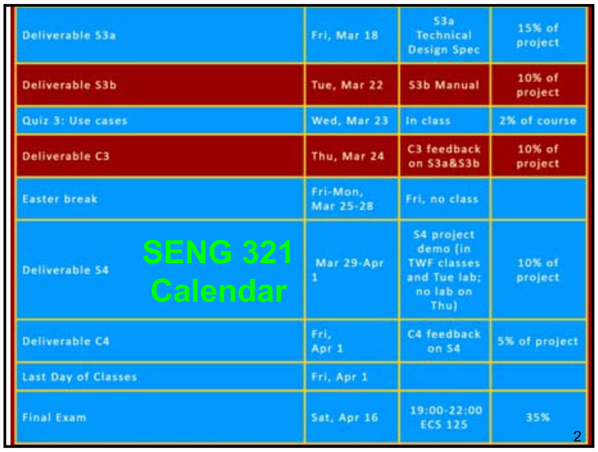
Professor Hausi A. Müller PhD PEng FCAE
Department of Computer Science
Faculty of Engineering
University of Victoria

www.engr.uvic.ca/~seng321/
courses1.csc.uvic.ca/courses/201/spring/seng/321



SENG 321 Calendar

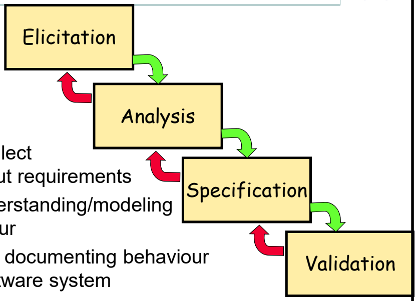
Deliverable S3a	Fri, Mar 18	S3a Technical Design Spec	15% of project
Deliverable S3b	Tue, Mar 22	S3b Manual	10% of project
Quiz 3: Use cases	Wed, Mar 23	In class	2% of course
Deliverable C3	Thu, Mar 24	C3 feedback on S3a&S3b	10% of project
Easter break	Fri-Mon, Mar 25-28	Fri, no class	
Deliverable S4	Mar 29-Apr 1	S4 project demo (in TWF classes and Tue lab; no lab on Thu)	10% of project
Deliverable C4	Fri, Apr 1	C4 feedback on S4	5% of project
Last Day of Classes	Fri, Apr 1		
Final Exam	Sat, Apr 16	19:00-22:00 ECS 125	35%

Announcements

- Fri, March 18
  - S3a due
  - Detailed technical design spec
- Tue, March 22
  - S3b due
  - User manual due
- Fri, March 25
  - Good Friday, no class
- Tue/Wed/Fri, March 29/30, April 1
  - In class and Tue lab demos
  - No labs on Thu
  - 3 presentations per hour
  - 15 mins per presentation

Final Exam
- Sat, April 16
- 19:00-22:00
- ECS 125

Requirement Engineering Process



- **Elicitation** – collect information about requirements
- **Analysis** – understanding/modeling desired behaviour
- **Specification** – documenting behaviour of proposed software system
- **Validation** – checking whether documented specification accomplishes customer's requirements

Describing Non-Behavioral or Non-Functional Requirements

- **Performance:** 80% of searches will return results in less than two seconds
- **Accuracy:** Will predict cost within 90% of actual cost
- **Portability:** No technology should be used to prevent from moving to Linux
- **Reusability:** DB code should be reusable and exported into a library
- **Maintainability:** Automated test must exist for all components. Over night tests must be run (all tests should take less than 24 hrs to ruin)
- **Interoperability:** All config data stored in XML. Data stored in a SQL DB. No DB triggers. Java
- **Capacity:** System must handle 20 Million Users while maintaining performance objectives!
- **Manageability:** System should support system administrators in troubleshooting problems

Functional Requirements

- Data Requirements
  - Specify the data to be stored in the system
- Functional Requirements: specify
  - Specify what data is to be used for,
  - Specify how data is recorded, computed, transformed, updated, transmitted
- Many data are recorded, updated, and shown through the user interface

Styles for Expressing Functional Requirements

- Each style differs in:
  - Notation — diagrams, plain text, structured text
  - Ease of validation by customer or developer
  - Whether it specifies the environment or the product
  - Whether identifies the functions or gives details on what they do
- We first focus on styles for identifying the necessary functions
- Later, we present techniques for specifying what the functions will do in more detail

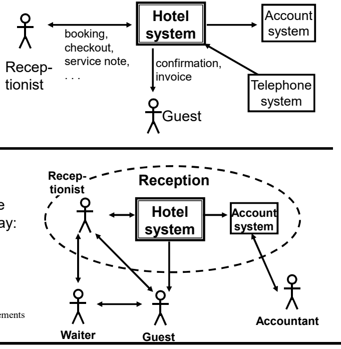
Context Diagrams

- Gives an overview of the required product interfaces
- Good for defining project scope
  - What is in (i.e., product)?
  - What is out (i.e., environment/domain)?
- Shows product as black box surrounded by
  - User groups
  - External systems with which it communicates
- Arrows indicate transfer of data
- Indicate the product domain and surroundings

Context Diagrams

R1:
The product shall have the following interfaces:

R2:
The reception domain communicates with the surroundings in this way:



From: Soren Lauesen: Software Requirements
© Pearson / Addison-Wesley 2002

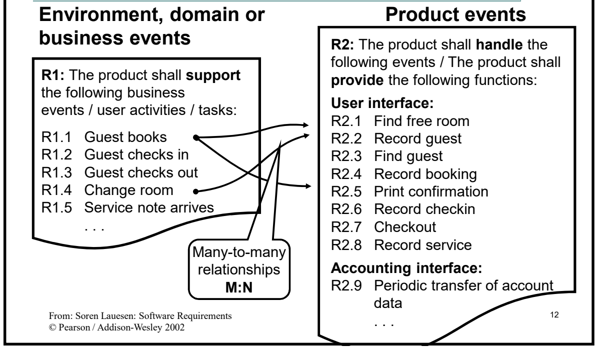
Using Context Diagrams

- Very useful at the beginning and at the end of a project
- Update as project progresses
  - Often out of date after design has progressed significantly
- Defines scope
- Advantages
  - Validation
    - Easy to read by customers who can spot problems
  - Verification
    - Gives an overview of interfaces for developers
    - Offers a high-level checklist

Event / Function Lists

- An event is a request sent to the system from the Environment to perform a function
  - Often used to form use cases
- Environment events are often called business events
  - Guest books room, guest checks in/out
- Each business event leads to an activity
  - Expressed as a use case, task
- Note: you only specify the events not how they are implemented
  - Guest checks in event, but does not specify all the updates in the database

Event List and Function List



Environment, domain or business events

R1: The product shall **support** the following business events / user activities / tasks:

R1.1 Guest books
R1.2 Guest checks in
R1.3 Guest checks out
R1.4 Change room
R1.5 Service note arrives
. . .

Product events

R2: The product shall **handle** the following events / The product shall **provide** the following functions:

User interface:
R2.1 Find free room
R2.2 Record guest
R2.3 Find guest
R2.4 Record booking
R2.5 Print confirmation
R2.6 Record checkin
R2.7 Checkout
R2.8 Record service

Accounting interface:
R2.9 Periodic transfer of account data
. . .

From: Soren Lauesen: Software Requirements
© Pearson / Addison-Wesley 2002

Using Event / Function Lists

- Organize lists
  - According to product interfaces
- Clock/time events
  - For example, to indicate nightly backup or syncing
- Event ➔ Function mapping
  - Functions can be used in multiple tasks
- Specify functions instead of product events
  - Focus on business events instead of product events which are often too low level
  - Gives designer more freedom
- Level of events is critical
  - UI events are usually too low level
  - Interface events are more appropriate

Using Event / Function Lists (cont.)

- Advantage
  - Validation: checklist for customers. Though some events are difficult to check
  - Verification: checklist for developers
- Disadvantage
  - Hard to validate them all
  - Give false sense of security that you gathered all possible events

Feature Requirements

- Most common and straightforward way to write requirements—but not the best way
  - A design or implementation is more than a collection of features (i.e., fulfill or realize business goals)
- Advantage
  - Validation: Uses the customer's language
    - Customers and users can readily articulate features
  - Verification: Easy to check in the final product
    - Is this feature implemented?
- Disadvantage
  - Feature vs. task: Customer dreams up too many features with no business tasks to support them
  - Hard to validate that a particular feature permits the customer to fulfill a particular business goal

Feature Requirements

R1: The product shall be able to record that a room is occupied for repair in a specified period.

R2: The product shall be able to show and print a suggestion for staffing during the next two weeks based on historical room occupation. The supplier shall specify the calculation details.

R3: The product shall be able to run in a mode where rooms are not booked by room number, but only by room type. Actual room allocation is not done until check in.

R4: The product shall be able to print out a sheet with room allocation for each room booked under one stay.

In order to handle group tours with several guests, it is convenient to prepare for arrival by printing out a sheet per guest for the guest to fill in.

From: Soren Lauesen: Software Requirements
© Pearson / Addison-Wesley 2002

What are the Business Goals behind these Feature Requirements?

R1: The product shall be able to record that a room is occupied for repair in a specified period.

R2: The product shall be able to show and print a suggestion for staffing during the next two weeks based on historical room occupation. The supplier shall specify the calculation details.

R3: The product shall be able to run in a mode where rooms are not booked by room number, but only by room type. Actual room allocation is not done until check in.

From: Soren Lauesen: Software Requirements
© Pearson / Addison-Wesley 2002

What are the Business Goals behind these Feature Requirements?

R1: The product shall be able to record that a room is occupied for repair in a specified period.
➔ Optimize when to repair, refurbish, and renovate.

R2: The product shall be able to show and print a suggestion for staffing during the next two weeks based on historical room occupation. The supplier shall specify the calculation details.
➔ Optimize staff hiring over time based on history.

R3: The product shall be able to run in a mode where rooms are not booked by room number, but only by room type. Actual room allocation is not done until check in.
➔ Allow flexibility and optimize for group reservations.

From: Soren Lauesen: Software Requirements
© Pearson / Addison-Wesley 2002

Mock-up User Interfaces, Screens, and Prototypes

- Very common and useful
  - A picture is worth a thousand words
- Mock-up UIs, screens, and prototypes should not be used before a good understanding of the requirements is reached
  - Customers and users can react quite negatively to a mock-up UI
    - Convey the wrong message
    - Not esthetically pleasing
- Use task descriptions instead
  - Much more difficult to disagree with a task than with a UI mock-up
- Establish links between customers and prototype developers and user interface designers

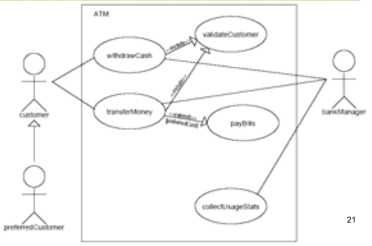
What are Use Cases?

- Use cases (and scenarios) address the problem of:
  - How can I make functional requirements easier to elicit/read/review?
- Other descriptions:
  - They are stories of using a system
  - Requirements in context
  - High-level descriptions of the system's functionality and its environment
  - "Cases of use"
  - Describe how the system meets user goals
  - A way of doing "user-centered analysis"
  - A first cut at the functionality of an application [Rumbaugh]

ATM Use Case

A Use Case describes sequences of actions a system performs that yield an observable result of value to a particular actor:

- Customer Inserts Card
- Customer Withdraws Cash



Use Cases
Selected Definitions

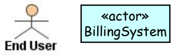
- A *use case* is a story of using the system to fulfill a goal.
  - It models an abstract task (with steps) performed by a user
    - *Rent videos, order blood*

Query blood

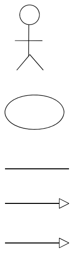
- An *actor* is a person or a program external to the system
  - An actor is an environmental entity that initiates or is otherwise involved with the system.
  - May be a human (*Client*) or a program (*BillingSystem*)
  - A better term for the notion of an actor might be role

Actors

- An actor is someone or something that interacts with the system
- A primary actor is one that initiates a use case
  - Uses cases are (usually) initiated by a primary actor
    - (Exceptions are those that «**extend**» / «**include**» other UCs)
- *Supporting actor* may be invoked by the system
- Off-stage actor, who has an "interest" in the use case
  - Often this concerns NFRs (e.g., government regulatory agency)
- Notation
  - UML stickman to represent a human actor
  - Non-stick figure diagram to represent a non-human actor
    e.g., a box with «**actor**» keyword



Use Case Legend



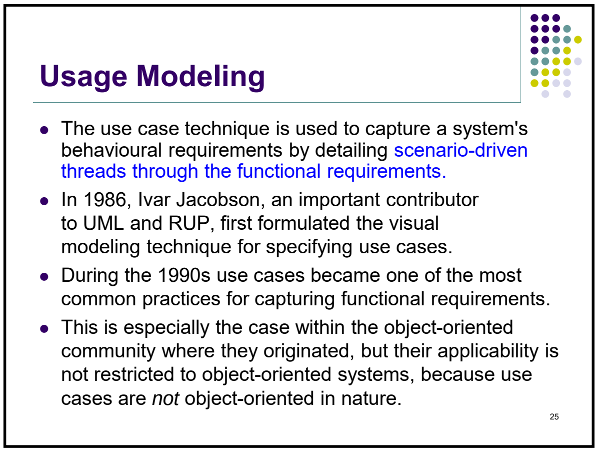
Actor: an entity in the environment that initiates and interacts with the system (i.e., person or program)

Use case: usage of system a set of sequences of actions

Association: relation between actor and use cases

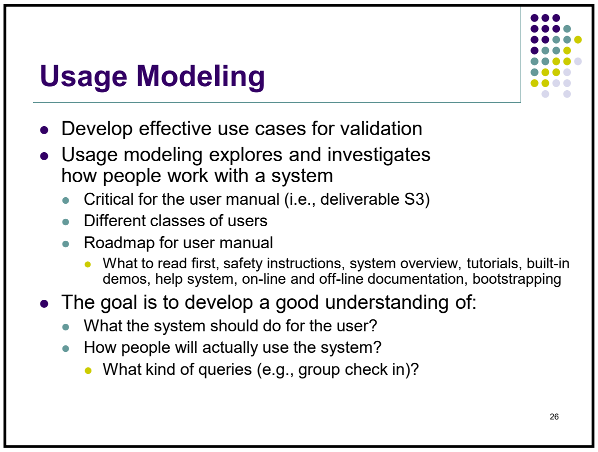
Includes dependency: a sub use case

Extends dependency: a sequence of use cases



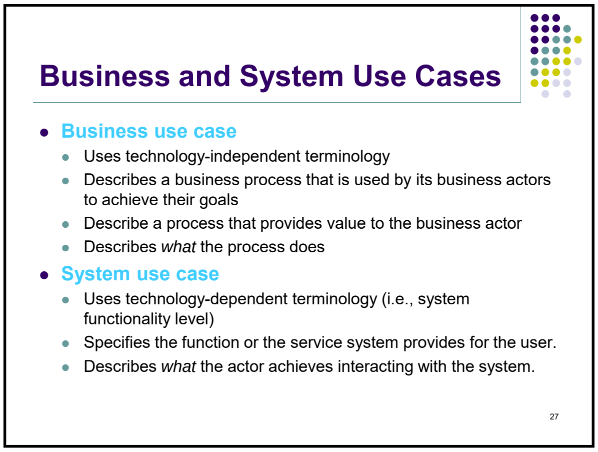
Usage Modeling

- The use case technique is used to capture a system's behavioural requirements by detailing scenario-driven threads through the functional requirements.
- In 1986, Ivar Jacobson, an important contributor to UML and RUP, first formulated the visual modeling technique for specifying use cases.
- During the 1990s use cases became one of the most common practices for capturing functional requirements.
- This is especially the case within the object-oriented community where they originated, but their applicability is not restricted to object-oriented systems, because use cases are *not* object-oriented in nature.



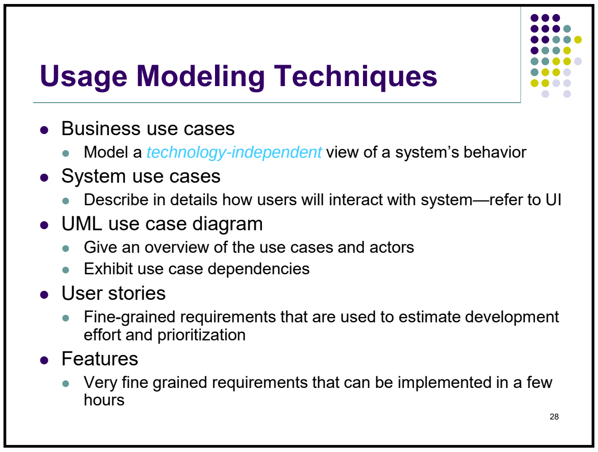
Usage Modeling

- Develop effective use cases for validation
- Usage modeling explores and investigates how people work with a system
  - Critical for the user manual (i.e., deliverable S3)
  - Different classes of users
  - Roadmap for user manual
    - What to read first, safety instructions, system overview, tutorials, built-in demos, help system, on-line and off-line documentation, bootstrapping
- The goal is to develop a good understanding of:
  - What the system should do for the user?
  - How people will actually use the system?
    - What kind of queries (e.g., group check in)?



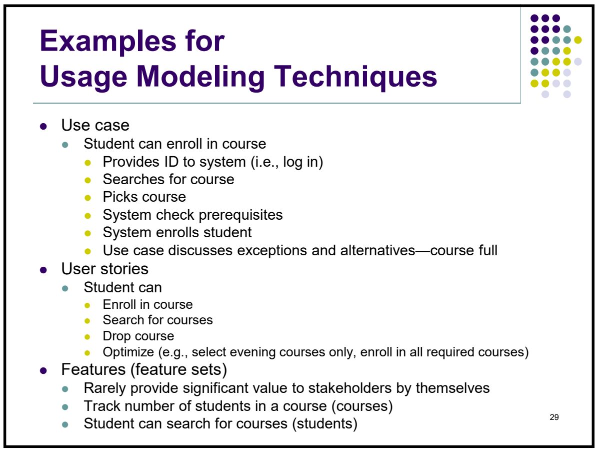
Business and System Use Cases

- **Business use case**
  - Uses technology-independent terminology
  - Describes a business process that is used by its business actors to achieve their goals
  - Describe a process that provides value to the business actor
  - Describes *what* the process does
- **System use case**
  - Uses technology-dependent terminology (i.e., system functionality level)
  - Specifies the function or the service system provides for the user.
  - Describes *what* the actor achieves interacting with the system.



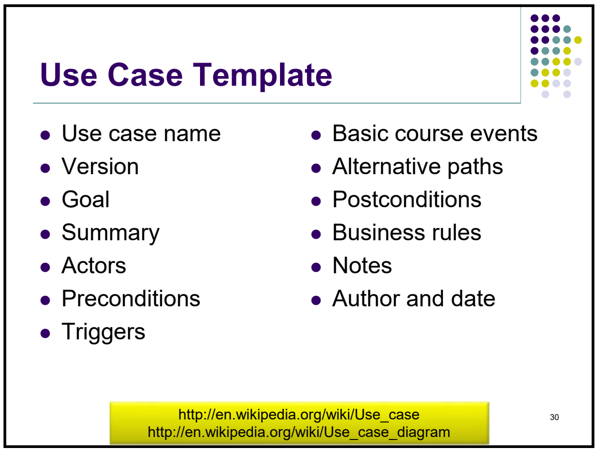
Usage Modeling Techniques

- Business use cases
  - Model a *technology-independent* view of a system's behavior
- System use cases
  - Describe in details how users will interact with system—refer to UI
- UML use case diagram
  - Give an overview of the use cases and actors
  - Exhibit use case dependencies
- User stories
  - Fine-grained requirements that are used to estimate development effort and prioritization
- Features
  - Very fine grained requirements that can be implemented in a few hours



Examples for Usage Modeling Techniques

- Use case
  - Student can enroll in course
    - Provides ID to system (i.e., log in)
    - Searches for course
    - Picks course
    - System check prerequisites
    - System enrolls student
    - Use case discusses exceptions and alternatives—course full
- User stories
  - Student can
    - Enroll in course
    - Search for courses
    - Drop course
    - Optimize (e.g., select evening courses only, enroll in all required courses)
- Features (feature sets)
  - Rarely provide significant value to stakeholders by themselves
  - Track number of students in a course (courses)
  - Student can search for courses (students)



Use Case Template

- Use case name
- Version
- Goal
- Summary
- Actors
- Preconditions
- Triggers
- Basic course events
- Alternative paths
- Postconditions
- Business rules
- Notes
- Author and date

http://en.wikipedia.org/wiki/Use_case
http://en.wikipedia.org/wiki/Use_case_diagram

Object-Oriented Analysis

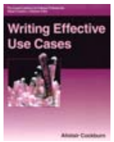
- The key steps of OOA are:

1. Define the *use cases* — including stories of use
   - Formatted text descriptions, maybe UML UC diagrams
2. Define the *domain model* — find the objects, classes
   - UML class diagram
3. Define the *interactions* between domain components
   - UML sequence/communication/collaboration diagrams

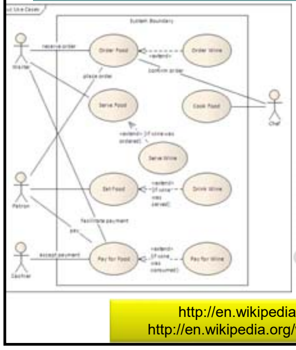
- Define *class diagrams*—is part of object-oriented design (OOD); not covered here

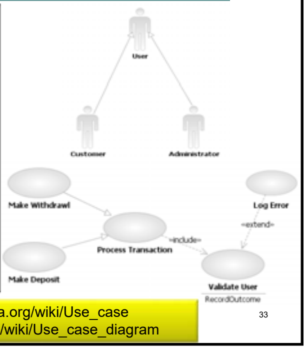
Writing Effective Use Cases

- Based on work of Ivar Jacobson
  - One of the UML/Rational "three amigos"
    - Grady Booch, Jim Rumbaugh and Ivar Jacobson
  - Based on experience at Ericsson building telephony systems
  - His book is old and considered hard to read.
- Use cases aren't inherently OO, but are often used in OOA&D
- Recommended reference
  - Writing Effective Use Cases
    by Alistair Cockburn, Addison-Wesley, 2001
    http://www.usecases.org



UML Use Case Diagram for a Simple Restaurant Model

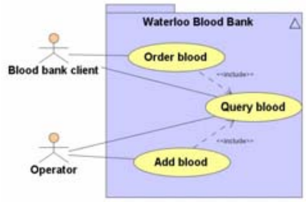




http://en.wikipedia.org/wiki/Use_case
http://en.wikipedia.org/wiki/Use_case_diagram

Blood Bank Use Case

- A blood bank Client logs in.
- The Client requests quantities of various types of blood.
- The blood bank generates a notice to Shipping and records that the blood has been removed from the system.
- An invoice for the order is sent to Billing.



- Basic idea
  - Map out *desired* core system functionality at a coarsely-grained level; consider variations. Explore. Discuss.