Lecture 4: Analysis modeling
- Software abstracts human activities into instructions, procedures, bits, objects, and so on.
- A system model may be created during the analysis activity in requirements development
- Analysts need the ability to create abstract models of systems
  - Analysis models help analysts to understand the functionality of a system
- 2 approaches:
  - Structured Analysis
  - Object Oriented Analysis
- PROS
  - They represent a system from two different perspectives
  - They help analysts to understand the functionality of a system
- CONS
  - Both analysis models are not good for identifying non-functional requirements
  - They may produce too much documentation
  - Too detailed that makes them difficult for the user to understand

Structured analysis based on data-flow analysis (model)
- Data-flow model shows how data flows through transformation processes
  - Model shows data and transformations using a data-flow diagram (DFD)
  - It models the system from a functional perspective
- DFDs are easier to understand by users
- DFDs show end-to-end processing of data
- DFD has three basic elements:
  - External entities (producers and consumers of data)
  - Processes and data flows
- DFDs may represent different levels of abstractions
  - Level 0: entire system is represented by a single process
  - Next levels should represent a system in more detail
- Guidelines for using DFDs
  - Information continuity must be maintained among levels
  - One process at a time should be refined
  - Limit the number of processes per page (up to 7)
  - Evaluate each process to ensure the outputs can be derived from the inputs
  - Label each data flow and process with meaningful names

- Example: DFD’s level-0 for a car reservation system

```
CUSTOMER
Rent-Car
Reservation
and
Billing System
CUSTOMER

CLERK

CAR/CUSTOMER DB

Rent-Car
Reservation
and
Billing System

personal info
car info
tentative bill
final bill
car info
confirmation info

pickup info
drop off info
car damaged
confirmation info
cost
car info
```
Level 1 for previous DFD:

**LEVEL-1**

1.0 Phone Reservation

2.0 in-person reservation

3.0 Biller

Level 2 for Process 1.0 of previous DFD

1.1 Get customer data

1.2 Find Car

2.1 Get person data

2.2 Get A Car

**CAR/CUSTOMER DB**
Object oriented analysis
- Model abstracts real world entities as software objects.
  - A book, a bicycle, a student, a payment, and a computer are examples of objects.
- Processing is associated with these objects
- An OO software simulates the real world objects and their processes by these software objects
- OO is not new. It is more than 30 years old
- It started with the introduction of OO programming (OOP), followed by OO analysis and design.
- Some OOP supports procedural as well as OO programming (e.g., Ada and C++)
  - Some OOP are pure (e.g., Smalltalk)
- Software must be modeled by capturing those aspects that are relevant for a particular task or problem
- OOA and OOD use this model to simplify the complexity of the software and be able to deal with changes

- OOA consists of:
  - Defining use cases
  - Identifying object/classes and their attributes
  - Determine how classes/objects are related
  - Building a behavioral (interaction) model
  - Iterating on these previous steps

- Use cases and class diagrams and behavioral model are described with UML notation

Unified Modeling Language (UML)
- It is a tool that helps in OOA tasks.
  - UML model is a "blueprint" of software that let users and software developers communicate
- UML is an industry standard visual language for OOA/D
UML has nine kinds of modeling diagrams:
  - Use cases (use case diagrams)
  - The static structure of a system (class and object diagrams)
  - The behavior of objects in a system (sequence, collaboration, activity, and state diagrams)
  - Physical implementation of a system (components and deployment diagrams)

Use case diagrams
- A use case describes what a system does from the standpoint of a user (anything external).
- A use case diagram is easy to understand by a user of a system
  - It allows the communication between a system developer and its users.
- A use case is represented by an oval, representing a task or function of a system.
- An external entity (e.g., a person) interacting with the use case is called an actor,
  - It is usually shown as a stick person.
- A use case diagram may show more than one use case.
- The box enclosing the use cases indicate the system boundary.
  - This boundary is labeled with the name of the system.
- Use cases are used to identify the user requirements for a particular software product.
- A use case is a task or function that an actor is expecting the software to perform.
- Use cases are related to scenarios.
  - A scenario is a written interaction between a user and a system.
  - There is a normal scenario for a task when everything works as expected.
  - Several other scenarios associated with a normal scenario respond to unusual situations of the interaction between a user and a system.
Example of scenarios for the use case “Borrow copy of book.”

**Normal scenario:** Book borrower Joe Doe needs to borrow a copy of the book “Programming Languages.  He finds a copy of the book and proceeds to check out the book.  The system verifies that Joe has not borrowed the maximum number of books, and the system let him borrows the book.

**Unusual scenario 1:** Book borrower Joe Doe needs to borrow a copy of the book “Programming Languages.  The library does not have a copy of the book.

**Unusual scenario 2:** Book borrower Joe Doe needs to borrow a copy of the book “Programming Languages.  He finds a copy of the book and proceeds to check out the book.  The system finds out that Joe has borrowed the maximum number of books allowed by the library. The system refuses to loan the book to Joe.

These three scenarios are possible instances of the use case. The outcome of any of these scenarios differs from the others.

**Relationship between use cases**
- Two types of relationship between use cases have been identified: <<include>> and <<extend>>.
- The <<include>> relationship (<<uses>> in Visio use case diagram) is used when a common behavior, feature, or subtask from two or more use cases are abstracted in a use case.
  - That is, a use case “includes” the functionality of another use case.
  - For instance, in the use cases “Borrow copy of book” and “Extend loan”, both use cases require the system to check whether another borrower has reserved the book. The <<include>> relationship is shown in the following diagram.
The <<extend>> relationship is used for a use case that have different scenarios, with a main case (normal scenario) and one or more special (unusual scenarios) cases. An arrow from use case A to use case B (main case) indicates that A is a special case of B. Keep in mind that the direction of the arrow is opposite to the direction of the <<include>> relationship. Example of an <<extend>> relationship is shown in the diagram below. Here, the “Refuse loan” is a special use case of the main case “Borrow copy of book”. “Refuse loan” represents the unusual scenario when a borrower has already borrowed the maximum number of books allowed for him or her.

A use case analysis involves the following steps:
1. Determine the actors (people, systems, etc.) that interact with the software produced to be implemented. Some actors (primary) initiate the interaction with the system, others (secondary) respond to system requests.
2. Identify use cases. Use cases are functionality of a system satisfying actor’s needs.
3. Determine the goal of the use case.
4. Identify pre- and post conditions. That is, identify the state of the system at the start and at the end of a use case.
5. Specify the main scenario that would result in a normal operation for the use case, and the various unusual or alternate scenarios that can happen during the process of a use case.
6. Draw the use case diagram.
7. Identify <<include>> and <<extend>> relationships and modify the use case diagram with these identified relationships.

- The minimum documentation of a use case consists of:
  - A use case name
  - Brief description of the use case
  - Actors involved
  - Preconditions necessary for the use case to start
  - Detailed description of the normal flow of events (main scenario) and alternate flows
  - Postconditions that define the state of the system after the use case ends

The following page shows a user case template that you could use to perform analysis.

Use cases could be used for structured analysis.
### Use Case Template

<table>
<thead>
<tr>
<th>Date:</th>
<th>Date created:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Updated by:</th>
<th>Created by:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use case ID:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Priority: [1=highest, ..., 5]</th>
<th>Frequency of use: [evt per timescale]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actors: [List primary and secondary actors]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description: [Reason for use case (the need) and outcome] [Or sequence of actions (high-level procedures) and outcome]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trigger: [Event that initiates the use case]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- E.g. external business event;
- E.g. system event;
- E.g. first step in normal flow.

<table>
<thead>
<tr>
<th>Includes: [Any addendum use cases, like a library of common functionality]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preconditions: [Any requirements before the use case can begin]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Postconditions: [The change in system state, caused by the use case]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Normal flow: [Normal user actions and system responses]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alternative flows: [Conditions that would affect the normal sequence; and the corresponding flow]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions: [Anticipated error conditions and how system should respond] [How to respond if the use case fails for some unanticipated reason]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- E.g. Durable state changing system:
  - Roll back state;
  - Completed correctly;
  - Partially completed with a known state;
  - Left in an undetermined state.

<table>
<thead>
<tr>
<th>Business rules: [Relevant business rules]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special requirements: [Quality attributes]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assumptions: [Reasons why this use case is important]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unresolved issues: [For each issue include description, assignee, due date, and result]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comments:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

### Class diagrams

- These diagrams capture and document the static structure of a software product.
- The structure shows which classes are parts of the system and how they are related.
- It does not show how they interact to produce some results.
- A class describes a collection of objects with the same properties (attributes) and behaviors (operations).
- In UML, a class is represented by a rectangle that includes three sections:
  - the topmost section has the name of the class;
  - the middle section lists its attributes; and
  - the bottom area lists its operations.
- In initial class diagrams, the bottom two sections are omitted.
- Instances of a class are called **objects**. The diagram below represents the Book class.
- It has two operations:
  - copiesOnShelf, takes no argument and returns an integer indicating the number of copies
  - borrow, takes an argument, the copy borrowed, and returns no result.
- These operations represent the messages that an instance of a book understands.

- Class diagram shows relationships (called associations) between object classes
  - They are shown as lines connecting classes with some label
  - It shows that an object of a class needs to interact with an object of another class
  - Each side shows the number of objects of a class that may be associated with objects of the other class
    - Called multiplicity of the form \( n..m \), where \( n \) and \( m \) indicate min, max, respectively.
    - They can be \( 1..1, 0..1, 0..*, 1..*, *..* \) (\( 1..1 \) could be written as \( 1 \))

- An instance of an association is called a link.
- Associations correspond to some verbs in the requirements document.
- Several special types of associations can be defined between classes
  - Generalization: representing inheritance
    - Shown with a small triangle pointing from subclasses to a parent class

- Aggregation: representing part-of relationships
  - Shown with an open diamond shape pointing to the “whole” class

- Composition is a special case of Aggregation where each part of the aggregate belongs to it.
- Shown with an filled diamond shape pointing to the “whole” class

![Diagram](image)

**Object identification**
- Objects include physical entities (e.g., book), or concepts (e.g., a purchase order).
- One technique identify objects by examining the nouns in a problem definition, SRS, or scenarios
- Then you eliminate those terms or classes that satisfy the following criteria:
  - **Redundant**: Two or more classes representing the same or similar information. For example, user, customer, and client may represent the same entity. You must keep the most descriptive name for the application.
  - **Irrelevant**: Class that has nothing to do with the problem must be removed. Example, library may not be a good class for the library system described before.
  - **Vague**: A class that is not well defined or too broad in scope.
  - **Attributes**: Nouns that are properties of objects. Define a class for an attribute if it needs to be modeled independently. For example, a room could be a class or attribute depending on the application.
  - **Operations**: Nouns describing operations that are applied to objects. A telephone call could be an operation or a class depending whether the call has features of its own that requires to be modeled as a class.
  - **Roles**: A class should not represent the role of an object of the class if that role is already covered by a more descriptive class. For instance, Owner is a role of a customer of a car service. Customer would be the class, who can play the role of an owner, or driver, or lessee.

- Example: **Good and bad objects**
  Design an application to support a computerized banking network including both human and ATMs to be shared by several banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank’s computers. Human cashiers enter account and transaction data. ATMs communicate with a central computer which clears transactions with the appropriate banks. An ATM accepts a card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts.

  Good objects: ATM, bank, account, transaction, station, cashier, card.

  Other objects may be added later (objects required for control, gathering data, computation process, interfaces, collection, and so on)

**Object interaction modeling**
- An interaction model shows how objects collaborate in some behavior
- Usually it captures the behavior of a single use case
- There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams
  - Both show how objects pass messages to other objects
- Sequence diagrams in UML
  - An object is shown as a box at the top
  - A vertical line (called lifeline) represents an object’s life during the interaction
    - An activation box in the lifeline shows when an object is active
- Each message is represented by an arrow between the lifelines of two objects
- It indicates who sends the message and when
- Each line is labeled with the message name
- A condition (in square brackets) indicates when a message is sent

**Example**: Class diagram and sequence diagram for an application that allows a sales representative for a bookstore to process book orders made by clients. This example was taken from lecture materials prepared by Professor A. Norton, for the MACS 306 (Software Engineering) course. Course was taught on spring semester 2004 at Colorado School of Mines.

**Class Diagram**:
The Book Order Application requires two classes, a SessionManager and a BookOrder. The SessionManager class handles the interaction with the InventoryManager and the Sales Rep. SessionManager uses an instance of BookOrder to store information of a Client's order. The class diagram is shown below. It includes the classes used by the implementation of the InventoryManager (InventoryManager and bookRecord classes).

The sequence diagram for this exercise is shown below:
InventoryManager

SessionManager

getTitle()

getISBN()

[!inCatalog] errorReport()

getNumInStock()

getPrice()

[!numOrdered<=0] endMessage()

getQuantity()

reportBookStatus()

getCustomerInfo()

new()

BookOrder

fillInOrderInfo()

[numToBackOrder>0] backOrder()

ship()

displayBill()

delete()

terminate()