Use case modeling

- Scenario: a sequence of steps describing an interaction (a "dialog") between a user and a system.
- Use case: a set of scenarios tied together by a common user goal.
- Actor (role): a role that a user plays with respect to the system; one that receives end-to-end service from the use case.

Use case functions (I)

- Use case captures functional requirement for a user
- Use cases are high-level, domain-oriented descriptions written in a way understood by project stakeholders
  - managers, developers, clients and users.

Use case functions (II)

- As a starting point in system modeling.
  - All the other artifacts are derived from the use cases.
- As a guide to testing the system built.
  - The system built must reflect the process presented in a use case.
  - Useful in function test and system test

Actor

- A role that a user plays with respect to the system.
- Role before people or job titles
- Most useful for discovering use cases: list actors, then the use cases for each actor
- Actor can be human or system

Dialog Between User and System

- User is an external actor who initiate interaction to the system
- In response, the system performs the desired actions.
- And so on.
Use Case Examples (I)

• “Good” use cases:
  – Withdraw money from an ATM
  – Order a product
  – Register for courses at a school
  – Check the spelling of a document in a word processor
  – Take a test
  – Write up test items

Use Case Examples (II)

• “Not-so-good” use cases:
  – Write data into a database
  – Create a table in a relational database
  – Formatting a result set retrieved
  – Check if certain items are in stock

Initial Use Cases

• From an analysis point of view, initial use cases should strongly reflect a complete domain process instead of individual steps.
• Later into the development, it may be helpful to identify use cases that are not so large as the initial use cases. These use cases are often result of use case factoring.

Use Case Format

• UML does not have a strict format for use case. A use case can include:
  – Use case name
  – Actors
  – Overview
  – Pre- and Post- conditions.
  – Course of events (steps)
  – Exceptions

Use Case Diagram


Use case classification

• A use case is
  – high-level if only an overview is written;
  – expanded if course of event is written
• A use case is
  – essential if it largely reflect the domain process without worrying about what technology or implementation are used (e.g., RDB and GUI);
  – real if implementing technology is committed
Use case identification

- Actor-based:
  - Identify the actors related to a system
  - For each actor, identify the processes they initiate or participate in.
- Event-based:
  - Identify external events system must respond to.
  - Relate the events to actors and use cases.

Use case and system

- Use cases constitute a system.
- To emphasize the system and its boundary, a system diagram is drawn.

System boundary

Use Case Tips (I)

- Remember, we are just starting. To make life easy, we can ignore pre- and post-conditions and exceptions (at least initially.)
- That said, use case is also constructed incrementally; don’t be worried that your initial use case looks too simple, and don’t be afraid to add things to your use case.

Use Case Tips (II)

- Write up high level use cases first.
  - Concentrate on name, actors, and overview.
  - Big, end-to-end use cases first; then additional use cases with the include, generalize and extend relationships.
- In one of the use case, write up the expanded version. Choose to expand the use case that
  - is most risky technically;
  - completes the largest percentage of the system function.

Use Case Tips (III)

- Normal flow first; alternatives in alternative sections.
High level use cases

Name: Take a Test

Actors: Student

Overview: Student logs into a test station to take a test. A number of problems are given upon completion, test grade is shown to the student.

Name: Write test items

Actors: Teacher

Overview: Teacher logs into system. For a chosen subject, a number of test items as well as their solutions are written. The teacher can observe/present previously written items. Upon completion, test item repository is updated.

Another example

Use case: Execute a plan

1. This use case begins when a plan has been created.
2. System posts the next action to implement to the to-do list.
3. Agent initiates Use case: Implement an action
4. If Use case: Implement an action returns completion, and this is not the last step, resume at step 2, otherwise resume at step 5.
5. System prompts the Agent to use case: Assess a plan.
6. If use case: Assess a plan returns success, System prompts the Agent to do use case: Record a sale, followed by use case: Do sales analysis; otherwise resume at Alternative: plan fails.
7. If use case: Assess a plan returns failure, System prompts the Agent if to redo sales analysis.

Use case relationships (I)

- The include relationship: a chunk of behavior that is similar across more than one use case.
- Generalization: one use case similar to another but does a bit more.
- The extend relationship: Similar to generalization but allows variations only at extension points. (Much like a Template (GoF))

Use case relationships (II)

- When use case development continues, you may find commonality among them. Factor commonality into new use cases and have the base use cases include them.
- Use case may have variants and optional steps. These variants are factored out of the base use case. Base case will keep the mandatory flows.
Use case factoring (II)

- Use cases can be generalized or specialized (gen-spec) much like a class. A specialized use case may appear where a base use case can.

Template method pattern (I)

- Source: GoF
- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Template method pattern (II)

- A good example of “refactoring to generalize” as described by Opdyke and Johnson.
  - You first identify the differences in the existing code
  - then separate the differences into new operation
  - Finally, you replace the differing code with a template method that calls one of these new operations.

Structure

- AbstractClass
  - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as
  - operations defined in AbstractClass or those of other objects.
- ConcreteClass
  - implements the primitive operations to carry out subclass-specific steps of the algorithm.

Further Readings

- Larman, Chapters 6-8, 26.
- Jacobson, Chapter 3.
- Fowler/Distilled, Chapter 3.
- GoF, Template method.