L8 - Requirements specification issues

CS3028 - Principles of Software Engineering

Ernesto Compatangelo

Department of Computing Science

UNIVERSITY OF ABERDEEN

8.1 Reminding past issues and mapping them to current topics

Where are we now?

Software development paradigms

⇒ The Unified Process (UP) paradigm

⇒ UP phases and UP disciplines (activities) within each phase

⇒ Inception (first UP phase)

⇒ Elaboration (second UP phase)

⇒ Elaboration requirements

⇒ Requirements specification issues

⇒ · · · · · ·

8.2 Focus on actors, scenarios, and use cases

Summarising use cases so far

- **Functional** requirements are user-triggered processes that a system must perform. Often specified as **use cases**.
- **Non-functional** requirements are behavioural properties a system must have (*e.g.*, usability, reliability, performance, supportability). Often grouped under the use case(s) they are associated to.
- Use cases are built following a step-by-step process:
  1. Identify **actors**
  2. Identify **scenarios**
  3. Merge related scenarios into a **use case**
  4. Refine the use cases by adding **extra information**

8.3 Use case writing guidelines

- Use cases should be named with verb phrases. The name of the use case should indicate what the user accomplishes (*e.g.*, `ReportEmergency`, `OpenIncident`).
- Actors should be named with noun phrases (*e.g.*, `FieldOfficer`, `Dispatcher`, `Victim`).
- The system boundary should be clear. Steps accomplished by the actor and steps accomplished by the system should be distinguished (*e.g.*, in a two-column flow of event, system actions are on the right).
- Use case steps in the flow of events should be phrased in the active voice. This makes it explicit who accomplishes the step.
- the casual relationship between successive steps in the flow of events should be clear.
- A use case should describe a complete user transaction (*e.g.*, the expected flow of events).
- Unexpected flows of event (*i.e.*, exceptions) should be described separately.
- A use case should not describe the user interface of the system. This takes away the focus from the actual steps accomplished by the user and is better addressed with visual mock-ups.
- A use case should not exceed ONE page in length. Otherwise, decompose it in smaller use cases (see later).

8.4 Exemplary questions for eliciting non-functional requirements

Category	*Exemplary question*
Usability	• What is the level of expertise of the user? • What user interface standards are familiar to the user? • What documentation should be provided to the user?
Reliability *(including robustness, safety, and security)*	• How reliable, available, and robust should the system be? • Is restarting the system acceptable in the event of a failure? • How much data can the system loose? • How should the system handle exceptions? • Are there safety requirements of the system? • Are there security requirements of the system?
Performance	• How responsive should the system be? • Are any user tasks time critical? • How many concurrent users should it support? • How large is a typical data store for comparable systems? • What is the worse latency that is acceptable to users?
Supportability *(including maintainability and portability)*	• What are the foreseen extensions to the system? • Who maintains the system? • Are there plans to port the system to different software or hardware environments?
Implementation	• Are there constraints on the hardware platform? • Are constraints imposed by the maintenance team? • Are constraints imposed by the testing team?
Interface	• Should the system interact with any existing systems? • How are data exported/imported into the system? • What standards in use by the client should be supported by the system?
Operation	• Who manages the running system?
Packaging	• Who installs the system? • How many installations are foreseen? • Are there time constraints on the installation?
Legal	• How should the system be licensed? • Are any liability issues associated with system failures? • Are any royalties or licensing fees incurred by using specific algorithms or components?

8.4.1 Review questions to ensure requirements are correct

- Is the glossary of entity objects understandable by the user?
- Do abstract classes correspond to user-level concepts?

- Are all descriptions in accordance with the users' definitions?
- Do all entity and boundary objects have meaningful noun phrases as names?
- Do all use cases and control objects have meaningful verb phrases as names?
- Are all error cases described and handled?

8.4.2 Review questions to ensure requirements are complete

- For each *object*, (i) Is it needed by any use case? (ii) In which use case is it either (a) created, (b) modified, or (c) destroyed? (iii) Can it be accessed from a boundary object?
- For each *attribute*, (i) When is it set? (ii) What is its type? (iii) Should it be a qualifier?
- For each *association*, (i) When is it traversed? (ii) Why was the specific multiplicity chosen? (iii) Can associations with one-to-many and many-to-many multiplicities be qualified?
- For each *control object*, (i) Does it have the necessary associations to access the objects participating in its corresponding use case?

8.4.3 Review questions to ensure requirements are consistent

- Are there multiple classes or use cases with the same name?
- Do entities (*e.g.*, use cases, classes, attributes) with similar names denote similar concepts?
- Are there objects with similar attributes and associations that are not in the same generalisation hierarchy?

8.4.4 Review questions to ensure requirements are realistic

- Are there any novel features in the system? If so, were any studies or prototypes built to ensure their feasibility?
- Can the performance and reliability requirements be met?
- Were performance and reliability requirements verified by any prototypes running on the selected hardware?

8.5 A closer look at the initial analysis objects

Initial analysis objects

Use cases specify the functional system view. However, they also

- Describe domain elements which have information associated to them; this information must be processed by the system
- Report domain terms that appear again and again in a requirements specification, making them central to the understanding of the system
- Specify physical elements and artifacts with which the system must interact (people, other systems)

All these elements (namely, the participating objects for the use case) must be unambiguously described in a glossary (data dictionary). They represent the initial objects of requirements analysis.

Heuristics for initial analysis objects identification Consider the following elements:

- Terms that developers or users must clarify to understand the use case
- Recurring nouns in the use cases
- Real-world entities that the system must track
- Real-world processes that the system must track
- Use cases themselves
- Data sources or sinks (*e.g.*, `Printer`)
- Artifacts with which the user interacts

8.5.1 Heuristics for objects-use cases crosschecking

For cross-checking use cases and participating objects, consider:

- Which use cases create this object (*i.e.*, during which use cases are the values of the object attributes entered in the system)?
- Which use cases modify and destroy this object (*i.e.*, which use cases edit or remove this information from the system)?
- Which actor can initiate these use cases?
- Whether this object is needed (*i.e.*, is there at least one use case that depends on this information?

8.6 Requirements Verification & Validation

Requirements Verification & Validation

- **Requirements verification, aka 'building the system right'** is the process of checking whether the software system meets the *specified requirements* elicited from the user (maybe not what s/he had in mind but rather what the elicitator understood)
  Requirements are verified from a system developer viewpoint
- **Requirements validation, aka 'building the right system'** is the process of checking whether the software system meets the *actual* user requirements (which may differ from the specified ones)
  Requirements are validated from a user viewpoint

- Beginning with elaboration, requirements must be **verified** to ensure that the system correctly perform its specified functionalities
- Requirements can be verified if each of them is **traceable**
- The traceability of a requirement means that it can be traced throughout any stage of the software development process to its corresponding system function *and vice versa*
- The traceability is the ability to track the dependencies among requirements, design elements (components, classes, methods, attributes) and corresponding implementation artifacts.
- Traceability enables a tester to assess the coverage of a test case, *i.e.*, which requirements are tested and which are not.

Before moving to construction, requirements must be **validated** to ensure that a system addresses the client's needs in terms of:

- **Completeness**: all features of interest are described by requirements
- **Consistency**: no two requirements in a system specification contradict each other
- **Unambiguity**: a requirement cannot be interpreted in two (or more) different or mutually exclusive ways
- **Correctness**: requirements only describe systemic and environmental features of interest, avoiding any unintended feature

8.7 requirements and different kinds of systems

Software system development paradigms & their limits (1)

- The UP has been created with Information Systems in mind;
- Ditto for software project management methodologies;
- Agile and XP methodologies are fine for information systems, but need a quantitative approach to estimates and planning;
- Open issue: how good/appropriate are current paradigms and methodologies for autonomous "intelligent" systems (*e.g.*, self-driving car autopilots, robots, detection systems . . .)

Software system development paradigms & their limits (2)

- Functional requirements have been created with information systems in mind; they can be inadequate to specify autonomous systems
- Parallel systems (characterised by threads) introduce synchronisation issues that cannot be easily captured by a functional model
- Distributed systems (characterised by clients, servers, peers) introduce delocalisation issues that cannot be easily captured by a functional model - interacting agents are a typical example
- Robotic systems include sensors and actuators; these elements do not comfortably fit in a framework developed for human users or for traditional external databases

User stories vs use cases

- Info sys grow incrementally from large base of existing software
  | Parallel/Distributed/Autonomous systems (PDA sys) currently grow from scratch from small base of existing software
- Info sys need substantial human interaction
  | PDA sys only need marginal human interaction
- Info sys flows of events can be quite long and articulated
  | PSA sys flows of events are generally short and simple
- Use cases are needed in many info sys cases
  | user stories adequate to capture PDA sys reqs in most cases

Requirements and the three-tier architectural pattern

- Info sys interfaces (GUIs) big & complex; many boundary classes
  | PDA sys interfaces (GUIs) smaller & simpler; few boundary classes
- The info sys business logic is not very complex
  | the PDA sys business logic is very complex
- Info sys storage layer uses SQL DB in most cases
  | PDA sys storage layer uses NOSQL DB or file systems with heterogeneous media in many cases

8.8 Preparing for the topic ahead

Next week...

The elaboration models:

- Static model (classes, objects...)
- Dynamic model (events, states...)