4. Requirements Elicitation: Solutions

4-1 *Consider your watch as a system and set the time 2 minutes ahead. Write down each interaction between you and your watch as a scenario. Record all interactions, including any feedback the watch provides you.*

This scenario could be written using any watch. Below is an example of a watch with a digital display and two buttons. Any solution should be detailed enough that an ignorant user can execute the scenario with somebody else's watch.

Scenario name	`setWatch2MinutesAhead`
Participating actor instances	`allen:WatchOwner`
Flow of events	1. The `WatchOwner` presses both `Watch` buttons simultaneously. 2. The `Watch` enters "set time" mode and indicates this by blinking the hour digits. 3. The `WatchOwner` presses the left button once. 4. The `Watch` stops blinking the hour digits and starts blinking the minutes. 5. The `WatchOwner` presses the right button twice. 6. The `Watch` increments the minutes by two. 7. The `WatchOwner` presses both buttons simultaneously. 8. The `Watch` stops blinking.

Figure 4-1 The `setWatch2MinutesAhead` scenario.

4–2 *Consider the scenario you wrote in Exercise 4-1 Identify the actor of the scenario. Next, write the corresponding use case* `SetTime`*. Include all cases, and include setting the time forward, backward, setting hours, minutes, and seconds.*

The following use case is generalized from the scenario of Figure 4-1. The level of detail of the use case should be the same as the level of detail of the scenario in the previous exercise.

Use case name	`SetTime`
Participating actor	Initiated by `WatchOwner`
Entry condition	1. The `Watch` is in "read time" mode.
Flow of events	2. The `WatchOwner` presses both `Watch` buttons simultaneously. 3. The `Watch` enters "set hour" mode and indicates this by blinking the hour digits. 4. In the "set hour" mode, the `WatchOwner` can advance the hour digits by pressing the right button. Each time the right button is pressed, the hours are incremented by one. If the hours reach 23 and the right button is pressed, the hours are reset to 0. The date is not changed. 5. At any time in the "set hour" mode, the `WatchOwner` can switch to the "set minutes" mode by selecting the left button. To indicate this, the `Watch` stops blinking the hour digits and starts blinking the minute digits. 6. In the "set minutes" mode, the `WatchOwner` can advance the minute digits by pressing the right button. Each time the right button is pressed, the minutes are incremented by one. If the minutes reach 59 and the right button is pressed, the minutes are reset to 0. The hours are not changed.

Figure 4-2 `SetTime` use case.

	7. At any time in the “set minutes” mode, the `WatchOwner` can switch to the “set seconds” mode by selecting the left button. To indicate this, the `Watch` stops blinking the minute digits and starts blinking the second digits.
	8. In the “set seconds” mode, the `WatchOwner` can advance the second digits by pressing the right button. Each time the right button is pressed, the seconds are incremented by one. If the seconds reach 59 and the right button is pressed, the seconds are rest to 0. The hours and the minute digits are not changed.
	9. At any time in the “set seconds” mode, the `WatchOwner` can switch to the “set hours” mode by pressing the left button. To indicate this, the `Watch` stops blinking the second digits and starts blinking the hour digits.
	10. At any time in the “set hours,” “set minutes,” and “set seconds” mode, the `WatchOwner` can return to “read time” mode by pressing both `Watch` buttons simultaneously.
Exit condition	11. The `Watch` is in “read time” mode with the new time set.
Special requirements	None.

Figure 4-2 `SetTime` use case.

4–3 *Assume the watch system you described in Exercises 4-1 and 4–2 also supports an alarm feature. Describe setting the alarm time as a self-contained use case named* `SetAlarmTime`.

The difference between setting the time of a watch and setting the alarm time should be minimal. A typical answer can be produced by copying and pasting solution of Exercise 4–2 and modifying the beginning and the end of the use case. The rationale behind this similarity is that any user interface should provide similar interfaces for addressing similar functionality (i.e., user interface consistency). Below is an example generated from the use case of Figure 4-2.

Use case name	`SetAlarmTime`
Participating actor	Initiated by `WatchOwner`
Entry condition	1. The `Watch` is in “read time” mode.
Flow of events	2. The `WatchOwner` presses both `Watch` buttons simultaneously. The `Watch` enters the “set hour” mode. The `WatchOwner` continues to press both buttons. After one second, a small alarm bell symbol appears on the display and blinks, indicating that the hours of the alarm are being set (as opposed to the normal time).
	3. In the “set hour” mode, the `WatchOwner` can advance the hour digits by pressing the right button. Each time the right button is pressed, the hours are incremented by one. If the hours reach 23 and the right button is pressed, the hours are reset to 0. The date is not changed.
	4. At any time in the “set hour” mode, the `WatchOwner` can switch to the “set minutes” mode by selecting the left button. To indicate this, the `Watch` stops blinking the hour digits and starts blinking the minute digits.
	5. In the “set minutes” mode, the `WatchOwner` can advance the minute digits by pressing the right button. Each time the right button is pressed, the minutes are incremented by one. If the minutes reach 59 and the right button is pressed, the minutes are reset to 0. The hours are not changed.
	6. At any time in the “set minutes” mode, the `WatchOwner` can switch to the “set seconds” mode by selecting the left button. To indicate this, the `Watch` stops blinking the minute digits and starts blinking the second digits.
	7. In the “set seconds” mode, the `WatchOwner` can advance the second digits by pressing the right button. Each time the right button is pressed, the seconds are incremented by one. If the seconds reach 59 and the right button is pressed, the seconds are rest to 0. The hours and the minute digits are not changed.

Figure 4-3 `SetAlarmTime` use case.

	8. At any time in the "set seconds" mode, the `WatchOwner` can switch to the "set hours" mode by pressing the left button. To indicate this, the `Watch` stops blinking the second digits and starts blinking the hour digits. 9. At any time in the "set hours," "set minutes," and "set seconds" mode, the `WatchOwner` can return to "read time" mode by pressing both `Watch` buttons simultaneously. The digits stop blinking and the alarm bell stays on and stops blinking.
Exit condition	10. The `Watch` is in "read time" mode with the new alarm time set.
Special requirements	None.

Figure 4-3 `SetAlarmTime` use case.

4–4 *Examine the `SetTime` and `SetAlarmTime` use cases you wrote in Exercises 4–2 and 4–3 Eliminate any redundancy by using an include relationship. Justify why an include relationship is preferable to an extend relationship in this case.*

The solution to this exercise should include three use cases: `SetTime`, `SetAlarmTime`, and a third use case (which we called `SpecifyTime`) made out of the common parts of `SetTime` and `SetAlarmTime`. The essential point is to factor out as much common functionality as possible into the third use case. The rationale is that, by eliminating redundancy, fewer inconsistencies are introduced in the future when the use cases are modified. Below is a sample solution for this exercise.

Use case name	`SetTime`
Participating actor	Initiated by `WatchOwner`
Entry condition	1. The `Watch` is in "read time" mode.
Flow of events	2. The `WatchOwner` presses both `Watch` buttons simultaneously. The `Watch` enters the "set hour" mode and indicates this by blinking the hour digits. 3. The SpecifyTime is included here. A the end of the SpecifyTime use case, the `WatchOwner` has specified a time. 4. The `WatchOwner` presses both buttons simultaneously to return to the "read time" mode. The new time is set.
Exit condition	5. The `Watch` is in "read time" mode with the new time set.
Special requirements	None.

Figure 4-4 `SetTime` use case.

Use case name	`SetAlarmTime`
Participating actor	Initiated by `WatchOwner`
Entry condition	1. The `Watch` is in "read time" mode.

Figure 4-5 `SetAlarmTime` use case.

Flow of events	2. The `WatchOwner` presses both `Watch` buttons simultaneously. The `Watch` enters the "set hour" mode and indicates this by blinking the hour digits. The `Watch` enters the "set hour" mode. The `WatchOwner` continues to press both buttons. After one second, a small alarm bell symbol appears on the display and blinks, indicating that the hours of the alarm are being set (as opposed to the normal time). 3. The SpecifyTime is included here. A the end of the SpecifyTime use case, the `WatchOwner` has specified a time for the alarm. 4. The `WatchOwner` presses both buttons simultaneously to return to the "read time" mode. The new alarm time is set.
Exit condition	5. The `Watch` is in "read time" mode with the new alarm time set.
Special requirements	None.

Figure 4-5 `SetAlarmTime` use case.

Use case name	`SpecifyTime`
Participating actor	Initiated by `WatchOwner`
Entry condition	1. The `Watch` is in "set alarm" or "set time" mode. The hour digits are blinking.
Flow of events	2. In the "set hour" mode, the `WatchOwner` can advance the hour digits by pressing the right button. Each time the right button is pressed, the hours are incremented by one. If the hours reach 23 and the right button is pressed, the hours are reset to 0. The date is not changed. 3. At any time in the "set hour" mode, the `WatchOwner` can switch to the "set minutes" mode by selecting the left button. To indicate this, the `Watch` stops blinking the hour digits and starts blinking the minute digits. 4. In the "set minutes" mode, the `WatchOwner` can advance the minute digits by pressing the right button. Each time the right button is pressed, the minutes are incremented by one. If the minutes reach 59 and the right button is pressed, the minutes are reset to 0. The hours are not changed. 5. At any time in the "set minutes" mode, the `WatchOwner` can switch to the "set seconds" mode by selecting the left button. To indicate this, the `Watch` stops blinking the minute digits and starts blinking the second digits. 6. In the "set seconds" mode, the `WatchOwner` can advance the second digits by pressing the right button. Each time the right button is pressed, the seconds are incremented by one. If the seconds reach 59 and the right button is pressed, the seconds are rest to 0. The hours and the minute digits are not changed. 7. At any time in the "set seconds" mode, the `WatchOwner` can switch to the "set hours" mode by pressing the left button. To indicate this, the `Watch` stops blinking the second digits and starts blinking the hour digits. 8. At any time in the "set hours," "set minutes," and "set seconds" mode, the `WatchOwner` can return to "read time" mode by pressing both `Watch` buttons simultaneously. The digits stop blinking and the alarm bell stays on and stops blinking.
Exit condition	9. The `Watch` is in "read time" mode.
Special requirements	None.

Figure 4-6 `SpecifyTime` use case.

4–5 *Assume the `FieldOfficer` can invoke a Help feature when filling an `EmergencyReport`. The `HelpReportEmergency` feature provides a detailed description for each field and specifies which fields are required. Modify the `ReportEmergency` use case (described in Figure 4-10) to include this help functionality. Which relationship should you use to relate the `ReportEmergency` and `HelpReportEmergency`?*

The ReportEmergency use case does not need to be modified when the HelpReportEmergency is written as an extension. The advantage of writing behavior such as help and exceptions as extensions is that they can be reused in different use cases or in different steps of a single use case. Below is a sample solution with the `<<extend>>` relationship.

Use case name	`HelpReportEmergency`
Participating actor	Initiated by `FieldOfficer`
Entry condition	1. The Emergency Report Form is visible on the `FieldOfficer`'s screen and has not been submitted yet. This use case extends steps 2 and 3 of ReportEmergency.
Flow of events	2. The `FieldOfficer` selects a field of the `Emergency Report Form` (one of "emergency type," "location," "incident description," "resource request," and "hazardous material") and presses the help key. 3. `FRIEND` displays in a separate window an explanatory text describing whether the field is required or not, what range of values is possible, and examples for typical cases. 4. The `FieldOfficer` may return to the form without closing the help window. For terminals with large enough screens, the `FieldOfficer` can display both the form and the help window side by side.
Exit condition	5. The use case ends when the `FieldOfficer` closes the help window.
Special requirements	None.

Figure 4-7 `HelpReportEmergency` use case.

Location	*Use case description*
Field Officer station	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal. 2. FRIEND responds by presenting a form to the officer. The form includes an emergency type menu (general emergency, fire, transportation), a location, incident description, resource request, and hazardous material fields. 3. The `FieldOfficer` fills the form by specifying minimally the emergency type and description fields. The `FieldOfficer` may also describe possible responses to the emergency situation and request specific resources. Once the form is completed, the `FieldOfficer` submits the form by pressing the "Send Report" button, at which point the `Dispatcher` is notified.
Dispatcher station	4. The `Dispatcher` is notified of a new incident report by a popup dialog. The `Dispatcher` reviews the submitted information and creates an `Incident` in the database by invoking the `OpenIncident` use case. All the information contained in the `FieldOfficer`'s form is automatically included in the `Incident`. The `Dispatcher` selects a response by allocating resources to the `Incident` (with the `AllocateResources` use case) and acknowledges the emergency report by sending a short message to the `FieldOfficer`.
Field Officer station	5. The `FieldOfficer` receives the acknowledgment and the selected response.

Figure 4-8 Refined description for the `ReportEmergency` use case.

4–6 *Below are examples of nonfunctional requirements. Specify which of these requirements are verifiable and which are not.*

- *"The system must be usable."* [not verifiable, no precise definition of "usable"]
- *"The system must provide visual feedback to the user within 1 second of issuing a command."*[verifiable]
- *"The availability of the system must be above 95%."* [verifiable, assuming sufficient resources]
- *"The user interface of the new system should be similar enough to the old system such that users familiar*

with the old system can be easily trained to use the new system." [not verifiable, no precise definition of "easily trained"]

4–7 The need for developing a complete specification may encourage an analyst to write detailed and lengthy documents. Which competing quality of specification (see Table 4-1) may encourage an analyst to keep the specification short?

Consistency. Long documents tend to have much redundancy and are difficult to maintain.

4–8 Maintaining traceability during requirements and subsequent activities is expensive, because of the addition information that must be captured and maintained. What are the benefits of traceability that outweigh this overhead?

- Ability to show that all requirements have been implemented
- Ability to show that all features in the system correspond to a requirement
- Ability to assess the impact of a change
- Ability to track the human source of a requirement

Which of those benefits are directly beneficial to the analyst?

- The last two benefits are beneficial to the analyst as it reduces the effort to change the specification without introducing new errors.

4–9 Explain why multiple-choice questionnaires, as a primary means of extracting information from the user, is not effective for eliciting requirements.

Multiple-choice questionnaires provide possible answers to the questions that are asked. This introduces two problems: first, the analyst must be familiar enough with the application domain to offer a good set of answers for each question. Second, the user is constrained to select these pre-designed answers. It is then preferable that the analyst use other elicitation methods, such as task analysis or unstructured interviews, to gather sufficient knowledge of the application domain and to discover implicit knowledge that the user assumes everybody has. Subsequently, the analyst can use multiple choice questionnaires to confirm a hypothesis or to prioritize certain functionality.

4–10 From your point of view, describe the strengths and weaknesses of users during the requirements elicitation activity. Describe also the strengths and weaknesses of developers during the requirements elicitation activity.

This is an open-ended question. The goal is to have students realize that no one person has the complete problem/solution under control and that it takes much work to gather all available sources of information.

Strengths of users:

- they usually have detailed knowledge of the problem that needs to be solved
- they have detailed knowledge of constraints imposed by the environment on the possible solutions
- they have detailed knowledge of the application domain.

Weaknesses of users:

- they usually have poor knowledge of the possible solutions.
- they usually have poor knowledge of the formal languages for describing the problem or the solutions.

Strengths of developers:

- they have detailed knowledge of different possible solutions
- they have detailed knowledge of formal languages to describe the problem or the possible solutions

Weaknesses of developers:

- they (initially) have poor knowledge of the problem to be solved
- they can make incorrect assumptions about the problem based on their prior knowledge of different problems.

4–11 Briefly define the term "menu." Write your answer on a piece of paper and put it upside down on the table together with the definitions of four other students. Compare all five definitions and discuss any substantial difference.

The goal of this exercise is to make students realize the ambiguity of natural language. The instructor can replace the term "menu" by any other computer-oriented terms whose original meaning is still in use.

4–12 Write the high-level use case `ManageAdvertisement` initiated by the `Advertiser`, and write detailed use cases refining this high-level use case. Consider features that enable an `Advertiser` to upload advertisement banners, to associate keywords with each banner, to subscribe to notices about new tournaments in specific leagues or games, and to monitor the charges and payments made on the advertisement account. Make sure that your use cases are also consistent with the `ARENA` problem statement provided in Figure 4-17.

Note: For all exercises involving writing use cases, the instructor could issue prior to the exercise writing guidelines that the students should comply with. This will reduce the variance in the length, detail, and style of use cases, making it easier to grade the exercise.

Figure 4-9 depicts a use case diagram illustrating a possible decomposition into a high-level use case and three detailed use cases for ManageAdvertisement.

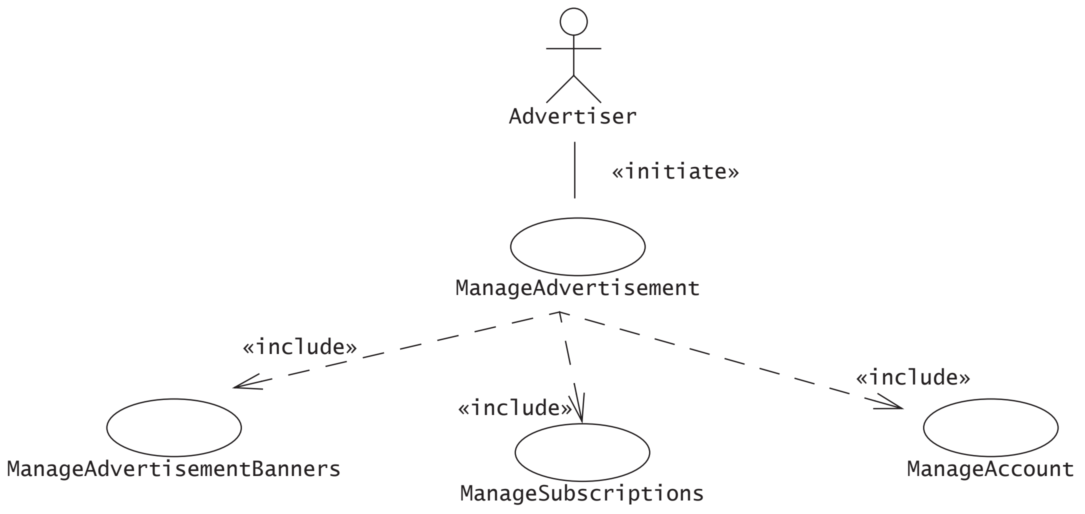

Figure 4-9 Detailed use cases refining the `ManageAdvertisement` high-level use case.

Use case name	`ManageAdvertisement`	
Participating actors	Initiated by `Advertiser`	
Flow of events	1. The `Advertiser` may upload any number of `AdvertisementBanners`, associate keywords with them, and remove previously uploaded `AdvertisementBanners`. (include use case `ManageAdvertisementBanners`).	
		2. `ARENA` displays the current state of the `Advertiser's` account.

Figure 4-10 An example of a high-level the `ManageAdvertisement` use case.

	3. The `Advertiser` may examine the details associated with his account and any charges or payments to the account. (include use case `ManageAccount`).	
		4. Based on the added advertisement banners and keywords, `ARENA` suggests leagues and games the `Advertiser` may subscribe to.
	5. The `Advertiser` may change his subscriptions to leagues and games (include use case `ManageSubscriptions`).	
Entry condition	• The `Advertiser` is logged into ARENA.	
Exit conditions	• New matches can use any newly uploaded banners that match the specified keywords. • The `Advertiser` is notified about any new tournaments in the specified leagues or games.	

Figure 4-10 An example of a high-level the `ManageAdvertisement` use case.

Figure 4-10 depicts a possible high-level use case for `ManageAdvertisement`. This use case should be relatively short and simply include the detailed use cases. Details such as the attributes of the forms involved should not appear at this level.

Figure 4-11 is an example of detailed use case for `ManageAdvertisementBanners`. There are many different other possibilities given the lack of details given in the exercise. Key features should include steps to add, remove, and modify banners, entry, and exit conditions. The use case should specify the attributes of advertisement banners that the `Advertiser` can change.

4–13 Considering the `AnnounceTournament` use case in Figure 4-24, write the event flow, entry conditions, and exit conditions for the use case `ApplyForTournament`, initiated by a `Player` interested in participating in the newly created tournament. Consider also the `ARENA` problem statement provided in Figure 4-17. Write a list of questions for the client when you encounter any alternative.

Figure 4-12 depicts a sample solution. There are, again, many different possibilities depending on the imagination of the students. The key features of a solution include system steps that check for a number of conditions which would make the application fail, and describe the corresponding steps. Given that this use case is relatively short, these actions should be described in the use case as opposed to using extends relationships and many one step use cases. The questions raised by the students should focus on when and how a Player is admitted into (and, correspondingly, thrown out of) a league. Students could also ask about alternative ways for handling too many applications, such as waiting lists, Players taking turns playing in different tournaments, and so on.

4–14 Write the event flows, entry conditions, and exit conditions for the exceptional use cases for the `AnnounceTournament` depicted in Figure 4-25. Use include relationships if necessary to remove redundancy.

This exercise should result in five or more short use cases describing a condition under which the exception occurs and a flow of events for handling it. Figure 4-13 is an example for one such exceptions. The student will discover that exceptional use cases should be used for complex cases, while trivial cases (e.g., name in use, maximum number of tournaments exceeded) could be either omitted or described within the invoking use case. This should to the realization that writing use cases is different than writing code, as the counterpart is the user and the client, not the computer or other developers.

Use case name	`ManageAdvertisementBanners`	
Participating actors	Initiated by `Advertiser`	
	1. The `Advertiser` requests the advertisement banner area.	
		2. `ARENA` displays the list of all banners owned by this actor. The `Advertiser` has the option of uploading new banners (step 3.), editing existing banners (step 7.), or removing banners (step 11).
Flow of events	3. If the `Advertiser` requests the uploading of a new advertisement banner,	
		4. `ARENA` checks if the `Advertiser` exceeded his quote. If not, `ARENA` presents the `Advertiser` with a form.
	5. The `Advertiser` specifies an image, a set of keywords, start and end dates, and a maximum frequency. The `Advertiser` can also specify that the banner can appear in any tournament.	
		6. `ARENA` updates the list of advertisement banners that can be served for each tournament and returns to step 2.
	7. If the `Advertiser` selects an existing banner for editing,	
		8. `ARENA` displays all the details associated with the banner (keywords, start and end dates, maximum frequency)
	9. The `Advertiser` specifies new parameters associated with the selected banners.	
		10. `ARENA` updates the list of advertisement banners that can be served for each Tournament and returns to step 2.
	11. If the `Advertiser` selects to view the details associated with his account,	
		12. `ARENA` displays the current total along with a list of recent charges and payments, including flat fees charged for exclusive sponsorships and charges for each individual banners, after which the `Advertiser` can select to go back to step 1.
Entry condition	• The `Advertiser` is logged into `ARENA`.	
Exit conditions	• New advertisement banners can appear in new matches. • Old banners are retired immediately.	

Figure 4-11 An example of a high-level the `ManageAdvertisement` use case.

Use case name	`ApplyForTournament`
Participating actors	Initiated by `Player`
Flow of events	1. The `Player` requests an application for an announced tournament
	2. `ARENA` checks if the maximum number of applications for this tournament has been reached. If yes, the `Player` is notified that the tournament is full. 3. Otherwise, `ARENA` checks if the player is already registered with the League. If yes, the `Player` is informed that his application was successful, that he will be notified once the matches are scheduled, and the use case completes. 4. Otherwise, `ARENA` checks if the `Player` expertise rating falls within the bounds of the tournament. If so, the `Player` is automatically added to the league, and the `Player` is informed about the acceptance to the league and the tournament, and that he will be notified once the matches are scheduled. 5. Otherwise, the `Player` is informed why he cannot be admitted into the League.
Entry condition	• The `Player` is logged into ARENA.
Exit conditions	• The player is accepted in the tournament OR • The player is informed why the application failed.

Figure 4-12 `ApplyForTournament` use case.

Use case name	`NameInUse`
Participating actors	Initiated by `Player`
Flow of events	1. `ARENA` informs the `LeagueOwner` that the maximum number of open tournament for this league has been exceeded and presents a list of all the open tournaments, sorted by date of completion 2. The `AnnounceTournament` use case completes.
Entry condition	• The maximum number of open tournaments in the current league has been exceeded and the `LeagueOwner` attempts to create a new tournament. • Extends `AnnounceTournament`.
Exit conditions	

Figure 4-13 `NameInUse` use case.