BIF-5-SSD

Lecture 11. Requirements Engineering

These lecture notes are partly drawn from chapters 21 to 27 of the set book for this module…

"Software Engineering for Students"
by Douglas Bell, Addison-Wesley, 2005.

It also draws extensively on lecture notes by Martin Bush for the previous module Software Engineering Principles

---

Types of requirements

The requirements for a software system fall into two broad categories: functional and non-functional.

Functional requirements are about *what* the software system provides for carrying out useful real world tasks for the individual, organisation or machine (or combination of these).

Non-functional requirements are about the quality with which the functional requirements are to be provided. That is *how* the system is to operate. Non-functional requirements are often not adequately addressed.

Satzinger et al in "Introduction to Systems Analysis And Design, An Agile Iterative Approach (2012)" outline the 'FURPS' categorisation of requirements with the 'F' denoting functional requirements and the the other letters denoting

- Usability requirements – for example standardising user interaction for similar tasks
- Reliability requirements – the degree to which the system must be available and operate without errors.

- Performance requirements – for example user response time related to number of users in on-line system
- Security requirements – for example user authentication, data integrity and confidentiality in transmission

In addition they add further constraints that form part of the requirements (The extended categorisation is called 'FURPS+'):

- Design constraints: specification of the hardware on which the system is to run
- Implementation requirements – for example the programming language or network protocols to be used
- Interface requirements – how the system interacts with other systems for example RSS feeds or interaction with social media sites.
- Physical requirements – for specialised hardware, this would include size, weight etc.
- Supportability requirements – for example provision of automatic configuration of software to different hardware platforms, upgrading to new versions of software.

What is requirements engineering?

Requirements engineering is about…

- analysing needs & desires ("requirements analysis"),
- considering the feasibility of possible solutions,
- negotiating a reasonable spec. for the solution,
- specifying the solution as clearly as possible,
- validating the specified solution,
- keeping track of the requirements as they are
- incorporated/transformed into the solution.

Developers who specialise in this are often called "systems analysts". Arguably, they have the most crucial job of all, since if the users' needs are not accurately captured the

there is no hope of producing a product that will satisfy them (let alone delight them).

Bell gives the following example of an engineering specification…

> "On a dry track, the locomotive must be able to start a train of up to 100 tonnes on an incline of up to 5% with an acceleration of at least 30 km/h/h."

This says something about what the system must do but nothing about how it might do it. It is very clear, and quantified – which is important for acceptance testing.

Often a specification for a software product will imply nothing about its implementation, but other times a specification will imply something about the implementation, especially if the specification has been derived by studying the details of some existing system.

The written document that the analysts must produce forms the basis of the project, and it also represents the goal of the project.

Sometimes two documents are produced; a statement of requirements that is written in language the customer can understand, and a specification that is written in more precise, technical jargon to avoid any potential misinterpretation by the developers. Bear in mind that the terms requirements and specifications are not used consistently in the literature on software engineering; sometimes they are used interchangeably.

Sometimes the term requirements specification is used to refer to the (single) document resulting from systems analysis.

Good communication is vital. During the requirements analysis process the analysts and users will hopefully have built up a shared understanding of the system requirements, but there may well be other developers who

were not involved in any of the discussions and so won't share this understanding, at least initially.

What do you think of the following specification?

> "Write a Java program to provide a personal telephone directory. It should implement functions to look up a number and to enter a new telephone number. The program should provide a friendly user interface."

Bell says this is not good, since…

- specifying the use of Java is how, not what,
- the two functions are not clearly specified,
- "friendly user interface" is too vague.

Sometimes a specification will deal separately with…

- functional requirements (behaviour),
- data requirements (input, stored, output),
- constraints (economic, technical),
- prioritisation of requirements and attitude to trade-offs (think of time-boxing).

Bell makes a nice distinction between…

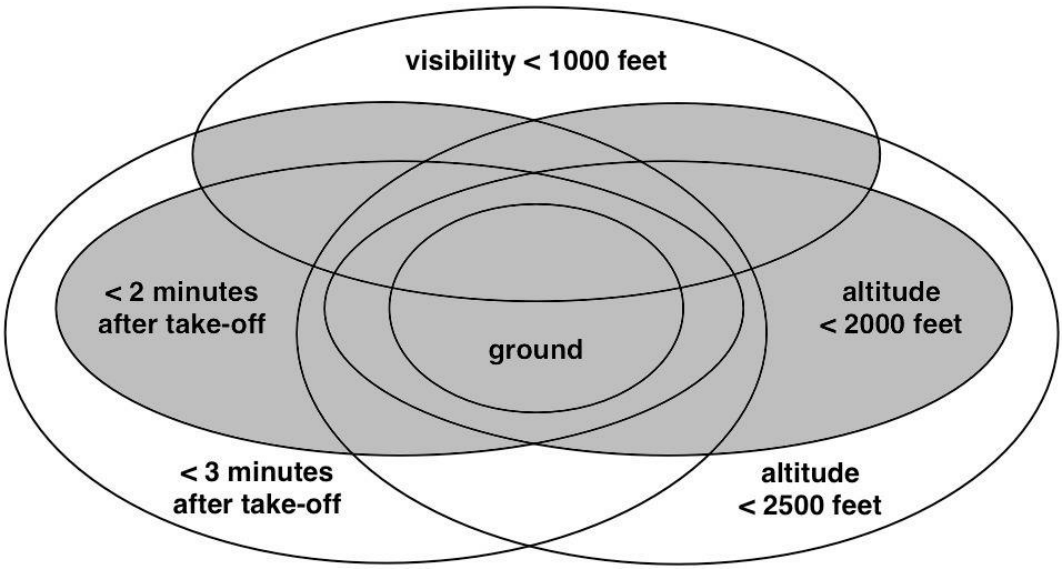
- *listening* ... requirements gathering,
- *thinking* ... requirements analysis,
- *writing* ... requirements definition.

Use of diagrams in requirements/specifications

Diagrams can be very helpful in terms of communicating ideas in an easy-to-understand way, although – just as with natural language – they are often open to misinterpretion. The use of standardised UML diagrams is now common. Examples of these are: use case diagrams, activity diagrams, state diagrams

In practice, a wide variety of diagrams are used to capture system requirements. For example here's a Venn diagram showing the conditions under which the computercontrolled

landing gear within a particular model of airplane either should or shouldn't be in the down positio in readiness for a potential landing…



The darkly shaded area represents *landing gear down*.

The importance of domain knowledge

The analyst(s) should ideally already have a good working knowledge of the system domain. For some domains this is crucial. For example, financial systems (e.g. pensions mortgages, payroll, share dealing) are often very specialised and require analysts who understand the terminology etc.

The concept of a system and its environment

Authors often use the term system boundary to denote the interface between a system that is being constructed and its environment. Sometimes the boundary is clear from the start, but it isn't always. Sometimes the introduction of a new system actually leads to a change in its environment.

Often, the requirements for a new system only become fully evident after the system has been in operation for a while, especially if users begin to adopt different methods

of working because of the new system. This is one way in which the introduction of a new system can actually change its environment.

Think about the UK's NHS (National Health Service) patient records system for example; it enables/demands changes in the way doctors, nurses and other NHS staff do their day-to-day work.

Many computerised systems actually embody some kind of idealised model of their environments, so that they can react correctly to the (expected) inputs from their environments. This is evident particularly within object oriented systems, since objects typically model something in the real world, such as a book in a library, a seat on an airplane, an invoice or a receipt.

The starting point for a software project…

…can vary considerably. The following is from "Object Oriented Software Engineering", Lethbridge & Laganière (McGraw Hill, 2005)

	Requirements need to be determined	Clients already have clear requirements
New development	A	B
Evolution of existing system	C	D

Quadrants B and D relate to some bespoke software for particular clients, while A and C could relate either to bespoke software or COTS software. Most software projects are of type C or D.

Requirements analysis is clearly needed for projects of type A and C, but it also shouldn't be neglected for types B and

D since the software developers must satisfy themselves that the requirements they have been given are clear and reasonable. Requirements engineering is applicable to all four types of project.

Requirements gathering ("requirements elicitation")

- This is the first stage in requirements analysis (and thus the first stage in requirements engineering). It involves…
- identifying the people (usually referred to as users or stakeholders) who will help to specify the requirements, and also understanding their biases,
- defining the technical environment and other constraints into which the product must fit,
- using a variety of methods such as observation, interviews and questionnaires,
- creating use cases and scenarios.

Requirements gathering through observation

One way for analysts to gain insights is by observing key users to see exactly what it is they do. Such observation is best done by "shadowing" the users, but it could also be done by watching videotaped work sessions.

Requirements gathering through interviews

Interviewing potential users is an obvious and widely used technique, but it must be done with care. Here are some guidelines…

- schedule a series of interviews spread out over time,
- allocate plenty of time for each interview,
- use a mix of pre-prepared questions and brainstorms,
- search/ask for alternative sources of information,
- ask interviewees to draw diagrams if they can,
- bear in mind that interviewees may…
  - be feeling nervous and/or intimidated,
  - be concerned about their job security,

- have some kind of hidden agenda.

A good analyst should be able to identify and resolve conflicting customer/user opinions, and advise on technical difficulties and limitations.

A good analyst should also try to think beyond what the customer has ordered, to identify not only…

- explicit requirements: those identified by the customer,

…but also…

- implicit requirements: those that are obvious in some sense although not explicitly identified,
- bonus requirements: features that the customer might not have expected but will (hopefully) be delighted by.

It is often the case that "80% of the user requirements can be solved with 20% of the effort" (Lethbridge & Laganière, 2005). This is one example of the often-quoted "80:20 rule"; it shouldn't be taken too literally, but there's a grain of truth in it.

Requirements elicitation is often tackled by interviewing various users in isolation, but some organisations prefer to arrange organised meetings involving a number of suppliers and users. These are often held at a neutral site. The goal of these meetings is to discuss and agree a set of requirements, and perhaps also to propose elements of a possible solution.

Use Cases, Scenarios and Testing

Analysts often try to clarify requirements by considering use cases, in which a user interacts with the system in order to achieve something. For example in a cash point system, one cash-point user might withdraw a certain amount of money while another user might just want to

check how much money they have left in their account. (Users in this context are often referred to as actors.)

Often there are obvious users and less obvious ones. For example, the bank employee who fills the cash-point with more cash will also have to interact with the system in some way.

Many authors use the term scenarios in relation to use cases. A scenario is normally regarded as referring to a dialogue between a user and the system. According to UML – which is now used very widely within industry – a scenario is just one route through a particular use case.

If possible, it's a good idea to define each of the scenarios that have been identified in terms of sequences of inputs to the system and the resulting outputs from the system. These can then form the basis of acceptance tests. Many experts say that this is indeed the best way to design acceptance tests; they should be designed purely on the basis of the specification, before any of the software is written.

Requirements analysis

Analysts must clarify the requirements – or rather, the set of requirements – and also try to make simplifications by combining similar requirements and eliminating spurious or unnecessary ones. They must address the following questions...

- is each requirement clear (i.e. unambiguous)?
- are all the requirements essential?
- is each requirement achievable and testable?

We have seen the importance of specifying requirements clearly and unambiguously. Many authors also talk about the need for consistency and completeness.

Consistency implies that the requirements should be free of internal conflicts. Careful analysis may reveal a clash

between two or more of the stated requirements, and in that case the requirements are said to contain an inconsistency or contradiction.

Negotiation is often needed to resolve contradictions. It may also be a good idea to seek agreement about the relative importance of different requirements (think of time-boxing again).

To achieve completeness, every conceivable combination or sequence of inputs must be taken into account. This is clearly desirable, but it may not always be achievable.

The best way to ensure high quality, according to many authors, is by inspection. A team of reviewers consisting of both users and developers should look for problems such

as…

- errors of omission or interpretation,
- ambiguities or inconsistencies,
- unrealistic requirements or trade-offs.

Some textbooks deal only with requirements analysis - i.e. the process of eliciting and defining requirements. However, that's only part of the story. Most authors stress the need for requirements engineering, which includes requirements management as well as requirements analysis.

The central issue in requirements management is traceability. Tables such as the following can be drawn up…

- Features vs. Requirements – showing how observable product features relate to requirements.
- Source of Requirements – showing where each requirement originated.
- Subsystems vs. Requirements – showing which subsystems relate to which requirements.
- Interfaces vs. Requirements – "shows how requirements relate to both internal and external system interfaces" (quote from Pressman).

Some authors even propose that the presence of every line of executable source code in the final product should be traceable back to one or more of the requirements.

Requirements Engineering and Agile

The principles of requirements engineering can be applied to each iteration of development using using the agile approach

However the agile approach to system development place considerably less emphasis on formal documentation than the traditional approach. 'User stories' are converted to either throwaway prototpyes or the latest editions of evolutionary prototypes. These can replace requirements documentation.

For small to medium-sized projects where an informal hands-on relationship can be established with the user or stakeholder during development, this can work well but may be more difficult for large scale projects.

In "Software Engineering, A Practitioner's Approach" by Pressman and Maxim (Mc Graw Hill 2005), the authors explain a further problem:

> "Although the agile approach to requirements elicitation is attractive for many software teams, critics argue that a consideration of overall business goals and nonfunctional requirements is often lacking. In some cases, rework is required to accommodate performance and security issues. In addition user stories may not provide a sufficient basis for system evolution over time."