Learning unit 2

Requirements engineering and quality attributes

INTRODUCTION

Choosing the software architecture for a system is the first step towards a system that fulfils the requirements. Software requirements, therefore, are important to the field of software architecture. Software requirements engineering is needed to understand and define what problem needs to be solved. We need to discover, understand, formulate, analyse and agree on what problem should be solved, why such a problem needs to be solved and who should bear the responsibility of solving that problem [35].

In this learning unit, we will discuss the concepts of stakeholder and concern, which are both important with respect to requirements. We will discuss the different types of requirements and show how to categorise them. We will show you different ways to elicit requirements from stakeholders, specify them and validate them. Use cases are often used for specification of software requirements, and we will discuss them in more detail. We will also introduce tactics as a strategy to meet quality requirements.

LEARNING GOALS

After having studied this learning unit, you will be expected to be able to:

- explain the concepts of stakeholder and concerns
- summarise the different types of requirements
- state the steps to be taken in the requirements engineering process
- describe how use cases can be used to specify functional requirements
- describe why concerns have to be prioritised
- describe the relationship between tactics and requirements.

LEARNING CORE

1 Important concepts

1.1 STAKEHOLDERS AND CONCERNS

The concepts of stakeholder and concern are defined in the first learning unit, in definitions 3 and 5.

The following are examples of stakeholders:
- Acquirers are those who decide which system to use.
- Assessors are those who check whether a given system conforms to needs or constraints.
- Communicators are responsible for training and documentation.
- Developers develop the system.
- Contributors develop or write documentation.
- Committers take decisions in the development process.
- Maintainers fix bugs and evolve the system.
- Suppliers provide components.
- Supporters provide help to users.
- System administrators keep the system running, administer users and configure the system.
- Testers test the system or parts of the system.
- Users use it.

Customers
End users

Customers should be distinguished from *end users*. The customer pays for the development; the end user uses the product. The customer is concerned with costs, often to the point of compromising usability. In many cases, it is useful to split some of those groups. Users of a content management system, for instance, may be divided in users who enter content and users who only consume content.

Concern

*Concern*s differ from requirements: requirements are expressed as properties that the system should have, while concerns are expressed from the point of view of a particular stakeholder. A concern may express the *interest* that the stakeholder has in the system. The owner, for example, may have the concern that using the system will increase his or her profit. A concern may also express something that worries the stakeholder. A user, for instance, may have the concern that the system should not be more difficult or cumbersome to use than the current system.

The concerns of different stakeholders are often mutually incompatible. Examples of such incompatible concerns are users who want the system to be fast and easy to use versus an owner who wants maximum security. Sometimes, the concerns of the same stakeholder are incompatible, as in the case of an owner who wants a system that is easy to modify and, at the same time, is built for very low cost. The concerns of the developing organisation are not the same as those of the customer. A customer may have already invested in existing architectures, which may result in constraints with respect to the architecture.

This means that the architecture of a system is usually a compromise or trade-off that takes incompatible concerns into account. It is important

to document and explicitly prioritise concerns. An architecture is the earliest artefact that allows the priorities among competing concerns to be analysed.

Standard architectures and architectural patterns have well-known effects on the quality attributes of a system. This makes it possible to reason about the effect of the chosen architecture on the qualities of the system that are requested by the different stakeholders. Therefore, architectural decisions can be analysed at an early stage (before design starts), in order to determine their impact on the quality attributes of the system. This analysis is worth doing because it is virtually impossible to change the architecture at a later stage of development.

1.2 SOFTWARE REQUIREMENTS

The definition of a software requirement given by the IEEE Standard Glossary of Software Engineering Terminology [30] is:

Definition 10 (Requirement) *A* software requirement *is a condition or capacity needed by a user to solve a problem or achieve an objective.*

Requirements come in three types: functional, non-functional and constraints:

Functional requirements
- *Functional requirements* present what the system should do. The rules of an online game are examples of functional requirements.

Non-functional requirements
- *Non-functional requirements* specify specific qualities the system should work with. A possible non-functional requirement for an online game is that the game should provide an interface that is easy to understand, or that the response to an action of the user should be given within less than a certain maximum time.

Constraints
- *Constraints* show the limits within which the system should be realised. For an online game, a constraint could be that it should work with both Firefox and Internet Explorer, without the need for a plugin.

We will discuss these different types of requirements further on in this learning unit.

1.3 REQUIREMENTS ENGINEERING

Requirements engineering

Requirements engineering is a cyclic process involving [35]:
- domain understanding and elicitation
- evaluation and negotiation
- specification and documentation
- quality assurance: validation and verification.

KAOS method

The KAOS method [35] is a technique for requirements engineering as a whole. The KAOS *method* offers techniques for modelling a systems requirement in the form of goals, conceptual objects, agents, operations, behaviour and the relations between those entities.

Definition 11 (Domain understanding) Domain understanding *means to acquire a good understanding of the domain in which the problem is rooted, and of what the roots of the problem are. [35].*

The following aspects are important for domain understanding:
- the organisation within which the current system (which may or may not already be supported by software) is located (if applicable): its structure, objective and so on
- the scope of the current system
- the set of stakeholders to be involved
- the strengths and weaknesses of the current system.

Definition 12 (Requirements elicitation) Requirements elicitation *is the activity of discovering candidate requirements and assumptions that will shape the system-to-be, based on the weaknesses of the current system as they emerge from domain understanding. [35].*

Techniques for requirements elicitation are:
- asking: interview, (structured) brainstorm, questionnaire
- scenario-based analysis: 'think aloud', use case analysis, storyboards
- ethnography: active observation
- form and document analysis
- knowledge reuse: reuse requirements from similar systems
- starting from an existing system
- prototyping and mock-ups
- following your own insight.

None of these techniques guarantee 'correct' requirements. In fact, it is impossible to formulate what constitutes a correct requirement. It is generally advisable to use more than one technique.

Note that stakeholders have *concerns* that are formulated from their point of view. These concerns have to be translated into *requirements*.

Definition 13 (Requirements evaluation) Requirements evaluation *is making informed decisions about issues raised during the elicitation process. [35].*

Negotiation may be required in order to reach a consensus.
- Conflicting requirements must be identified and resolved.
- The risks associated with the system must be assessed and resolved.
- Alternative options must be compared.
- Requirements must be prioritised.

Definition 14 (Requirements specification) Requirements specification *means rigorous modelling of requirements, to provide formal definitions for various aspects of the system [41].*

A requirements specification document should be:
- as precise as possible: it is the starting point for architecture and design;
- as readable as possible: it should be understandable for the user.

A requirements specification should also preferably be correct, unambiguous, complete, consistent, ranked for importance, verifiable, modifiable and traceable.

Among the techniques used for requirements specification are E-R modeling (Entity-Relationship), the Structured Analysis and Design Technique (SADT), Finite State Machines, use cases and UML.

E-R modelling
- E-R *modelling* is a semantic data modelling technique, developed as an extension to the relational database model (to compensate for its absence of typing and inadequate modelling of relations), not unlike the UML class diagram.

SADT
- SADT was developed in the late 1970s by Ross [52]. It is based on a data-flow model that views the system as a set of interacting activities. As notation it uses rectangles representing system activity, with four arrows. An arrow from the left signifies input, an arrow to the right signifies output, an arrow from above signifies control or database and an arrow from below signifies a mechanism or algorithm.

Finite state machines
- *Finite state machines* show states and transitions, with guards and actions. A finite state machine models the different states the system can be in, where a state is characterised by the actions that are enabled. Finite state machine modelling is part of UML, in the form of state diagrams.

UML
- UML incorporates class models, state diagrams and use cases, but not data flow (data flow diagrams mix static and dynamic information, and are replaced by activity and collaboration diagrams).

The previously mentioned KAOS method uses goal diagrams that are based on UML. Use cases will be treated in more detail below.

Definition 15 (Requirements validation) Requirements validation *is concerned with checking the requirements document for consistency, completeness and accuracy ([33]).*

Requirements should be validated with stakeholders in order to pinpoint inadequacies with respect to actual needs. [35]

Requirements verification

Requirements verification is something else: a mathematical analysis, possibly automated, of formal specifications for consistency. Verification only checks consistency; completeness or accuracy cannot be checked with mathematical analysis. You need user interaction to validate requirements: the requirements document may not reflect the real requirements.

Among the techniques used for requirements validation are:
- reviews (reading, checklists, discussion)

- prototyping (the process of constructing and evaluating working models of a system in order to learn about certain aspects of the required system and/or a potential solution)
- animation (the ability to graphically depict the dynamic behaviour of a system by interactively going through specification fragments to follow some scenario [8]).

2 Use cases

Use cases

Use cases form a technique for specifying functional requirements: use cases are helpful when it comes to specifying what the system should do. A use case captures a contract with the stakeholders of a system about its behaviour. A use case describes the system's behaviour under various conditions, as the system responds to a request from one of the stakeholders, called the primary actor. Use cases are fundamentally a description of usage scenarios in textual form.

The following is an example of a use case [1]:

```
Name
  Withdraw cash

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

Actors
  1 Bank Customer
  2 Bank

Primary Actor
  Bank Customer

Level
  user goal

Stakeholders and Concerns
  Users want to be certain that they can get their money, and
      that they will not lose money. Bank owner wants to be
      certain that no money is lost.

Preconditions
  1 There is an active network connection to the Bank.
  2 The ATM has cash available.

Main Success Scenario
  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 the 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.
```

[1] from: http://epf.eclipse.org/wikis/openup/core.tech.common.extend_supp/guidances/examples/use_case_spec_CD5DD9B1.html

```
 8. Card ID, PIN, amount and account is sent to Bank as a
     transaction. The Bank Consortium replies with a go/no go
     reply, stating whether the transaction is OK.
 9. Money is dispensed.
 10. The Bank Card is returned.
 11. The receipt is printed.
 12. The use case ends successfully.

Extensions
 1 Invalid User
 If, in step 2 of the basic flow, the use case: Validate User is
      not completed successfully, then
  1.1. the use case ends with a failure condition.
 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
  2.1. the ATM shall display the message 'Invalid Account,
       please try again'
  2.2. the use case resumes at step 4.
 3 Wrong amount
 If, in step 7 of the basic flow, the Bank Customer enters an
     amount that cannot be created with the kind of cash in the
     ATM [See Special Requirement WC-1 for valid amounts], then
  3.1. the ATM shall display a message indicating that the
       amount must be a multiple of the bills on hand, and ask
       the Bank Customer to re-enter the amount
  3.2. the use case resumes at step 7.
 4 Amount Exceeds Withdrawal Limit
 If, in step 7 of the basic flow, the Bank Customer enters an
     amount that exceeds the withdrawal limit (See Special
     Requirement WC-2 for the maximum amount), then
  4.1. the ATM shall display a warning message, and ask the Bank
       Customer to re-enter the amount
  4.2. the use case resumes at step 7.
 5 Amount Exceeds Daily Withdrawal Limit
 If, in step 8 of the basic flow, the Bank response indicates
     that the daily withdrawal limit has been exceeded (this is
     determined by the Bank and depends on the specific account)
     , then
  5.1. the ATM shall display a warning message, and ask the Bank
       Customer to re-enter the amount
  5.2. the use case resumes at step 7.
 6 Insufficient Cash
 If, in step 7 of the basic flow, the Bank Customer enters an
     amount that exceeds the amount of cash available in the ATM
     , then
  6.1. the ATM will display a warning message and ask the Bank
       Customer to re-enter the amount
  6.2. the use case resumes at step 7.
 7 No Response from Bank
 If, in step 8 of the basic flow, there is no response from the
     Bank within three seconds, then
  7.1. the ATM will re-try, up to three times
  7.2. if there is still no response from the Bank, the ATM
       shall display the message 'Work unavailable, try again
       later'
  7.3. the ATM shall return the card
  7.4. the ATM shall indicate that it is closed
  7.5. the use case ends with a failure condition.
 8 Money Not Removed
 If, in step 9 of the basic flow, the money is not removed from
     the machine within 15 seconds, then
  8.1. the ATM shall issue a warning sound and display the
       message 'Please remove cash'
  8.2. if there is still no response from the Bank Customer
       within 15 seconds, the ATM will re-tract the money and
```

```
       note the failure in the log
   8.3. the use case ends with a failure condition.
  9 Quit
  If, at any point prior to step 8 of the basic flow, the Bank
      Customer selects Quit, then
    9.1. the ATM shall print a receipt indicating that the
        transaction was cancelled
    9.2. the ATM shall return the card
    9.3. the use case ends.

Postconditions
  1 Successful Completion
  The user has received their cash and the internal logs have
      been updated.
  2 Failure Condition
  The logs have been updated accordingly.

Special Requirements
  [SpReq:WC-1] The ATM shall dispense cash in multiples of $20.
  [SpReq2:WC-2] The maximum individual withdrawal is $500.
  [SpReq:WC-1] The ATM shall keep a log, including date and time,
       of all complete and incomplete transactions with the Bank.
```

The format of a use case description may vary. As you can see, the format for this use case is the following:

```
Name:
   <Preferably start with a verb>
Brief Description:
  <the goal of the use case>
Level:
  <user goal or sub-function>
Actors:
   <the role names of the actors that are involved>
Actor:
   <the one who calls on the system>
Stakeholders and Concerns:
  <those who care about this use case, and why>
Preconditions:
   <must be met before the use case can take place>
Main Success Scenario:
   <the steps in the main case (also called the basic flow)>
Extensions:
  <alternative flows>
Postconditions:
   <conditions that are met after the use case has taken place>
Special Requirements:
   <extra details>
```

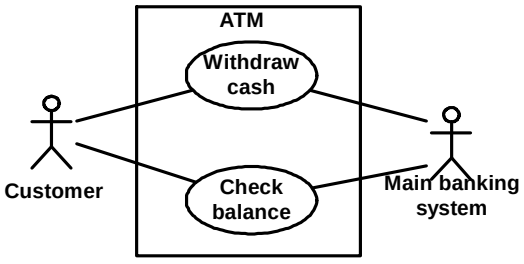

FIGURE 2.1 Use cases for an ATM

Use case diagram

Several use cases may be combined in a *use case diagram*. Figure 2.1 shows an example. The line drawn between an actor and a use case is an association: the participation of an actor in a use case. Instances of the actor and instances of the use case communicate with each other. This is the only relationship between actors and use cases.

An actor can be thought of as a role. In Figure 2.1, we see two actors: the customer and the main banking system. Both actors are associated with two use cases: withdraw cash and check balance.

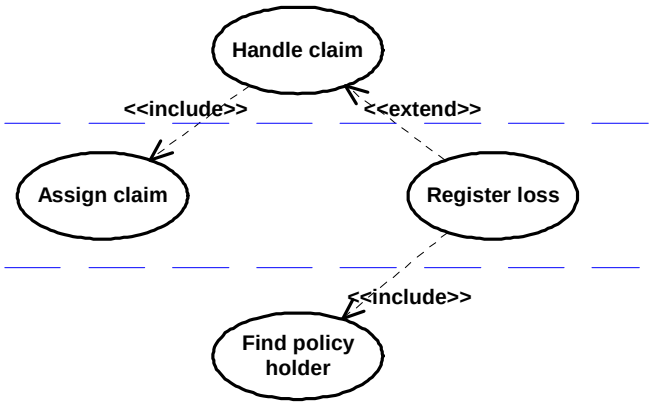

FIGURE 2.2 Different levels in use cases

Use cases can be described at different levels:

Summary level
- A use case at *summary level* involves multiple user goals and shows the context of user goals, the life cycle of user goals or a table of contents. A use case such as `Handle an insurance claim` (see Figure 2.2) is an example of this.

User goal level
- A use case at *user goal level* shows a primary actor's goal in trying to get work done. An example of a use case on the use level is `Register a loss`.

Sub-function
- A *sub-function* is a step that is needed to carry out a user goal. An example is `Find policy holder's file`.

In Figure 2.2, we see three annotations on the relationships between the different levels. These are:

Extend relation
- An *extend relation* from a use case A to a use case B indicates that an instance of use case B may be augmented (subject to specific conditions specified in the extension) by the behaviour specified by A. The behaviour is inserted at the location defined by the extension point in B, which is referenced by the extend relation.

Generalisation
- A *generalisation* from a use case C to a use case D indicates that C is a specialisation of D.

Include relation
- An *include relation* from a use case E to a use case F indicates that an instance of use case E will also contain the behaviour as specified by F. The behaviour is included at the location which is defined in E. (see [45]).

Some advice on how to write use cases:
- A use case is a prose essay. Make the use cases easy to read using active voice and present tense, and describe an actor successfully achieving a goal.
- Include sub-use cases where appropriate.
- Do not assume or describe specifics of the user interface.
- An actor is not the same as an organisational role: an actor is a person, an organisation, or an external system that plays a role in one or more interactions with the system.
- Use UML use case diagrams to visualise relations between actors and use cases or among use cases. Use text to specify use cases themselves!
- It is hard, and important, to keep track of the various use cases.

3 Three types of requirements

3.1 FUNCTIONAL REQUIREMENTS

The following are pitfalls with respect to functional requirements [37]:
- having an undefined or inconsistent system boundary. The system boundary defines the scope of the system: what does or does not belong to the system. The system boundary, therefore, determines which problems the system should solve (and which problems belong to the world outside the system)
- describing use cases from the point of view of the system instead of describing them from the point of view of the actor. The correct point of view is that of the actor
- using inconsistent actor names: actors names should be consistent throughout.
- creating spider webs of actor-to-use case relations: relations between actors and use cases should be clear
- writing long, excessive, or confusing use case specifications that are incomprehensible to the customer: use case descriptions should be clear so that the customer can understand them.

Beware of:
- a 'shopping cart' mentality. Stakeholders often have the tendency to treat requirements as items that can be put into a shopping cart. You should always make clear that every requirement comes at a price
- the 'all requirements are equal' fallacy: architectural requirements must be prioritised to indicate to the architect, or anyone else, which requirements are most important to the finished system. No design trade-offs can be made if all requirements are assigned the same priority
- stakeholders who will not read use case descriptions because they find them too technical or too complicated. It is important to ensure that your stakeholders understand the value of taking the time to understand the descriptions.

3.2 QUALITY REQUIREMENTS

Quality requirements

Quality requirements are the main category of non-functional require-

ments. Quality requirements are important parameters for defining or assessing an architecture. For example, the architecture of a safety-critical system will differ from the architecture of a computer game.

Software quality model

Quality requirements may be specified using a *software quality model*. A software quality model serves as a discussion framework, a scheme that allows users and developers to discuss different kinds of quality, to prioritise different kinds of quality and to check the completeness of quality requirements.

3.2.1 *ISO 25010 quality model*

ISO 25010 quality model

The ISO *25010 quality model* [31] is the international standard quality model. The model classifies software quality in a structured set of factors.

Every factor is divided into a set of sub-characteristics.

- *Functional suitability:*
  - Functional completeness
  - Functional correctness
  - Functional appropriateness
- *Performance efficiency:*
  - Time behaviour
  - Resource utilisation
  - Capacity
- *Compatibility:*
  - Coexistence
  - Interoperability
- *Usability:*
  - Appropriateness, recognisably
  - Learnability
  - Operability
  - User error protection
  - User interface aesthetics
  - Accessibility
- *Reliability:*
  - Maturity
  - Availability
  - Fault tolerance
  - Recoverability
- *Security:*
  - Confidentiality
  - Integrity
  - Non-repudiation
  - Accountability
  - Authenticity

- *Maintainability:*
  - Modularity
  - Reusability
  - Analysability
  - Modifiability
  - Testability

- *Portability:*
  - Adaptability
  - Installability
  - Replaceability

When using ISO 25010 as a software quality model, keep in mind that not all 31 quality attributes are equally important. A requirements engineer should prioritise the requirements.

Quality requirements should be measurable. For example, the requirement 'The system should perform well' is not measurable, but the requirement 'The response time in interactive use is less than 200 ms' is measurable.

3.2.2 *Change scenarios*

Some quality requirements concern other aspects of the system rather than functionality. These quality requirements are mainly attributes from the Maintainability and Portability group, such as Changeability and Adaptability. These requirements cannot be linked to use cases.

Such quality requirements should be linked to specific change scenarios. By doing that, you avoid being vague. For instance, instead of writing 'The system should be very portable', you should write 'The software can be installed on the Windows, Mac and Unix platforms without changing the source code'. Instead of writing 'The system should be changeable', you should write 'Functionality that makes it possible for users to transfer money from savings to a checking account can be added to the ATM within one month'.

3.3 CONSTRAINTS

Constraints

Although functional and quality requirements specify the goal, *constraints* limit the (architectural) solution space. Stakeholders should therefore not only specify requirements, but also constraints.

The following are possible constraint categories:
- technical constraints, such as platform, reuse of existing systems and components, use of standards
- financial constraints, such as budgets
- organisational constraints, such as processes, availability of customer
- time constraints, such as deadlines.

3.4 SUMMARY OF REQUIREMENTS ENGINEERING

To summarise: requirements engineering is a cyclic process involving domain understanding and requirements elicitation, evaluation and negotiation, requirements specification documentation, and quality assurance through requirements validation and verification. Use cases can be used to describe interaction between actors and the system. ISO 25010 can be used to describe the quality requirements linked to usage scenarios (use cases) or change scenarios.

Requirements engineering is necessary to software architects to provide them with descriptions of:
- the scope
- stakeholders, concerns and their relations
- functional requirements
- quality requirements
- a prioritisation of requirements
- constraints.

4 Tactics

Tactic
Architectural patterns

Once determined, the quality requirements provide guidance for architectural decisions. An architectural decision that influences the qualities of the product is sometimes called a *tactic*. Mutually connected tactics are bundled together into *architectural patterns*: schemes for the structural organisation of entire systems. We will discuss patterns in detail in the learning unit about patterns.

Tactics to achieve recoverability

As an example, several *tactics to achieve recoverability* (a sub-characteristic of the Reliability factor in ISO 25010) are shown below:
- *Voting* between processes running on redundant processors. This tactic detects only processor faults, not algorithmic errors.
- In a *hot restart*, only a few processes will experience state loss. In a cold restart, the entire system loses state and is completely reloaded. In a hot restart model, the application saves state information about the current activity of the system. The information is given to the standby component so it is ready to take over quickly. There may be redundant standby components that respond in parallel. The first response is used, the others are discarded.
- *Passive redundancy* means switching to a standby backup on failure. This tactic is often used in databases.
- *Rollback*: a consistent state is recorded in response to specific events, which makes it possible to go back to the recorded state in case of an error. This tactic is often used in databases, or with software installation.

Tactics for changeability

Some *tactics for changeability* (a sub-characteristic of Maintainability) are:
- maintaining *semantic coherence*: high cohesion within every module, loose coupling to other modules
- *hiding information*. Provide public responsibilities through specified interfaces that are maintained when the program evolves

– using an *intermediary*. For example, a data repository uncouples producer and consumer; design patterns such as Facade, Mediator, and Proxy translate syntax. Brokers hide identity.

DISCUSSION QUESTIONS

a *Buildability* is defined as the ease with which a desired system can be constructed in a timely manner. Possible tactics include decomposition into modules with minimal coupling and assignment of modules to parallel development teams. This results in minimisation of construction cost and time. Where in ISO 25010 would you locate this quality?
b Where in ISO 25010 would you locate *Scalability*?
c Can you think of other important qualities that are not explicitly mentioned in these models?
d What tactics would you advocate to promote the *Authenticity* subcharacteristic?
e What tactics would you advocate to promote the *Fault-tolerance* subcharacteristic?
f What tactics would you advocate to promote the *Availability* subcharacteristic?