Use Case Modeling Techniques

From Universal Modeling Language (UML)

Use Cases

- **Interaction** between a **user** and a **system**
- A **complete and meaningful** use
- Focus on **value** – how the system will be used to **satisfy** a specific ***user goal***
- **Observable and testable functionality -** "black box" view of the system
- The **first system functional decomposition**
- All use cases = **{all things the system must do}**
- **Understand** the **big picture**

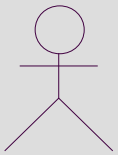
Use-Case Modeling Phases

Three phases of requirements analysis:

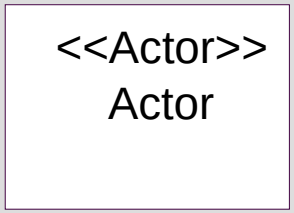
1. Model the **user roles** – detailed actor descriptions
   - Identify **user goals** for system interaction
2. Model (specify) requirements as **use cases**
   - Use case **diagrams** – for context and reference
   - Use case **descriptions**
3. Model **use-case realizations**
   - An interaction of objects that realize the requirements
   - Class diagrams and object interaction diagrams
   - (Also known as robustness analysis, use case analysis, task modeling, or scripting)

UML Use-Case Modeling

- An actor represents **anything that interacts with the system**

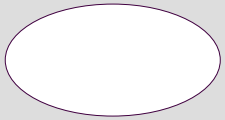


Actor



People, external systems, devices

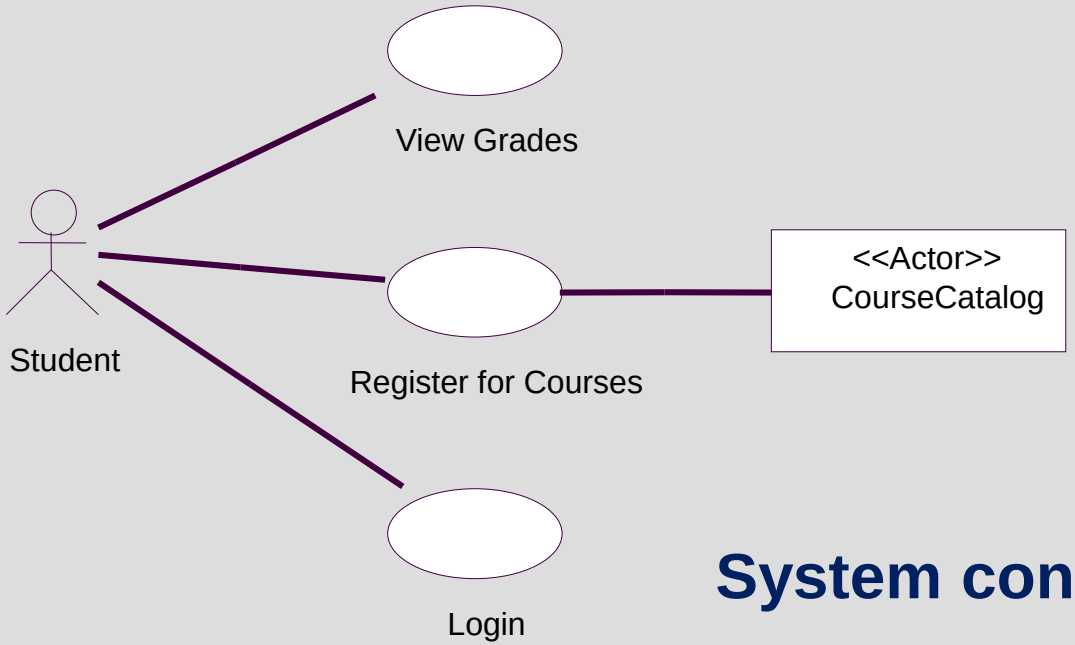
- A use case is a set of system actions that yields an **observable result of value** to a particular actor



UseCase

Use-Case Diagram

- A use-case diagram shows relationships between actors and use cases
  - Relationships: “communicates with” (exchanges data, signals, events)



System context

Use Case Descriptions

- For each use case describe **functional steps** in sufficient detail to …
  - Enable (or represent) **requirement specification**
  - Begin early **design** work
  - Achieve **stakeholder and user understanding and approval**
- The details …
  - Name and description
  - Actors
  - Primary **flow of events** (as related stories)
  - Secondary **alternative** and/or **exception flow of events**
  - System **preconditions**
  - System **post conditions**
  - **Supplemental** information – **non-functional** requirements

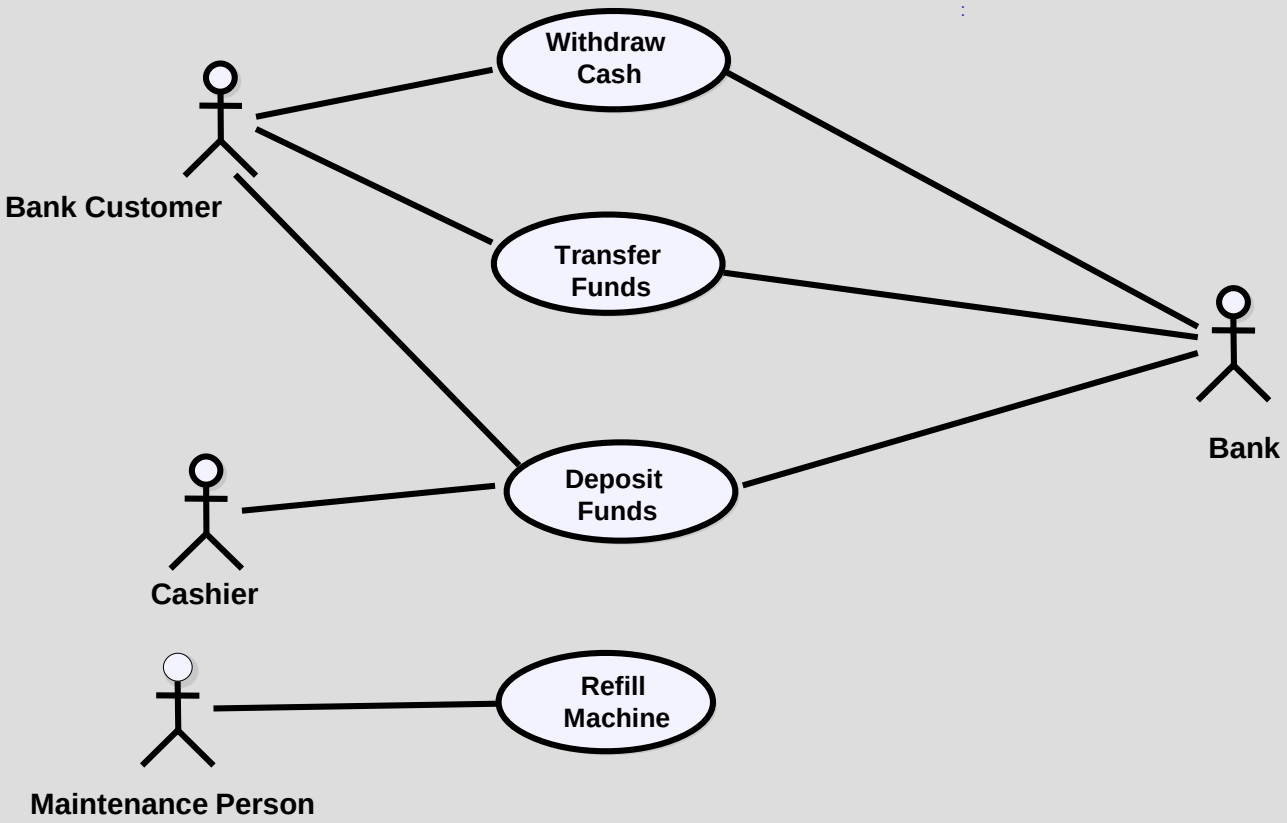
“Why Use Cases at All?”

- **A good compromise – use cases are semi-formal, structured, but understandable stories** (people like stories)
- Use cases **add value** to analysis
  - At first as a succinct outline of mainline features and capabilities (get your head around the functionality)
  - Later a basis for innovation, extension, revision of requirements
- **Address exceptions** – a large source of system complexity
- **Start functional decomposition** that transitions to requirement specifications and early design
- Good **basis for pursuing related project information** – estimates, plans, user interface design, software design, testing

But Use Cases Have Limitations

- **Too much detail** – can be hard to work with
- Developers **need use case supplemental requirements** to design
  - **Ancillary functionality** such as system administration
  - **Non functional** quality requirements and business rules
- **Functional decomposition guidance for design has limits**
- **May not be as effective for non-user interactive systems**
  - Concurrent applications, batch processing, data warehousing, computational intensive, etc.

Use Case Example - ATM



ATM Model: Withdraw Cash Use Case

1 **Brief Description**

This use case describes how the Bank Customer uses the ATM to withdraw money his/her bank account.

2 **Actors**

2.1 Bank Customer

2.2 Bank

3 **Preconditions**

There is an active network connection to the Bank.

The ATM has cash available.

4 **Basic Flow of Events**

1. The use case begins when Bank Customer inserts their Bank Card.
2. Use Case: Validate User is performed.
3. The ATM displays the different alternatives that are available on this unit. [See Supporting Requirement SR-xxx for list of alternatives]. In this case the Bank Customer always selects “Withdraw Cash”.
4. The ATM prompts for an account. See Supporting Requirement SR-yyy for account types that shall be supported.
5. The Bank Customer selects an account.
6. The ATM prompts for an amount.
7. The Bank Customer enters an amount.
8. Card ID, PIN, amount and account is sent to Bank as a transaction. The Bank Consortium replies with a go/no go reply telling if the transaction is ok.
9. Then money is dispensed
10. The Bank Card is returned.
11. The receipt is printed
12. The use case ends successfully

ATM Model: Withdraw Cash Use Case (2)

5 **Alternative Flows**

5.1 Invalid User

If in step 2 of the basic flow Bank Customer the use case: Validate User does not complete successfully, then

1. the use case ends with a failure condition

5.2 Wrong account

If in step 8 of the basic flow the account selected by the Bank Customer is not associated with this bank card, then

1. Tthe ATM shall display the message “Invalid Account – please try again”
2. The use case resumes at step 4

.

.

.

7 **Post-conditions**

7.1 Successful Completion

The user has received their cash and the internal logs have been updated.

7.2 Failure Condition

The logs have been updated accordingly.

Developing the Use Case Model

- Steps
  - Find Actors
  - Find Use Cases
  - Describe How Actors and Use Cases Interact
  - Present the Use-Case Model in Use-Case Diagrams
  - Package Actors and Use Cases
  - Develop a Survey of the Use-Case Model
  - Evaluate Your Results

Step: Find Actors

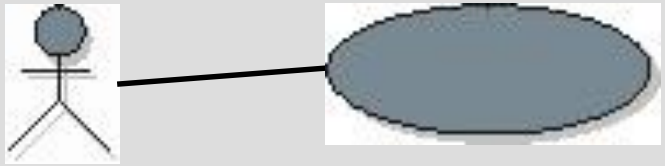
- Actor: Define a coherent **set of user roles** for system interaction
  - An **individual** or an **external system**
  - **Primary users** for main functions
  - **Secondary users** for ancillary functions
- **Name** the actor to clearly describe the actor's role
- Briefly describe the actor
  - **Responsibilities and goals** for what the system needs to accomplish
  - **Capabilities** (skills, environment, etc.) relevant to the system

Step: Find Use Cases

- For each actor (human and not)
  - What are the **primary tasks** the actor wants to perform?
    - E.g., create, retrieve, update, delete data
  - What are the **secondary tasks** the actor wants to perform?
    - E.g., system maintenance tasks
  - What are the actor **trigger events** to initiate action between actors and the system?
- **Logically coherent tasks are use case candidates**

Step: Find Use Cases (cont)

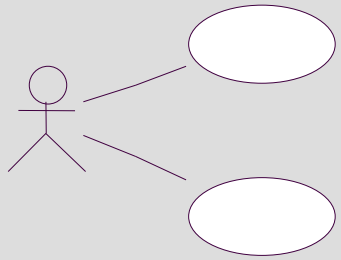
- **Name** the use case: a **verb phrase** that represents the user's goal
- Briefly **describe the purpose** of the use case
- Outline the **basic and alternative flow of events** – details follow
- Collect **additional (non-functional) requirements as supplementary specifications**
- Iterate to add, remove, combine, and divide the use cases



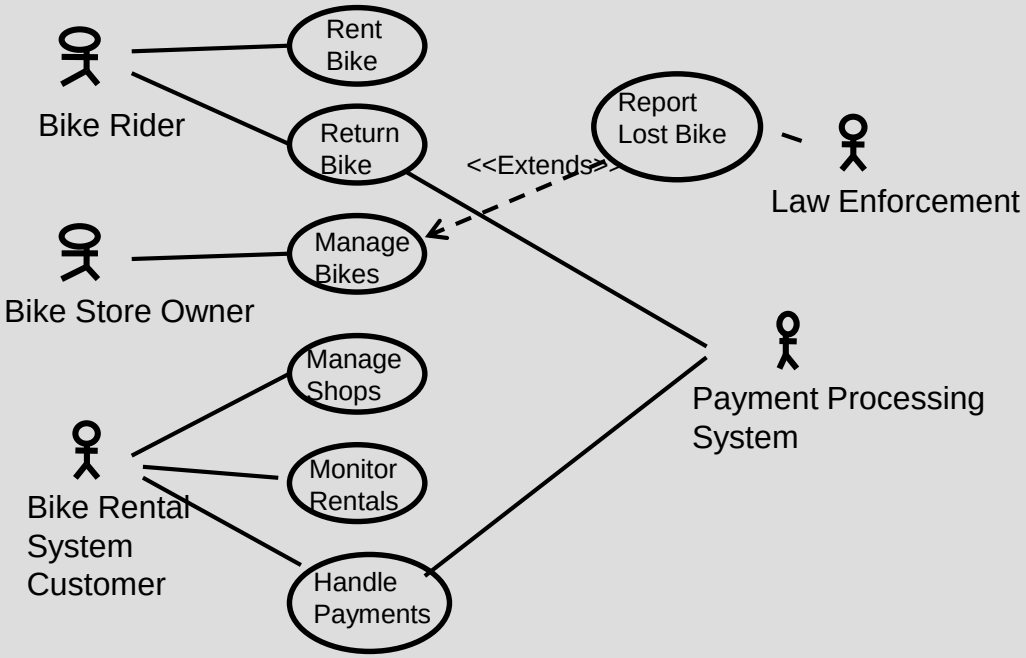
Step: Describe How Actors and Use Cases Interact

- Establish which actors will interact with each use case
- For each actor-and-use-case pair
  - Define, at the most, one **communicates-association**
    - The flow of events and data to support the tasks
  - The communicates-association **navigation is bidirectional**
  - Briefly describe each communicates-association

Step: Present the Use-Case Model in Use Case Diagrams



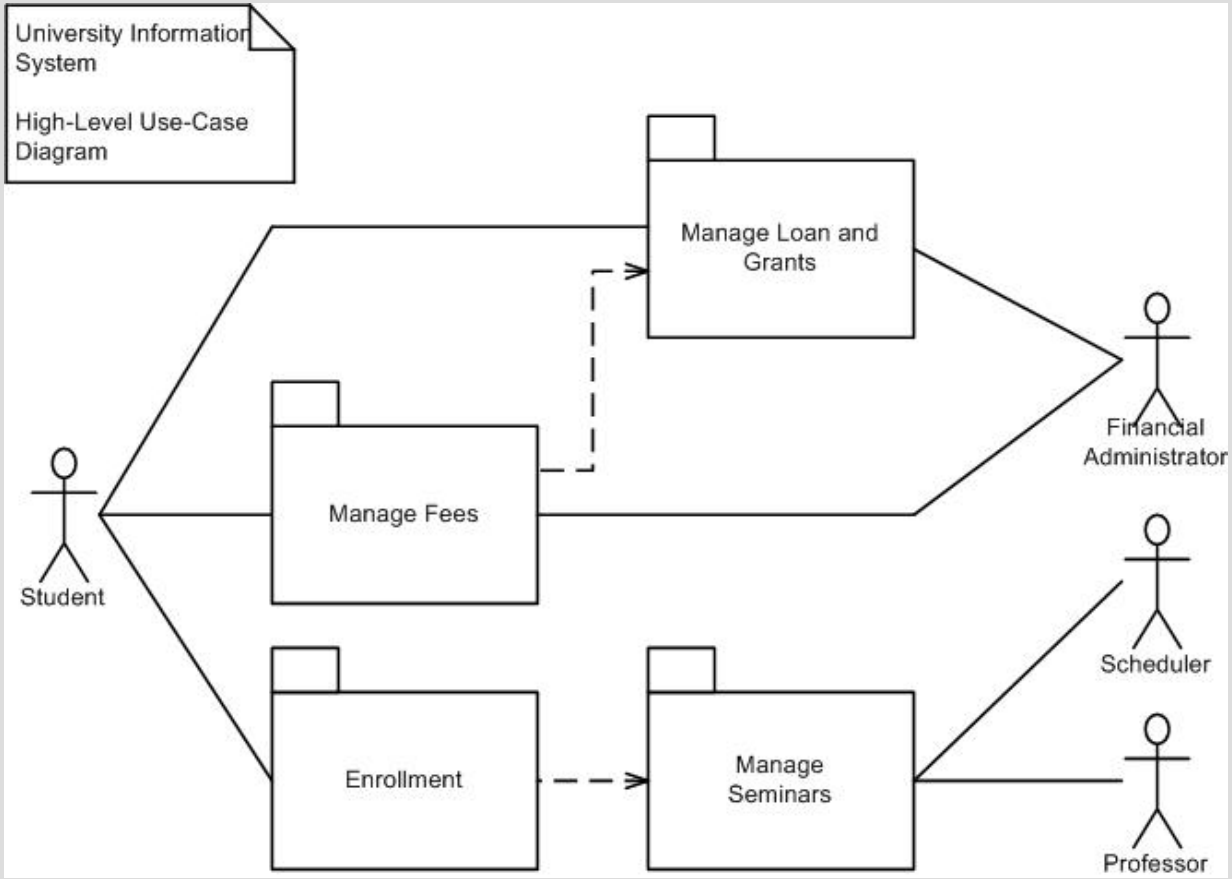
- Illustrate the relationships among use cases and actors, as well as among related use cases in diagrams



Step: Package Use Cases and Actors

- If the number of actors or use cases becomes too great, divide them into **use-case packages**
  - A collection of **functionally related use cases and actors** – think **functional sub-system**
  - Easier to understand and maintain the model
  - Future architectural implications
- Packaging alternatives:
  - Actor to use case relationships; 1:N or N:1
  - Use case relationships:
    - Manage **common information**
    - **Work or data flow sequences**
    - Most important
    - Hierarchies (but **breath before depth**)
    - Other criteria such as release packaging

Use Case Package Example



There should be a use case diagram for each package

Step: Develop a Survey of the Use-Case Model

- Write a **descriptive summary** (an abstract)
  - The **primary actors** and their roles
  - The **primary use cases** of the system (the reason the system is built)
  - Primary system **workflow sequences**
  - **Package/sub-package hierarchies**
  - **System boundaries** – What is in the system and external dependencies
  - The **system's environment**, for example, target platforms and existing software
  - **Non-functional requirements** not handled by the use-case model – quality attributes

Step: Evaluate Use-Case Model

- Are all **essential actors and use cases** identified?
- Identify **unnecessary actors or use cases**
  - Provide little or no value
  - Use cases that should be combined for greater value
- The **flow** of actor-use case interaction is reasonably correct, complete, and understandable at this stage
- The survey description of the use-case model makes it understandable

Detail a Use Case

Detail Each Use Case

- Describe **functional steps** in sufficient detail to …
  - Enable **requirement specification**
  - Early **design** work to begin
  - Achieve **stakeholder and user understanding and approval**
- Steps
  - Structure and detail the **flow of events**
  - Describe **preconditions**
  - Describe **post conditions**
  - Describe any **special requirements** of the use case
  - Describe any communication protocols [optional]
  - Evaluate the results

Basic Use Case Template

- Unique identifier
- [Metadata – e.g., author, priority, etc.]
- Name
- Actors
- Description
- Preconditions
- Post conditions
- Primary scenario of events
- Secondary (alternative and exceptional scenarios)
- Special requirements
- [Extension points]

Style Considerations

- **Choose** the **template** structure and **language style** ahead of time
  - Many template styles – be consistent and complete
- Describe the **flow of events**, not just the use case's functionality or purpose.
- **Self containment** - avoid references to other actors and use cases
- **Do not describe** the details of the **user interface**
- **Do not discuss** implementation **technology**

Style Considerations (cont)

- Express in **natural language**, avoid code-like constructs
  - **Simple**, active **action steps**
  - Concise and explicit well written sentences
  - **Minimalist**, essential detail
- Make it **understandable for customers, users, and developers**
  - Use **domain terminology** , not technology or methodology terminology; add a glossary
  - Avoid the use of methodology-specific terminology, such as “use case”, “actor”, and “signal”
  - Avoid vague terminology such as "for example", "etc. " and “the information“

Style Considerations (cont)

- **Breadth-Before-Depth** – work on an **overview** of use cases **first** and then progressively add detail
- **Quitting-Time** (when are we done?) – use cases are **complete (versus goals) and achieve review approval**
- **Find the right user goal level**; rule of thumb 5-10 steps; goal granularity
- Find the **right balance for the number of use cases**
  - Rule of thumb - no more than **two dozen** use cases
  - Partition larger systems into packages

Step: Detail the Flow of Events of a Use Case

- How and when the use case starts – the **trigger event**
  - Trigger – the actor, the system, time
- How and when the use case **terminates – success and failure**
- **How the actors and the system interact**
  - Describe what the use case does for each actor action:

  "When the user does …, the system does …."
  - Describe what the use case does for each system action:

  "When the system does …, the user does …."
- **Data exchange** between actors and use cases
- **Information storage and retrieval**

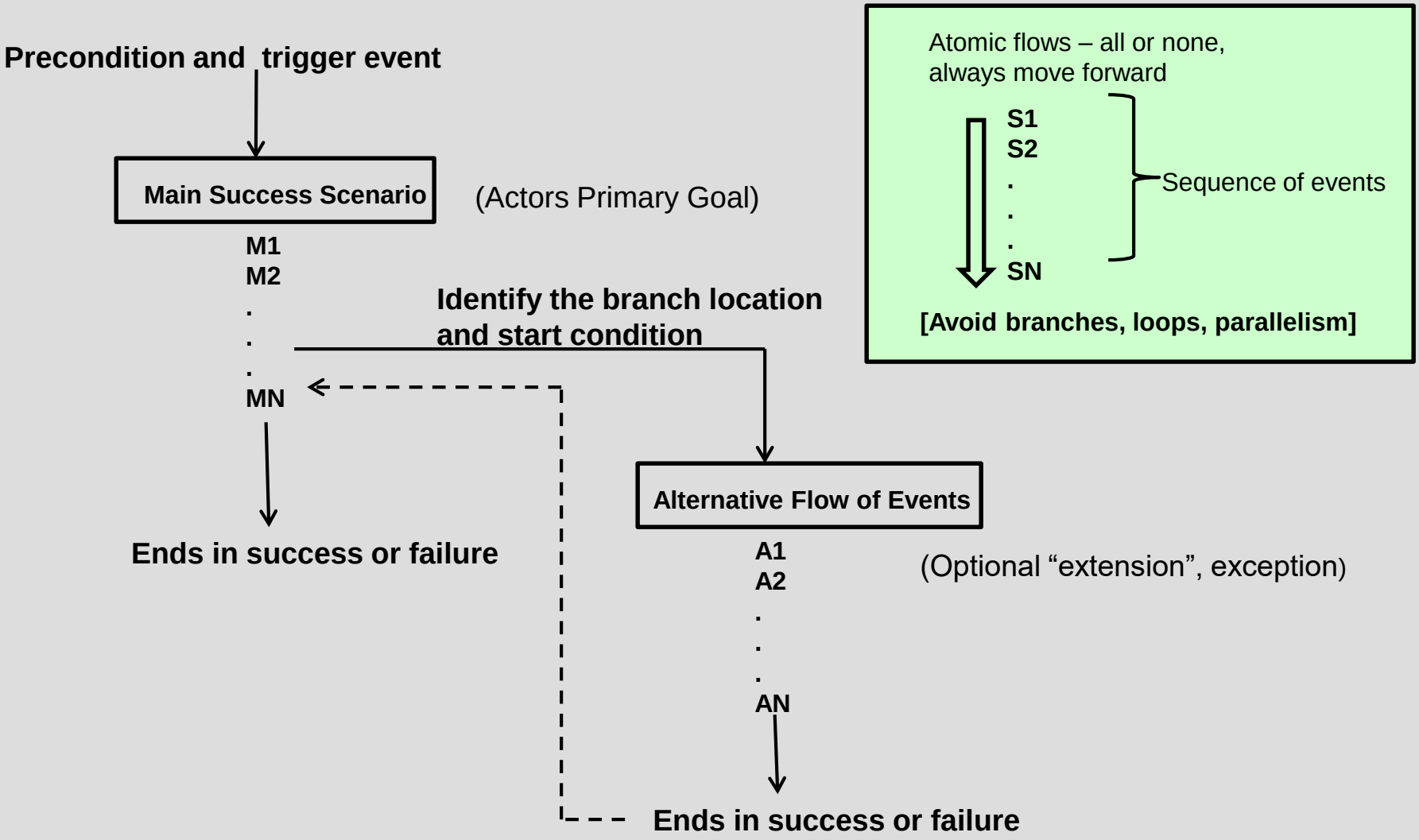
Step: Detail the Flow of Events of a Use Case
(Continued)

- Structure the flow of events: **main scenario plus alternative flows**
- Alternatives - **branches** in behavior from the main scenario **due to some condition (extensions)**
  - **Alternative flows** due to **user or system action**
  - **Exception conditions**
  - Where the most interesting system requirements are found

Step: Detail the Flow of Events of a Use Case
(Continued)

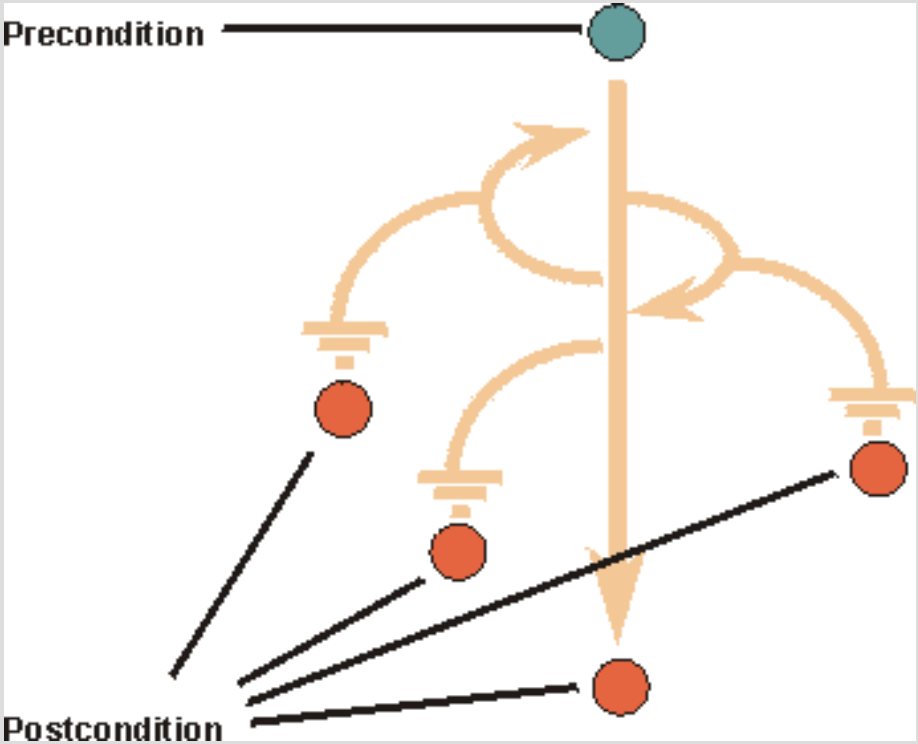
- Think of an **alternative workflow as its own stripped down use case**
  - Starting condition
  - Sequence of action steps
  - Completion state that ends in goal success or failure
  - **Avoid duplicating main flow steps in alternative flows**
  - Reference main flow step branch
  - Identify main flow return step or termination

Structure the Flow of Events of the Use Case



Preconditions and Post Conditions

- A **precondition** is the state of the system required **before** the use case can be started
- A **post condition** is the state the system can be in **after** the use case has ended
- Identifying post conditions can help describe use cases themselves
  - First define what the use case is supposed to achieve, the post condition
  - Then describe how to reach this condition (the flow of events needed)



Step: Describe Preconditions of the Use Case

- **System state** required to start the use case
  - It is **not the trigger event** that starts the use case
  - Avoid describing prior incidental activities that may have happened
- Pre- or post condition states should be **observable by the user**
- A precondition **applies to the entire use case**, not only one sub flow
  - (Although you can define preconditions and post conditions at the sub flow level)

Step: Describe Post Conditions of the Use Case

- Possible **system states** at the end of the use case
- Post conditions can also state **system actions** performed at the end of the use case
- Post conditions should be **true regardless** of what occurred in the use case
  - Cover use case exceptions in the post condition description

Sequencing Use Cases with Pre/Post Conditions

- Best practice says **you should not use pre- and post conditions to create a dependency sequence of use cases**
- The sequentially dependent use cases should be combined into a single use case
- Possible exceptions:
  - When a common “sub-use-case” is factored out
    - For example, a “Log In” use case
  - Complexity – the combined use case is too large but consider sub flows

Step: Describe the Special Requirements of the Use Case

- Related requirements not considered in the use case scenarios
  - Described in the Survey Description of the use case
  - Such requirements are likely to be **nonfunctional quality requirements or design constraints**
  - (Also called supplemental requirements)

Step: Describe Communication Protocols [Optional]

- Describe the “applications layer” communication protocol **if the actor is another system or external hardware**
  - Specify if some existing protocol (perhaps a standardized one) is to be used
  - If the protocol is new, it will be fully described in design
- The protocol may be expressed as the message interaction through an established application programming interface (API)

Structure the Use Case Model

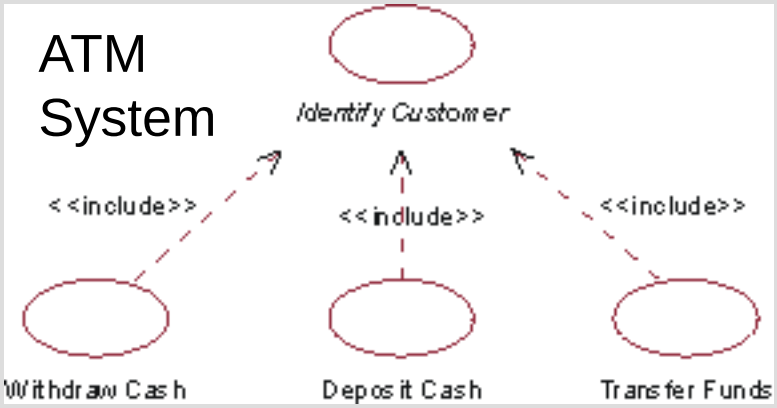
Decompose the model to enhance understandability and maintainability

Structuring the Use-Case Model

- **Factor** use case behavior into more **abstract use cases**
  - Identify common, optional, exceptional, or deferred out of scope scenarios
- Possible relationships
  - **Include-Relationships** Between Use Cases
  - **Extend-Relationships** Between Use Cases
  - **Generalizations** Between Use Cases
  - Generalizations Between Actors
- Best performed **after** you have made **your first attempts** at a use-case model

Include-Relationships

The include-relationship exists between a base use case and subordinates



Use the include-relationship to **factor out behavior**:

- **Subordinate** to the primary purpose of the base use case
- **Common** for two or more use cases
- (~ library module in programming)

Include-Relationships

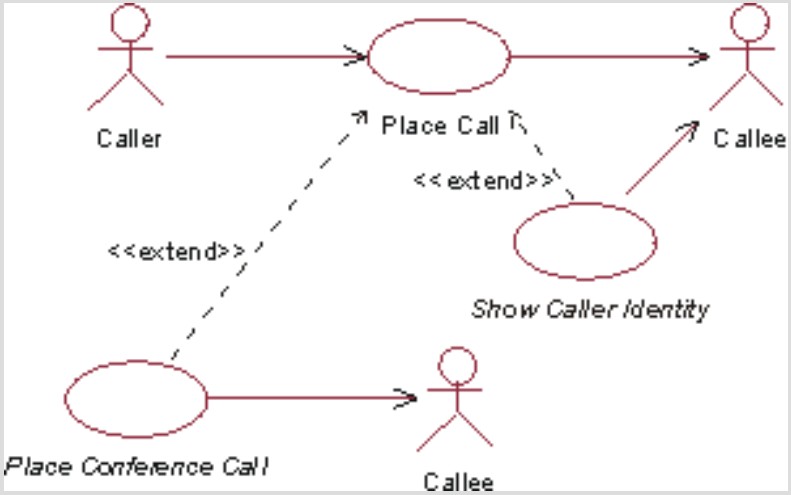
- The **base use case owns the relationship** to the inclusion use case
  - **It depends on the behavior of the included use case**
  - Refer to the included use case in the step where the inclusion is inserted
  - Only the base use case knows about the inclusion use case
  - No inclusion use case knows what base use cases include it
- An inclusion use case has a **communication-association to an actor** only if its **behavior explicitly involves interaction with an actor**

Extend-Relationships

Extend-relationships **branch behavior** from the use case main scenario **due to some condition into sub use cases**

Use case extensions made into **separate use cases**

- Used in several places in the base use case
- Reduce base use case complexity to improve readability

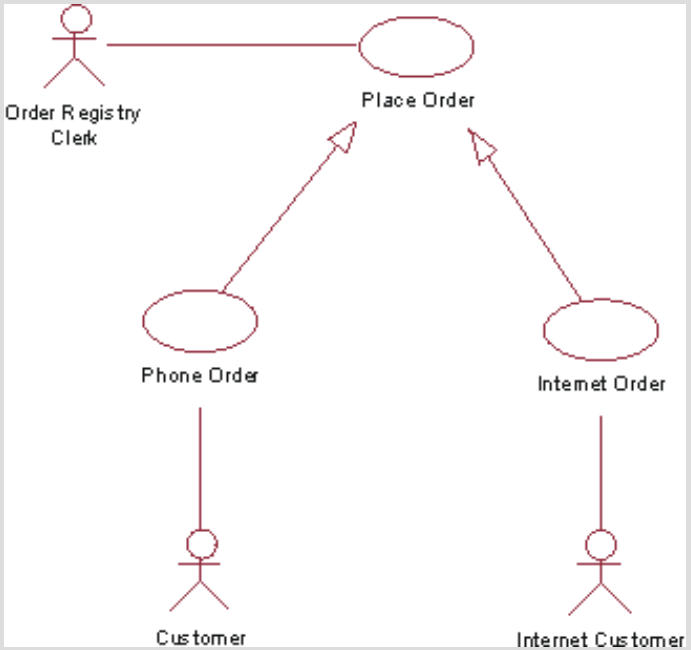


Extend-Relationships

- Use extensions to:
  - Show use case behavior that is **optional or conditional** from the primary use case purpose
    - Primary flow is “interrupted”
    - Alternative flows due to user or system action
    - Flows executed only under certain (sometimes exceptional) conditions
  - Allow new extending behavior to be added over time without impacting the base use case
- The **extension is conditional**, its **execution is dependent on base use case flow**
  - Only the base and extending use cases knows of the relationship between the two use cases

Generalizations

The generalization relationship **generalizes** the **common behavior** of two or more use cases to create a new parent use case



Generalizations

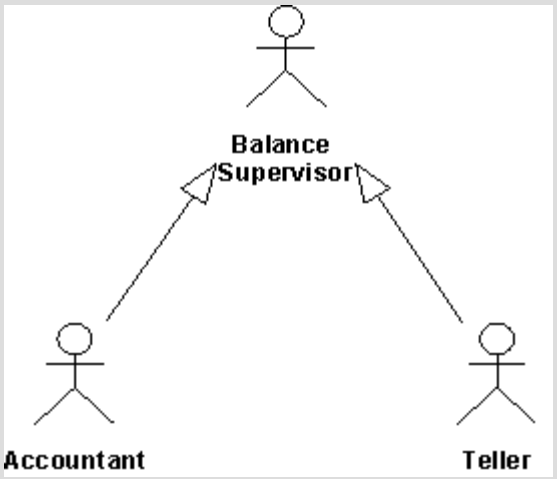
- Use generalization when:
  - Two or more use cases have **commonalities in behavior, structure, and purpose**
  - **Common behavior leads to specialized behavior** in flow steps
    - These could be modeled as extensions
- Describe the **shared parts** in a **new**, often **abstract**, **use case**, that is then **specialized by child use cases**
  - The child use case inherits all behavior described for the parent use case
    - The description of a child use case needs to follow the description of the parent use case in order to be considered complete
    - The child use case event flow explains how the inherited parents behavior is modified
    - Only the child use case knows of the relationship between the two use cases

Differences Between Include and Generalization

- Think the difference between inheritance and a sub-function
- Use case generalization:
  - The execution of the children is **dependent on the structure and behavior of the parent** (the reused part)
  - The children share similarities in purpose and structure
- Include relationship:
  - The execution of the base use case **depends only on the result of the function performed by the inclusion use case** (the reused part)
  - The base use cases reusing the same inclusion can have completely different purposes, but they need the same function to be performed

Generalizations Between Actors

- **Several actors can play the same role** in a particular use case
- Actors with common characteristics should be modeled by using actor-generalizations
- **A user can play several roles** in relation to the system (the user corresponds to several actors)
  - Represent the user by one actor who inherits several actors
  - Each inherited actor represents one of the user's roles relative to the system



A Teller and an Accountant, both of whom check the balance of an account, are seen as the same external entity by the use case that does the checking. The shared role is modeled as an actor, Balance Supervisor, inherited by the two original actors.

Review the Use Case Model

- Review the contents and structure of the use case model
  - Complete – all user roles, all tasks, understandable descriptions
  - Well structured – model is readable and understandable
  - **Don’t over do it** – the model may become too complex and less understandable
- Validate that the results of use case modeling conform to the customer's view of the system
- Review participants should include the analyst, stakeholders, users, and developers
- In practice the use case model *may* be transitional in project longevity as requirements are identified

“Edge” Use Cases

What Are Edge Use Cases?

- Apply use case modeling to **system features that go beyond end user goals and requirements**
- Why? **Holistic system thinking**
  - To better understand and capture system **quality attribute requirements**
  - Leads to **better system architecture and design**
  - Identifies **boundary and exception test cases**
- Candidate system features
  - **Misuse** – malicious security scenarios
  - **What if**, what can go wrong scenarios; e.g., exceptions, system safety
  - **System administration tasks**
  - **System life cycle**

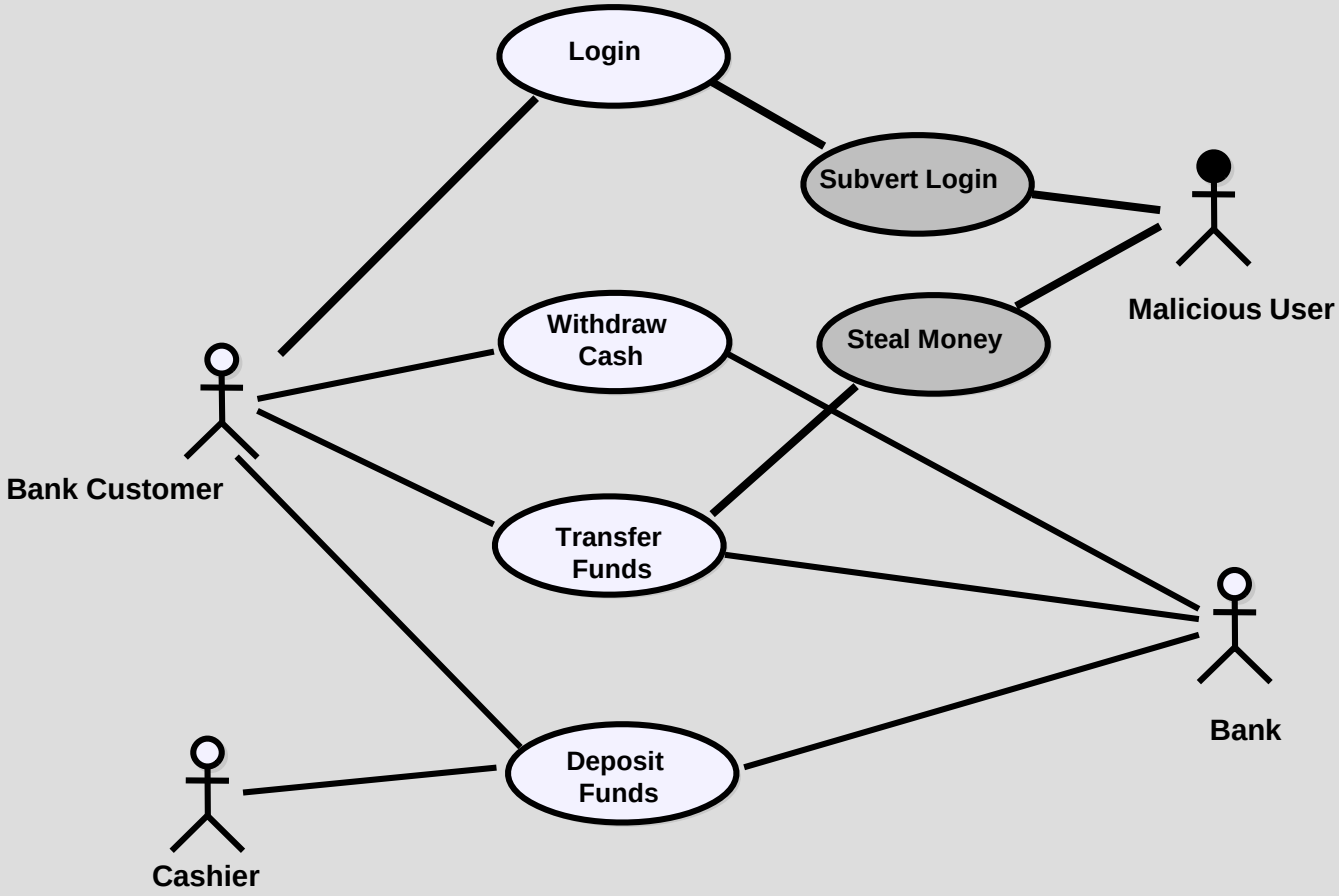
Misuse (Abuse) Cases

- **Misuse Case**: How the system shall respond to illegitimate use
- **Mis-Actor**: Attacker or malicious user
  - Undermine the assumptions and boundary conditions of the system
  - Identify **common patterns of attack**
- Derived use cases lead to **system quality requirements**
- **Apply normal use case style** to misuse case descriptions

Misuse Case Analysis Pattern

- The misuse case analysis pattern:
  - For a legitimate use case**, identify misuse cases that *threaten* legitimate use case** success
  - **Derive a use case to *mitigate*** the misuse case (as an included use case)
  - **What is the malicious actor's response** to the mitigation use case? (included misuse case)
  - **Derive another mitigation use case** to respond
  - Continue play-counter play …

Misuse Case Example



Misuse Case Example