

[Handout for L3P1]

Problem before Solution: Figuring out Requirements

Requirements

A requirement specifies the intended usage of a software product. A development project may aim to replace or update an established software system (called a *brown-field* project) or it may aim to develop a totally new system with no obvious precedent (called a *green-field* project). In either case, the stakeholders' needs and expectations have to be understood, discussed, refined, clarified, scoped or re-scoped. Identifying users' requirements is not as straightforward as it sounds. It is not simply writing a wish list.

There are two different kinds of requirements: functional requirements and non-functional requirements. Functional requirements specify what the system should do, and non-functional requirements specify the constraints under which system gets developed and operates. Data requirements e.g. size, volatility, persistency etc., and Environment requirements e.g. technical environment in which system would operate or need to be compatible with, are categorized under non-functional requirements. Non functional requirements are easy to miss. We should spend extra effort in digging them out as early as possible because sometimes they are critical to the success of the software. E.g. A web application that is too slow or that has low security is unlikely to succeed even if it has all the right functionalities.

This handout covers two requirements-related activities that go hand-in-hand:

1. Establishing requirements: *Requirements gathering, requirements elicitation, requirements analysis, requirements capture* are some of the terms commonly and interchangeably used to represent the activity of understanding what a software product should do. The overall aim of such activity is to specify unambiguous requirements.
2. Specifying requirements: As we establish requirements, they should be recorded in some way for future reference. Furthermore, it is advisable to show these requirements to stakeholders, get their feedback and refine them further. The next phase is to convert them into a product specification that specifies how the product will address the requirements.

Establishing requirements

There are many techniques used in requirement gathering activity. Given next are some of those techniques.

Brainstorming

Brainstorming is a group activity designed to generate a large number of diverse and creative ideas for the solution of a problem. In a brainstorming session there are no "bad" ideas. The aim is to generate ideas; not to validate them. Brainstorming encourages you to "think outside the box" and put "crazy" ideas on the table without fear of rejection.

User surveys

Carefully designed questionnaires can be used to get responses and opinions of a large number of users on any current system or a new innovation.

Observation

Observation of users in their natural work environment is a common technique used to understand the task and environment of the user. Interaction logging on an existing system can also be made to gather information about how an existing system is being used.

Interviews

Interviewing potential stakeholders and domain experts can give us useful information about a domain. Interview is a good technique at getting users to explore what users feel about the required system. Interviews also provide opportunities for development team members to meet stakeholders and to make stakeholders feel involved in the development process.

Focus groups

Focus groups are kind of informal interviews in an interactive group setting. A group of people (e.g. potential users, beta testers) are asked about their understanding of a specific issue or a process. Focus groups can bring out undiscovered conflicts and misunderstanding among stakeholder interests which can then be resolved or clarified as necessary.

Prototyping

A prototype is a mock up, a scaled down version, or a partial system constructed

- (a) to get users' feedback, or
- (b) to validate a technical concept (a "proof-of-concept" prototype), or
- (c) to give a preview of what to come, or to compare multiple alternatives in a small scale before committing fully to one alternative, or
- (d) for early field-testing under controlled conditions.

Early UI prototyping, i.e. sketching user interface for the intended product, is a good technique to uncover requirements, in particular, those related to how users want to interact with the system. UI prototypes are often used in brainstorming sessions, or in meetings with the users to get quick feedback from them.

Given below is a simple text UI prototype for a primitive CLI Minesweeper.

D:\>java MinesweeperTextUI

Enter command: new

[0,0:H][0,1:H][0,2:H]

[1,0:H][1,1:H][1,2:H]

[2,0:H][2,1:H][2,2:H]

Enter command: mark 2 1

[0,0:H][0,1:H][0,2:H]

[1,0:H][1,1:H][1,2:H]

[2,0:H][2,1:M][2,2:H]

Enter command: clear 0 0

[0,0:0][0,1:H][0,2:H]

[1,0:H][1,1:H][1,2:H]

[2,0:H][2,1:M][2,2:H]

Enter command: clear 2 2

[0,0:0][0,1:H][0,2:H]

[1,0:H][1,1:H][1,2:H]

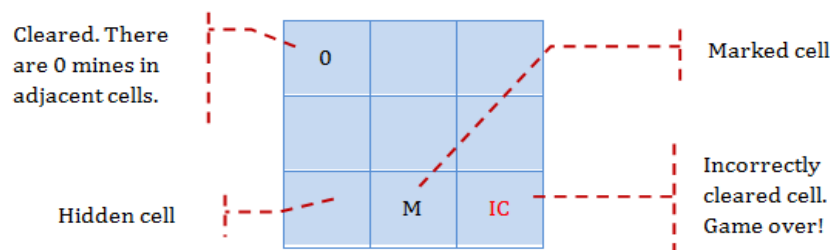
[2,0:H][2,1:M][2,2:IC]

You Lost :-)

Format → [coordinates : cell appearance]

Cell appearance: H=hidden, IM=Incorrectly Marked, IC=Incorrectly Cleared, M=Marked 0-8:number of mines in adjacent cells.

Here is a simple GUI prototype for the same Minesweeper, created using Powerpoint.



Here is a more realistic example of a GUI prototype created using the Balsmiq (a software for creating UI prototypes)².



Analyzing similar products and documentation

Studying existing products can unearth shortcomings of existing solutions that can be addressed by a new product. For example, when developing a game for a mobile device, a look at a similar PC game can give insight into what kind of features and interactions

² Image taken from <http://balsamiq.com/products/mockups>. Balsmiq has a free version for creating UI prototypes.

the game might offer. Product manuals and other technical documentation of an existing system can be a good way to learn about how the existing solutions work.

Specifying requirements

Given next are some tools and techniques that can be used to specify requirements. Note that they can also be used for establishing requirements too.

Textual descriptions

This is the most straight forward way of describing requirements. A textual description can be used to give a quick overview of the domain/system that is understandable to both users and development team. Textual descriptions are especially useful when describing the vision of a product. However, lengthy textual descriptions are hard to follow.

Feature list

As the name suggests, it is a list of features (or functionalities), possibly grouped according to some criteria such as priority (e.g. must have, nice to have,...), order of delivery, object or process related (e.g. order-related, invoice-related, ...). Here is a sample feature list from Minesweeper (only a brief description have been provided to save space).

1. Basic play – Single player play.
2. Difficulty levels – Additional Medium and Advanced levels.
3. Versus play – Two players can play against each other.
4. Timer – Additional fixed time restriction on the player.

User stories

User stories are brief (typically, 1-3 sentences) descriptions what the system can do for users, written in the customers language. Often, User stories are written by the customers themselves. Here are some example User stories for the IVLE system:

- Students can download files uploaded by lecturers.
- A lecturer can create discussion forums.
- A tutor can generate attendance sheets for printing.

User stories are mainly used for early estimation and scheduling purposes.

According to [1], the biggest difference between user stories and traditional requirements specifications is in the level of detail. User stories should only provide enough detail to make a reasonably low risk estimate of how long the story will take to implement. When the time comes to implement the story, developers will go to the customer and hammer out a more detailed description of the requirements face-to-face.

Use cases

A use case describes an interaction between the user and the system for a specific functionality of the system (i.e. 'cases of users using the system'). For example, 'check account balance' can be use case for an Automated Teller Machine (ATM). Therefore, a *use case* model is a way of capturing the functional requirements of a system. Use cases are part of the modeling notation called UML.

Side note: Unified Modeling Language (UML)

Unified Modeling Language (UML) <http://www.uml.org/#UML2.0> is a graphical notation to describe various aspects of a software system. UML is the brainchild of three software modeling specialists James Rumbaugh, Grady Booch and Ivar Jacobson (also known as the Three Amigos). Each of them has developed their own notation for modeling software systems before joining force to create a unified modeling language (hence, the term 'Unified' in UML). UML is currently the de facto modeling notation used in the industry. CS2103 uses UML version 2.0.

A *use case* can include a narrative of how the system is used for that case. The following is a such a description of ATM's 'check account balance' use case:

1. User inserts an ATM card
2. ATM prompts for PIN
3. User enters PIN
4. ATM prompts for withdrawal amount
5. User enters the amount
6. ATM ejects the ATM card and issues cash
7. User collects card and the cash.

Note that a use case describes only the externally visible behavior, not internal details, of a system.

Next, let us look at various components of a use case. A use case is an interaction between *Actors* and the *system*. Each use case is given a unique identification. An actor is a role played by a user. An actor can be a human or another system. For example, consider a Software System: IVLE (Integrated Virtual Learning Environment, an e-learning tool). Actors are not part of the system. They live outside the system. Note that the system being modeled is also called the *subject* of the model.

Its Actors would be: Guest, Student, Staff, Admin, IMMS (an exam management system), LINC (a library management system).

A use case can involve multiple actors.

Software System: IVLE
Use case: UC01 conduct survey
Actors: Staff, Student

An actor can be involved in many use cases.

Software System: IVLE
Actor: Staff
Use cases: UC01 conduct survey, UC02 Set Up Course Schedule, UC03 Email Class, ...

A single person/system can play many roles.

Software System: IVLE
Person: a student
Actors (or Roles): Student, Guest, Tutor

Many persons/systems can play a single role.

Software System: IVLE

Actor(or role) : Student

Persons that can play this role : undergraduate student, graduate student, a staff member doing a part-time course, exchange student.

The formal definition of a use case is *a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.* (The Unified Modeling Language User Guide, 2e, G Booch, J Rumbaugh, and I Jacobson)

The *flow of events* is a sequence of steps that describes the interaction between the system and the Actors for a use case. Each step is given as a simple statement.

Use case: UC01 conduct survey

1. Staff creates the survey.
 2. Student completes the survey.
 3. Staff views the survey results.
- Use case ends.

Every step should clearly show *who* does *what*. A step gives the intention of the actor (not the mechanics). That means UI details are usually omitted.

User right-clicks the text box and chooses 'clear': This contains UI-specific details and is not a good use case step.

User clears the text entry: This is better because it omits UI-specific details.

The following is an illustration of how do we include repetitive steps in a scenario.

Software System: Square game

Use case: UC02 - Play a Game

Actors: Player (multiple players)

1. A Player starts the game.
2. SquareGame asks for player names.
3. Each Player enters his own name.
4. SquareGame shows the order of play.
5. SquareGame prompts for the current Player to throw die.
6. Current Player adjusts the throw speed.
7. Current Player triggers the die throw.
8. Square Game shows the face value of the die.
9. Square Game moves the Player's piece accordingly.

Steps 5-9 are repeated for each Player, and for as many rounds as required until a Player reaches the 100th square.

10. Square Game shows the Winner.
- Use case ends.

The *Main Success Scenario* (MSS) describes the most straightforward interaction for a given use case, which assumes that nothing goes wrong. This is also called the *Basic*

Course of Action or the *Main Flow of Events* of a use case. Following is another example of an MSS of a use case Transfer Money for an Online Banking System.

System: Online Banking System (OBS)

Use case: UC23 - Transfer Money

Actor: User

MSS:

1. User chooses to transfer money.
2. OBS requests for details of the transfer.
3. User enters the requested details.
4. OBS requests for confirmation.
5. User confirms transfer.
6. OBS transfers the money and displays the new account balance.

Use case ends.

Extensions:

- 3a. OBS detects an error in the entered data.
 - 3a1. OBS requests for the correct data.
 - 3a2. User enters new data.

Steps 3a1-3a2 are repeated until the data entered are correct.

Use case resumes from step 4.

- 3b. User requests to effect the transfer in a future date.
 - 3b1. OBS requests for confirmation.
 - 3b2. User confirms future transfer.
- Use case ends.

- *a. At any time, User chooses to cancel the transfer
 - *a1. OBS requests to confirm the cancellation
 - *a2. User confirms the cancellation
- Use case ends.

- *b. At any time, 120 seconds lapse without any input from the User
 - *b1. OBS cancels the transfer
 - *b2. OBS informs the User of the cancellation
- Use case ends.

Note how the MSS assumes that all entered details are correct and ignores problems such as timeouts, network outages etc. MSS does not tell us what happens if the user enters incorrect data.

Extensions, given below the MSS, are kind of "add-on"s to the MSS. They are also called *exceptional flow of events* or *alternative flow of events*. They describe variations of the scenario that can happen if certain things are not as expected by the MSS. Extensions appear below the MSS. Note that the numbering style is not a rule, but just a convention. The third extension, labeled as '*a' can happen at any step (hence, the '*').

When separating extensions from the MSS, keep in mind that the MSS should be self-sufficient. That is, the MSS should give us a complete usage scenario. Also note that it is not useful to mention events such as power failures or system crashes as extensions because the system cannot function beyond such catastrophic failures.

A use case can "include" another use case. Underlined text is commonly used to show an *inclusion* of a use case. Inclusions are useful,

- when you don't want to clutter a use case with too many low-level steps.
- when a set of steps is repeated in multiple use cases.

Software System: IVLE
 Use case: UC01 - Conduct Survey
 Actors: Staff, Student
 MSS:
 1. Staff creates the survey (UC44).
 2. Student completes the survey (UC50).
 3. Staff views the survey results.
 Use case ends.

Preconditions specify the specific state we expect the system to be in before the use case starts.

Software System: Online Banking System
 Use case: UC23 - Transfer Money
 Actor: User
Preconditions: User is logged in.
 MSS:
 1. User chooses to transfer money.
 2. OBS requests for details for the transfer.
 ...

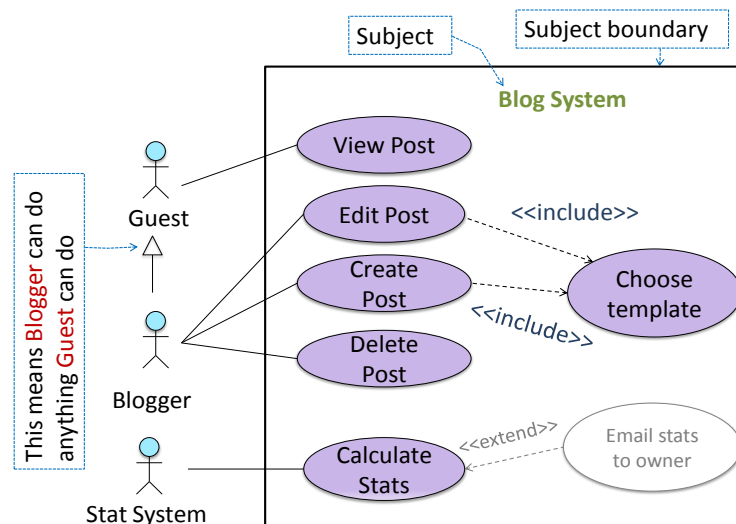
Guarantees specify what the use case promises to give us at the end of its operation.

Software System: Online Banking System
 Use case: UC23 - Transfer Money
 Actor: User
 Preconditions: User is logged in.
Guarantees:
 * **Money will be transferred to the destination account.**
 * **All steps of the transaction will be logged.**
 MSS:
 1. User chooses to transfer money.
 2. OBS requests for details for the transfer.
 ...

Use cases should be easy to read. Note that there is no strict rule about writing all details of all steps or a need to use all the elements of a use case.

UML is not very specific about the text contents of a use case. Hence, many styles exist for writing use cases. For example, the steps can be written as a continuous paragraph. However, UML does specify a diagrammatic notation for use cases.

Use case diagrams help to give an overview of a set of use cases (a kind of a "graphical table of contents" for the use cases).



<<extend>> relationship between use cases is used to capture extensions to the use cases. Note how the arrow is pointing the other way.

Note that use cases can be specified at various levels of detail. Consider the three use cases given below for the IVLE system.

- (a) conduct a survey
- (b) take the survey
- (c) answer survey question

Clearly, (a) is at a higher level than (b) and (b) is at a higher level than (c). While modeling user-system interactions, start with high level use cases and progressively work toward lower level use cases. It is also important to be mindful at which level of details you are working on and not to mix use cases of different levels.

Here are some of the advantages of documenting system requirements as use cases:

- Since they use a simple notation and plain English descriptions, they are easy for users to understand and give feedback.
- They decouple user intention from mechanism (note that use cases should not include UI-specific details), allowing the system designers more freedom to optimize how a functionality is provided to a user.
- Identifying all possible extensions encourages us to consider all situations a software product might face during its operations.
- Separating typical scenario from special cases encourages us to optimize the typical scenario.

One of the main disadvantages of use cases is that they are not good for capturing requirements that does not involve a user interacting with the system. Hence, they should not be used as the sole means of specifying requirements.

Glossary

Glossary serves to ensure all stakeholders have a common understanding of the noteworthy terms, abbreviation, acronyms etc. e.g. here is a partial Glossary from a variant of the *Snakes and ladders* game:

- *Conditional square*: A square that specifies a specific face value which a player has to throw before his/her piece can leave the square.
- *Normal square*: a normal square does not have any conditions, snakes, or ladders in it.

Supplementary requirements

Supplementary requirements section contains elements which do not fit elsewhere. The following are a few examples of requirements typically found under this heading.

- Business/domain rules: e.g. the size of the minefield cannot be smaller than five.
- Constraints: e.g. the system should be backward compatible with data produced by earlier versions of the system; system testers are available only during the last month of the project; the total project cost should not exceed \$1.5 million.
- Technical requirements: e.g. the system should work on both 32-bit and 64-bit environments.
- Performance requirements: e.g. The system should respond within 2 seconds.
- Quality requirements: e.g. the system should be usable by a novice who has never carried out an online purchase before.
- Process requirements: e.g. the project is expected to follow a process that delivers a feature set every one month.
- Notes about project scope: e.g. The product is not required to handle printing of reports.
- Any other noteworthy points: e.g. The game should not use images offensive to those injured in real mine clearing activities.

A note on categorizing and prioritizing requirements

Short timelines and limited resources often mean you cannot implement all requirements at once. The prioritization criteria include importance and urgency of requirements from user point of view against the constraints of schedule, budget, staff resources, and quality goals as seen by the development team.

A common approach to prioritization is to group requirements into three priority categories. Typical three-level scales are High, Medium, Low or Minimal, Typical, Novel. Note that all such scales are subjective, and stakeholders define the meaning of each level in the scale for the project at hand. For example:

- Essential: The product must have this requirement fulfilled else it does not get user acceptance
- Typical: Most similar systems have this feature although the product can survive without it.
- Novel: New features that could differentiate this product from the rest.

At the same time, some requirements may get thrown out as 'out of scope'.

References

- [1] <http://www.extremeprogramming.org/rules/userstories.html>. This is the main website for eXtreme Programming (XP), an approach to software development

currently popular among practitioners. User stories are commonly used among XP practitioners to capture requirements.

Worked examples

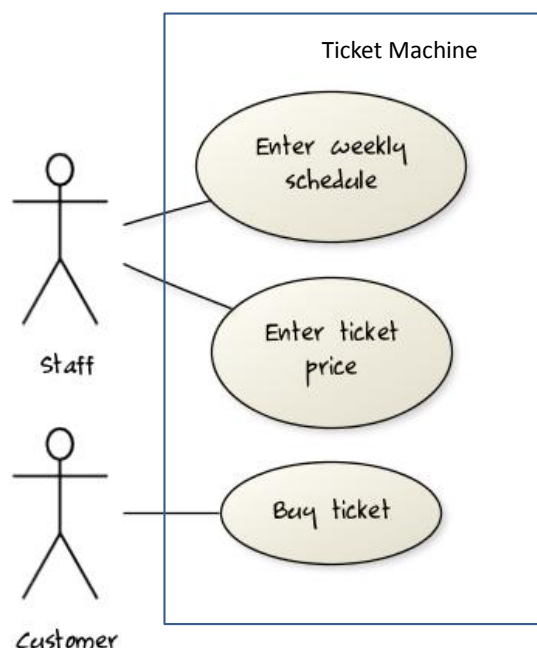
Note that most these questions can be answered in multiple ways. The answer given is just one way of answering it, just for illustration.

[Q1] Use case diagram for ticket vending machine

Consider a simple movie ticket vending machine application. Every week, the theatre staff will enter the weekly schedule as well as ticket price for each show. A customer sees the schedule and the ticket price displayed at the machine. There is a slot to insert money, a keypad to enter a code for a movie, a code for the show time, and the number of tickets. A display shows the customer's balance inside the machine. A customer may choose to cancel a transaction before pressing the "buy" button. Printed tickets come out of a slot at the bottom of the machine. The machine also displays message e.g. "Please enter more money" or "request fewer tickets" or "SOLD OUT!". Finally, there is a "Return Change" button so the customer can get his unspent money back.

Draw a use case diagram for the above requirements.

[A1]



Note that most of the details in the description are better given as part of the use case description rather than as low-level use cases in the diagram.

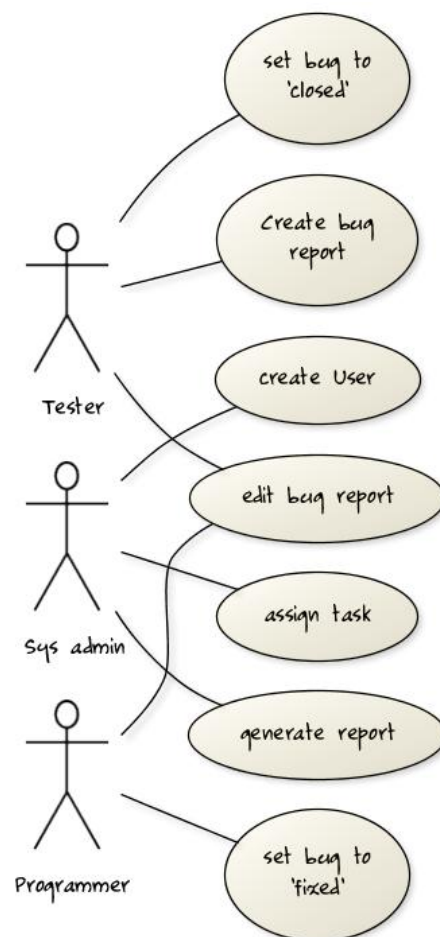
[Q2] Use case diagram for QA system

A software house wishes to automate its Quality Assurance division. Develop a Use Case diagram to capture their requirements given below.

The system is to be used by Testers, Programmers and System Administrators. Only an administrator can create new users and assign tasks to programmers. Any tester can

create as well as set the status of a bug report to 'closed'. Only a programmer can set the state of a bug report to 'fixed', but a programmer cannot set the status of a bug report to 'closed'. Each tester is assigned just one task at a time. A task involves testing of a particular component for a particular customer. Tester must document the bugs they find. Each bug is given a unique identifier. Other information recorded about the bug is component id, severity, date and time reported, programmer who is assigned to fix it, date fixed, date retested and date closed. The system keeps track of which bugs are assigned to which programmer at any given time. It should be able to generate reports on number of bugs found, fixed and closed e.g. number of bugs per component and per customer; number of bugs found by a particular tester ; number of bugs awaiting to be fixed; number of bugs awaiting to be retested; number of bugs awaiting to be assigned to programmers etc.

[A2]



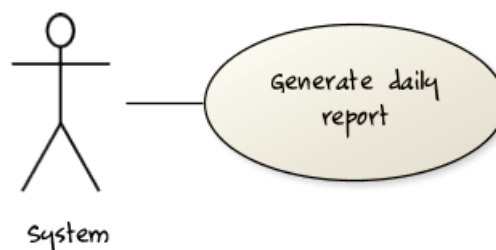
Explanation: The given description contains information not relevant to use case modeling. Furthermore, the description is not enough to complete the use case diagram. Both those are realities of real projects. However, the process of trying to create this use case diagram prompts us to investigate issues such as:

- Is 'edit bug report' a use case or editing the bug report is covered by other use cases such as those for setting the status of bug reports? If it is indeed a separate use case, who are the actors of that use case?

- Does 'assign task' simply means 'assign bug report' or is there any other type of tasks?
- There was some mention about Customers and Components. Does the system have to support use cases for creating and maintaining details about those entities? For example, should we have a 'create customer record' use case?
- Which actors can do the 'generate report' use case? Are reports generated automatically by the system at a specific time or generated 'on demand' when users request to view them? Do we have to treat different types of reports as different use cases (in case some types of reports are restricted to some types of users)? The above diagram assumes (just for illustration) that the report is generated on demand and only the system admin can generate any report. Also refer to the 'System as an actor' sidebar given below.

System as an actor

Some use System as an actor to indicate that something is done by the system itself without being initiated by a user or an external system. For example, to indicate that the system generates daily reports at midnight, as shown below.



However, some argue that only use cases providing value to an external user/system should be shown in the use case diagram. For example, they argue that 'view daily report' should be the use case and 'generate daily report' is not to be shown in the use case diagram because it is simply something the system has to do to support the 'view daily report' use case.

In CS2103, we recommend you follow the latter view (i.e. not to use System as a user) and use use cases to model behaviors that involve an external actor.

[Q3] Adding 'standing ground' to minimal Minesweeper

Consider the minimal CLI Minesweeper referred to in the handout. Given below is its main use case.

Use case: 01 - play game

Actors: Player

MSS:

1. Player starts a new game.
 2. Minesweeper shows the minefield with all cells initially hidden.
 3. Player marks or clears a hidden cell.
 4. Minesweeper shows the updated minefield.
- Repeat steps 3-4 until the game is either won or lost.

5. Minesweeper shows the result.

Use case ends.

Modify it to incorporate the following new feature.

Feature id: standing_ground

Description: At the beginning of the game, the player chooses five cells to be revealed without penalty. This is done one cell at a time. If the cell so selected is mined, it will be marked automatically. The objective is to give some 'standing ground' to the player from which he/she can deduce remaining cells. The player cannot mark or clear cells until the standing ground is selected.

[A3]

Use case: 01 - play game

Actors: Player

MSS:

1. Player starts a new game.
 2. Minesweeper shows the minefield with all cells initially hidden.
 3. Player clears a cell.
 4. System reveals the cell (without penalty).
 - Repeat the above 2 steps 5 times.
 5. Player marks or clears a hidden cell.
 6. Minesweeper shows the updated minefield.
 - Repeat steps 5-6 until the game is either won or lost.
 7. Minesweeper shows the result.
- Use case ends.

[Q4] EZ-Link top-up use case

Complete the following use case (MSS, extensions, etc.). Note that you should not blindly follow how the existing EZ-Link machine operates since it will prevent you from designing a better system. You should consider all possible extensions without complicating the use case too much.

System: EZ-Link machine (those found at MRTs)

Use case: UC2 top-up EZ-Link card

Actor: EZ-Link card user

[A4]

System: EZ-Link machine (those found at MRTs)

Use case: UC2 top-up EZ-Link card

Actor: EZ-Link card user

Preconditions: All hardware in working order.

Guarantees: MSS -> the card will be topped-up.

MSS:

1. User places the card on the reader.
2. System displays card details and prompts for desired action.
3. User selects top-up.

4. System requests for top-up details (amount, payment option, receipt required?).
 5. User enters details.
 6. System processes cash payment (UC02) or NETS payment (UC03).
 7. System updates the card value.
 8. System indicates transaction as completed.
 9. If requested in step 5, system prints receipt.
 10. User removes the card.
- Use case ends.

Extensions:

- *a. User removed card or other hardware error detected.
 - *a1. System indicates the transaction has been aborted.
- Use case ends.

Notes:

- We assume that the only way to cancel a transaction is by removing the card.
- By not breaking step 4 into further steps, we avoid committing to a particular mechanism to enter data. For example, we are free to accept all data in one screen.
- In step 5, we assume that the input mechanism does not allow any incorrect data.

System: EZ-Link machine

Use case: UC03 process NETS payment

Actor: EZ-Link card user

Preconditions: A transaction requiring payment is underway.

Guarantees: MSS → Transaction amount is transferred from user account to EZ-Link company account.

MSS:

1. System requests to insert ATM card.
 2. User inserts the ATM card.
 3. System requests for PIN.
 4. User enters PIN.
 5. System reports success.
- Use case ends.

Extensions:

- 2a. Unacceptable ATM card (damaged or inserted wrong side up).
- ...
- 4a. Wrong PIN.
- ...
- 4b. Insufficient funds.
- ...
- *a. Connection to the NETS gateway is disrupted.
- ...

Note: UC02 can be written along similar lines.

[Q5] IVLE – reply to post use case

Complete the following use case (MSS, extensions, etc.).

System: IVLE

Use case: UC01 reply to post in the forum

Actor: Student

[A5]

System: IVLE

Use case: UC01 reply to post in the forum

Actor: Student

Preconditions: Student is logged in and has permission to post in the forum. The post to which the Student replies already exists.

Guarantees:

- MSS→ post will be added to the forum.

MSS:

1. Student chooses to reply to an existing post.
2. IVLE requests the user to enter post details.
3. Student enters post details.
4. Student submits the post.
5. IVLE displays the post.

Use case ends.

Extensions:

*a. Internet connection goes down.

...

*b. IVLE times out.

...

3a. Student chooses to 'preview' the post.

3a1. IVLE shows a preview.

3a2. User chooses to go back to editing.

Use case resumes at step 3.

3b. Student chooses to attach picture/file

...

3c. Student chooses to save the post as a draft.

3c1. IVLE confirms draft has been saved.

Use case ends.

3d. Student chooses to abort the operation.

...

4a. The post being replied to is deleted by the owner while the reply is being entered.

...

4b. Unacceptable data entered.

...

---End of document---