)
  - **Security requirements** (e.g., level of encryption)
  - **etc.**

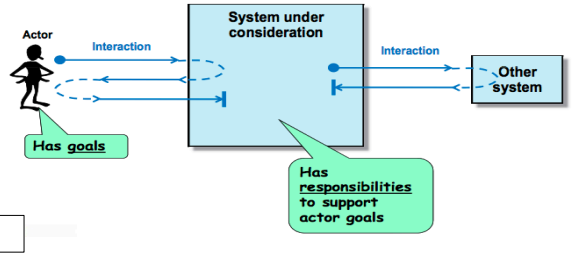
Structuring Requirements

- **Organizing requirements is a very important task**
  - Requirements are typically incomplete:
    - » we must capture their evolution (i.e., tracking changes)
    - » we must often complete them with our own assumptions
  - Requirements capture must be structured so that it facilitates quick access to specific requirements
  - Throughout documentation, traceability is key:
    - » To verify completeness
    - » To motivate functionality (but not commit to design decisions!)
- **So let's look at how to deal with assumptions and requirements for a specific example: see Poker case study.**

Use Case Modeling

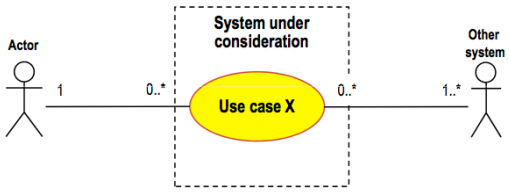
The Use Case Conceptual Framework

- An interaction with a system for the purpose of achieving a clearly defined goal
  - Something which is of value to the actor



UML Representation of Use Cases

- The UML graphical representation is merely a shorthand rendering of a use case
  - The important (and hard) part is specifying the use case itself



Key Concepts

- *NB: There is still no visible consensus on the definitions of key use case concepts*
  - A workshop of 14 leading OO consultants had 14 definitions of "use case"
  - Occurs on the boundary between the informal (i.e., what users want) and formal (i.e., what engineers implement)
- Actor: an entity (person or system) which interacts with a system to achieve one or more goals
- Use case: a *collection* of possible interactions between actors and a system relating to one or more goals
  - Being a collection of possible scenarios, a use case is similar to a class
  - ..we talk about "instances" (occurrences) of use cases

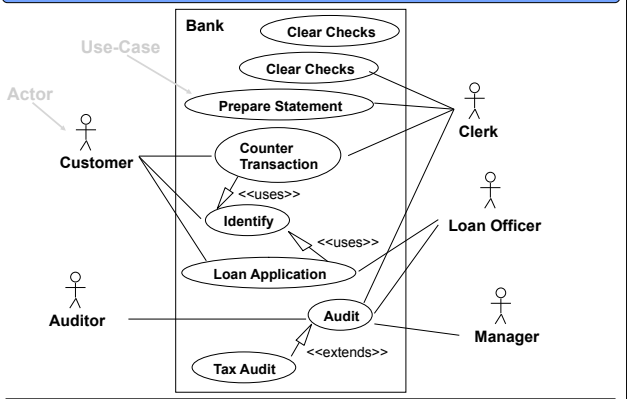
Steps of UC Modeling

- For event-driven systems, UC modeling consists of the following steps:
  - Scope the system (by considering different Actor perspectives)
  - Identify events and *actors*
    - » Actors are abstractions generating events
    - » Think of internal and external events
  - List use-case titles (and prioritize them)
  - Produce a use-case diagram
  - Document use-cases using a scenario textual description (STD) technique
    - » We want the STDs in the design document.

Identifying External Events

Event	System Resp.	Arrival	Response
offhook -->	dialtone	aperiodic <100/min	<500msec
first digit -->	cancel dialtone	aperiodic <20 sec after dialtone	Digit tone <100msec
last digit -->	translation result	interdigit time = 4sec	a.s.a.p
<--ringing			a.s.a.p after last digit
answer-->	cancel ringing and ringtone	aperiodic	<100msec

A Use-Case Diagram



An Example Use Case

Use Case: Making a successful POTS connection

- Actors: calling party, called party
- **Scenario:**
- Caller lifts telephone receiver
- Caller hears dial tone
- Caller dials digits
- Caller receives audible ring tone
  Called party's phone rings
- Called party lifts receiver
- Caller and Called party are now connected and can talk
- Called party hangs up
- Caller receives dial tone
- Caller hangs up

Key points: System as Black Box, event/response

How to Start?

- You must start by writing down a list of *verifiable* requirements and going from them to UCs.
- Each use case captures a cluster of scenarios:
  - the scenarios of a UC must be logically clustered together
  - a scenario is formed by a (more or less abstract) sequence of input/output events processed by the system (as a black box)
  - through the use of words such as 'OR', 'AND', 'eventually', 'optionally', 'repeatedly', each step of a UC, each scenario, and ultimately each UC can be viewed as a grammar of events
  - In OO, we use a set of UCs to describe system behavior:
    - » unless otherwise documented, UCs are taken to be independent of and concurrent with each other
    - » inter-UC relationships (annotated with stereotypes) are important to identify: the more the UCs are tied to each other, the less partial the overall specification is!
    - » there is generally no overall grammar to build for the whole system but we do aim for req. coverage (via traceability)

Organizing Use Cases

- We propose that each use case be documented using an STD that ideally contains the following information:
  - » a unique identifier
  - » a brief textual description of the overall objective of the UC
  - » the set of external actors that participate in the UC
  - » a set of possible triggering events
  - » a pre-condition that must be satisfied in order to enable the execution of the UC
  - » a sequence of system *responsibilities (or steps)* for the main scenario (JP: if not for ALL scenarios!!!)
  - » a set of possible resulting events for the UC
  - » a post-condition that must evaluate to true after the execution of the UC
  - » a set of alternative scenarios (optional but important!)
  - » a set of nonfunctional requirements that apply to the UC (optional)
  - » a comment section that may be used by designers as a free format text window to specify different issues related to the UC (e.g., which scenarios were grouped into this UC)

Example STD (1)

Use Case Identifier: UC-9 ATM withdraw transaction
Description: Describes the steps of a normal withdraw transaction
External Actors: User, Central Bank System (CBS)
Related Use Cases: UC-15 Transaction
Precondition: ATM is idle **Triggering event:** A user inserts a valid bank card
1. User enters a valid bank card. **2.** ATM swallows the bank card and reads the card information. **3.** ATM initiates the transaction. **4.** ATM asks the user to enter PIN. User enters PIN. **5.** CBS validates PIN. **6.** ATM asks user to choose a transaction option. User chooses the withdraw option. **7.** ATM asks for amount to withdraw. User enters amount. **8.** ATM sends a withdraw transaction request to CBS. **9.** CBS verifies that the user account balance is sufficient to cover the requested amount. **10.** CBS registers the withdraw transaction. **11.** ATM dispenses cash. User picks up cash. **12.** ATM prints a transaction receipt. User picks up the receipt. **13.** ATM returns the bank card. User picks up the card.

Example STD (2)

Use Case Identifier: ATM withdraw transaction
Resulting event: ATM returns the bank card **Postcondition:** ATM is idle again
Alternatives: - If the user enters three successive invalid PINs, then the transaction is refused and the card is kept. - If the user's account balance is insufficient, then the transaction is refused. - If the ATM does not have enough cash, then the transaction is refused.
Nonfunctional requirements: - A transaction must be completed in less than two minutes - ATM can only handle one transaction at the time.
Comments: - A transaction can be cancelled at any time before the transaction is sent to the CBS.

UML 's Stereotypes and Packages

- A *package* can be used to regroup a set of use-cases
  - a package can also be used to regroup other UML entities, such as classes
  - it constitutes a grouping mechanism for scalability in UML
- A *stereotype* is a user-defined label that allows extensions to the semantics of UML
  - this is a key mechanism to introduce your own semantics into the modeling process

Stereotypes for Use-Cases

- A *Basic* Use-Case:
  - must describe a typical usage of the system from end-to-end
  - must keep an external, event-driven perspective
- An *Extension*
  - captures functionality that is optional or additional to one or more basic use cases
    - » We prefer to list alternatives inside an STD.
  - is related to basic a use-case using an *extends* arrow
- A *Reference*
  - gives a name to a group of steps repeated in several use-cases
  - is related to basic a use-case using a *uses* arrow

Examples and Exercise

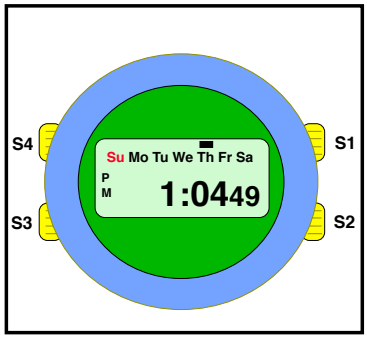
On Gomaa's Elevator Use Cases

- Consider figure 18.2 and associated UCs
  - Stop Elevator UC:
    - » sensor talks to system (not to elevator)
    - » system determines whether elevator is to stop or not
    - » system commands to elevator 's door...
  - Dispatch Elevator UC:
    - » awkward wording: « the system moves »…
    - » the elevator does not determine its direction
  - Select Destination UC: (elevator request)
    - » system is to keep a list of floors to visit for the elevator
    - » step 4 is clumsy from a temporal viewpoint
  - Request Elevator UC: (floor request)
    - » step 2: commitment to an early-decision approach!!
- Let's now look at other examples☺

Exercise

- Read the requirements for the watch described in the next two pages.
- Scope the system:
  - What must you worry about? Is it clearly stated in the reqs?
- Identify relevant internal and external events.
- List and possibly prioritize:
  - relevant functional requirements
  - relevant UC titles
- Produce one Use-Case diagram.
- Develop the use-case that addresses the setting of time. Try to use the proposed format.
- Time-permitting, develop other use-cases.

The Watch



The Watch Requirements

A watch must display and maintain current time and date. In future iterations, it is envisioned it will also have a few bells and whistles such as a light (activated by button S4), a stopwatch, and an alarm.

The watch has 3 other buttons. S3 is the function selector to toggle between the different displays (time, date, stopwatch, alarm and back to time). A long S3 (pressed at least 2 seconds) is used to go in update mode for the time, date and alarm displays. In update mode, S1 can move between the items to update in the current display and S2 used as an increment/toggle button.

The time functions consist in 1) displaying time (in the form of day-of-the-week, am/pm indicator, hours, minutes and seconds) and 2) setting the time.

The date functions consist in 1) displaying date in the form of day and month and 2) setting the date.

Setting the time is initiated by a long S3 while in time display. S1 is used to select the item(s) to update. The order for updates is: seconds, minutes, hours, am/pm, day-of-the-week, and back to seconds. The user chooses what to update and in what order. For am/pm S2 acts as a toggle, for day-of-the-week it increments from 'Su' to 'Mo' to 'Tu' to 'We' to 'Th' to 'Fr' to 'Sa' and loops. For all other items, S2 acts as an increment by 1.

Setting the date is initiated by a long S3 while displaying the date. Using S1, the day and then the month can be updated.

The stop-watch is accessed by S3 from the date display. It is started and stopped by S1. It can keep running even if the display is changed to some other thing. A long S1 resets the stopwatch to zero.

The alarm display is accessed by S3 after the stopwatch display. It shows the minutes, hour and am/pm for the alarm, as well as an on/off toggle. The alarm rings for 20 seconds when the watch reaches the alarm time and the alarm is on.

While in alarm display mode, a long S3 will put the watch in alarm update mode. S1 is then used to select minutes, hour, am/pm, on/off. S2 is again the increment button.

While in any update mode, current item to update will flash. Also in update mode, pressing S3 will immediately bring the display back in normal mode.

Some Conclusions

Why OO people like Use-Cases

Use-cases:

- constitute a simple, intuitive form of scenario modeling
  - temporal logic for event specification is much more complicated
- are not object-oriented
  - only solutions to the requirements are OO!
- make clear what external functionality is expected
  - the system is treated as a black-box
  - the interface and the DB functionality are typically separated
- may be helpful infinding objects
  - how to do this is discussed later in COMP 3004
  - only *domain* (i.e., problem as opposed to solution) objects should be mentioned in use-cases
- are traceable to detailed interaction diagrams used later in the design process
- may be used as a basis for black-box testing of the system

Working with Use-Cases

- Use-cases proliferate quickly:
  - It is naive to think you can simply write down all of the use-cases and *exhaustively* describe the behavior of the system
  - We repeat, it is easy to confuse scenarios, their steps, and use-cases
- Several authors suggest finding "key" scenarios and use-cases
  - but no one gives good guidelines for selecting such "key" use-cases... See Wirfs-Brock tutorial
- Don't forget about scenario interactions (next slide)

About Scenario Relationships

- It is important to group together related scenarios into a single scenario *cluster*:
  - A use-case should be thought of as a cluster of related scenarios
  - Exception handling scenarios can be viewed as extensions or alternatives of basic use-cases
- Individual scenarios are typically straightforward. It is essential to capture the *relationships* between scenarios! The same holds for use-cases!!
  - Such relationships typically define at least a temporal, if not a *causal* order between scenarios, and/or between use-cases.
  - A use-case diagram may be used to document inter-UC relationships. But we need lots of stereotypes!

The Bottom Line

- Don't do procedural decomposition through the use-cases
  - don't describe algorithms or specific paths of execution inside the system
  - Each scenario of a UC is an end-to-end sequence of events corresponding to a typical use of the system, which is viewed as a black box.
- Review use-cases with respect to completeness and consistency
  - trace individual scenarios to requirements
  - inter-scenario and inter-UC relationships are crucial in verifying consistency