Chapter 4

Requirements Engineering
Topics covered

- Why RE is hard...
- *Functional* and *non-functional* requirements
- *Product* vs. *organizational* vs. *external* requirements
- *Domain* requirements
- Requirements *specification*
  - The Software Requirements Requirements Document
  - Requirements uses
  - *User* vs. *system* requirements specification

(cont’d)
Topics covered (cont’d)

- Requirements engineering processes
  - Requirements elicitation and analysis
  - Requirements validation
  - Requirements management
Requirements engineering (RE)

- The process of eliciting, analyzing, documenting, and validating the services required of a system and the constraints under which it will operate and be developed.

- *Descriptions* of these services and constraints are the *requirement specifications* for the system.
RE is both very hard and critical

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult… No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

– Fred Brooks, “No Silver Bullet…”
Why is RE so hard?

- Difficulty of *anticipation*
- Unknown or conflicting requirements / priorities
  (“I’ll know what I want when I see it.”)
- Conflicts between & among users and procurers
- Fragmented nature of requirements
- Complexity / number of distinct requirements
anticipation [an-tis-uh-pey-shuhn]

nouns, 14th century
1. a prior action that takes into account or forestalls a later action;
2. the act of looking forward;
3. visualization of a future event or state.

We can never know about the days to come, but we think about them anyway…Anticipation, anticipation is makin' me late, Is keepin' me waitin’. -Carly Simon
Why is RE so hard?

- Difficulty of *anticipation*
- **Unknown** or conflicting requirements / priorities
  (“I’ll know what I want when I see it.”)
- Conflicts between & among users and procurers
- Fragmented nature of requirements
- Complexity / number of distinct requirements for large or complex systems
Why is RE so hard? (cont’d)

- Some analogies:
  - Working a *dynamically changing* jigsaw puzzle
  - Blind men describing an elephant
  - Different medical specialists describing an ill patient
Functional versus non-functional requirements

- **Functional requirements** – *services* the system should provide, how it should *react* to particular inputs, or how it should *behave* in particular situations.

- **Non-functional requirements** – *constraints on services or functions* (e.g., response time) or *constraints on development process* (e.g., use of a particular CASE toolset), *standards to be observed*, etc.
Examples of *functional* requirements descriptions

- The user *shall be able to* search either all of the initial set of databases or select a subset from it.
- The system *shall provide* appropriate viewers for the user to read documents in the document store.
- Every order *shall be allocated* a unique identifier (ORDER_ID) which the user *shall be able to* copy to the account’s permanent storage area.

*(Test of function: “We want the product to…” or “The product shall…”)*
Non-functional requirements

- Define **system attributes** (e.g., reliability, response time) and **constraints** (e.g., MTTF $\geq$ 5K transactions, response time $\leq$ 2 seconds).
  - Attributes are often **emergent system properties** – i.e., only observable when the entire system is operational.

- Define **process constraints** (e.g., use of a particular CASE system, programming language, or development method).
Non-functional requirements are not second class requirements

- Non-functional requirements may be just as critical as functional requirements (if not more so). If not met, the system may be useless.
General non-functional classifications

- **Product requirements** – concern product behaviour.

- **Organizational requirements** – derived from policies / procedures in customer’s or developer’s organization (e.g., process constraints).

- **External requirements** – derived from factors external to the product and development / procurement organizations (e.g., legislative requirements).
General non-functional classifications (cont’d)

- Non-functional requirements
  - Product requirements
    - Efficiency requirements
    - Reliability requirements
  - Organizational requirements
    - Portability requirements
    - Interoperability requirements
  - External requirements
    - Ethical requirements
    - Standards requirements
      - Legislative requirements
    - Privacy requirements
      - Safety requirements
  - Usability requirements
  - Performance requirements
  - Space requirements
Examples

- **Product** requirement statement:
  
  4.C.8 It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard Ada character set. APSE = Ada Programming Support Environment

- **Organizational** requirement statement:
  
  9.3.2 The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95.

- **External** requirement statement:
  
  7.6.5 The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.
“GOALS” vs. (verifiable) REQUIREMENTS

- Non-functional requirements may be very difficult to state precisely, and imprecise requirements may be difficult to verify.

- General goals such as “system should be user friendly” or “system should have fast response time” are not verifiable.

- Goals that convey the intentions of users may be helpful to developers, but should be translated into quantitative requirements that can be objectively tested.
Example of system “GOAL” versus verifiable system REQUIREMENT

- A system *goal* statement:
  
  *The system should be easy to use by experienced controllers and should be organized in such a way that user errors are minimized.*

- A (more) verifiable non-functional system *requirement* statement:
  
  *Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.*
### Attribute measures for specifying non-functional requirements

<table>
<thead>
<tr>
<th>Property</th>
<th>Measure</th>
</tr>
</thead>
</table>
| **Speed**      | Processed transactions/second  
User/event response time  
Screen refresh time   |
| **Size**       | Mbytes  
Number of ROM chips    |
| **Ease of use**| Training time  
Number of help frames   |
| **Reliability**| Mean time to failure  
Probability of unavailability  
Rate of failure occurrence  
Availability       |
| **Robustness** | Time to restart after failure  
Percentage of events causing failure  
Probability of data corruption on failure |
| **Portability**| Percentage of target dependent statements  
Number of target systems |
Requirements interactions

- Competing/conflicting requirements are common.
- Spacecraft system example:
  - To minimize weight, the number of chips in the unit should be minimized.
  - To minimize power consumption, low-power chips should be used.
  - But using low-power chips means that more chips have to be used.
- For this reason, preferred points in the solution space should be identified.
Preferred points in a solution space

-Star = feasible solutions

Prefered solutions

Weight constraint

Power Consumption constraint

Weight

low

high

low

high

Power Consumption
Domain requirements

- Domain requirements – requirements *derived from* application domain *rather than the specific needs of users* (e.g., legal requirements or physical laws)
- May be functional or non-functional.
- If domain requirements are not satisfied, the system may be unworkable.
Train protection system domain requirement

- The deceleration of the train shall be computed as:

\[ D_{\text{train}} = D_{\text{control}} + D_{\text{gradient}} \]

where \( D_{\text{gradient}} \) is \( 9.81\text{m/s}^2 \times \text{compensated gradient/alpha} \) and where the values of \( 9.81\text{m/s}^2/\alpha \) are known for different types of trains. \((\text{physical law})\)

- It may be difficult for a non-specialist to understand the implications of this requirement and how it interacts with other requirements.
Domain requirements problems

- **Understandability** – requirements are often expressed in the language of the application domain and may not be understood by software engineers.

- **Implicitness** – domain experts may not communicate such requirements because they are so obvious (to the experts).
Requirements Specification
Specification issues

- Uses and abstractions
- “User” vs. “system” specifications
- The software requirements document
- Natural language vs. PDL’s vs. graphical representations
- “Interface” vs. “operational” specifications
Requirements uses

- Requirements range from being high-level and abstract to detailed and mathematical.
- Inevitable, as requirements serve multiple uses.
  - May be the basis for a bid for a contract – must be open to interpretation;
  - May be the basis for the contract itself – must be defined in detail;
  - May be the basis for design and implementation – must bridge requirements engineering and design activities.
Requirements abstraction (Davis)

“If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.”
Types of requirements specifications

- **User requirements** – statements in *natural language plus diagrams* of system services and constraints. Written primarily for customers.
  
  - Understandable by system users who don’t have detailed technical knowledge.
  
  - Specify *external* system behaviour (plus any user-specified implementation constraints.)
  
  - Utilize *natural language, forms/tables*, and simple, intuitive *diagrams*. 
Types of requirements specifications (cont’d)

- **System requirements** – *structured document* setting out *detailed descriptions* of services and constraints *precisely*.
  - More detailed, precise descriptions of user requirements.
  - May serve as the *basis for a contract*.
  - *Starting point* for system design and implementation.
  - May utilize *different system models* such as object or dataflow.
**User vs. system requirements**

**“User requirement” statement:**

1. The software must provide a means of representing and accessing external files created by other tools.

**Corresponding “System requirements” statements:**

1.1 The user should be provided with facilities to define the type of external files.
1.2 Each external file type may have an associated tool which may be applied to the file.
1.3 Each external file type may be represented as a specific icon on the user’s display.
1.4 Facilities should be provided for the icon representing an external file type to be defined by the user.
1.5 When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.
Requirements specification readers

User requirements

- Client managers
- System end-users
- Client engineers
- Contractor managers
- System architects

System requirements

- System end-users
- Client engineers
- System architects
- Software developers
- Lawyers
The Software Requirements Document

- The Software Requirements Document (a.k.a. an “SRS”) is the official statement of what is required of the system developers.
- Should include both user requirements and system requirements.
- It is NOT a design document. In general, it should set out WHAT the system should do rather than HOW it should do it. But...

(cont’d)
...Some design info may be incorporated in a requirements document since:

- **Sub-systems may be defined** to help structure the requirements. *(Requirements may be grouped by sub-system.)*
- **Interoperability requirements** may constrain the design.
- **Use of a specific design model** may be a requirement.
### The structure of a requirements document

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>This should define the <em>expected readership</em> of the document and describe its version history, including a <em>rationale</em> for the creation of a new version and a <em>summary of the changes</em> made in each version.</td>
</tr>
<tr>
<td>Introduction</td>
<td>This should describe the <em>need for the system</em>. It should briefly describe the system’s functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.</td>
</tr>
<tr>
<td>Glossary</td>
<td>This should define the <em>technical terms</em> used in the document. You should not make assumptions about the experience or expertise of the reader.</td>
</tr>
<tr>
<td>User requirements definition</td>
<td>Here, you describe the <em>services</em> provided for the user. The <em>nonfunctional system requirements</em> should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. <em>Product and process standards</em> that must be followed should be specified.</td>
</tr>
<tr>
<td>System architecture</td>
<td>This chapter should present a <em>high-level overview</em> of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.</td>
</tr>
</tbody>
</table>

(cont’d)
### The structure of a requirements document (cont’d)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System requirements specification</strong></td>
<td>This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.</td>
</tr>
<tr>
<td><strong>System models</strong></td>
<td>This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.</td>
</tr>
<tr>
<td><strong>System evolution</strong></td>
<td>This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.</td>
</tr>
<tr>
<td><strong>Appendices</strong></td>
<td>These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.</td>
</tr>
</tbody>
</table>
Requirements completeness and consistency

- In principle, a requirements specification should be both complete and consistent.
  - Complete – descriptions of all required services and constraints should be included.
  - Consistent – there should be no conflicts or contradictions in the descriptions.

- In practice, it’s nearly impossible to produce a complete and consistent requirements document.
Natural language-based specification

- Both **user** and **system requirements** are generally based on natural language sentences plus other notations such as tables, forms, graphics, etc.

- Natural language is **expressive**, **intuitive** and **universal**, and can therefore normally be understood by users, managers, developers, etc.

- But there are also **potential problems** with conveying requirements information using natural language…
Some potential problems with using natural language

- **Ambiguity** – requirements may be **unclear** or may be interpreted in different ways.
  
  • Consider the term “appropriate viewers” in:
    
    “The system shall provide appropriate viewers for the user to read documents in the document store.”
  
  • Expressing requirements **unambiguously** is difficult without making documents wordy and hard to read.

(cont’d)
Mary had a little lamb heuristic

(From Gause & Weinberg, Quality Before Design)

Mary had a little lamb.

Mary had a little lamb.

Mary had a little lamb.

Mary had a little lamb.

Mary had a little lamb.
Mary conned the trader heuristic

(From Gause & Weinberg, Quality Before Design)
Mary conned the trader heuristic

(From Gause & Weinberg, Quality Before Design)

Mary owned a petite lamb.  

Mary consumed a small amount of lamb.

Mary had a little lamb.

Mary was involved with a young sheep.

Mary conned the trader.
Some potential problems with using natural language (cont’d)

- **Requirements confusion** – functions, constraints, goals, and design info may be mixed-up.
- **Requirements amalgamation** – several different requirements may be expressed together.

Basic problem: the need for different models of / perspectives on requirements.
Guidelines for writing natural language-based requirements

- Adopt a **standard format** and use it for all requirements.
- **Use language in a consistent way.** (E.g., use *shall* for mandatory requirements, *should* for desirable requirements.)
- **Use text highlighting** to emphasize key parts of the requirement.
- Avoid the use of computer (and other types of) *jargon.*
## Alternatives to NL specification

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured natural language</td>
<td>This approach depends on defining standard forms or templates to express the requirements specification.</td>
</tr>
<tr>
<td>Design &quot;PDL’s&quot;</td>
<td>This approach uses a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system.</td>
</tr>
<tr>
<td>Graphical notations</td>
<td>A graphical language, supplemented by text annotations is used to define the functional requirements for the system. An early example of such a graphical language was SADT (Ross, 1977; Schoman and Ross, 1977). More recently, use-case descriptions (Jacobsen, Christerson et al., 1993) have been used. I discuss these in the following chapter.</td>
</tr>
<tr>
<td>Mathematical specifications</td>
<td>These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don’t understand formal specifications and are reluctant to accept it as a system contract. I discuss formal specification in Chapter 27.</td>
</tr>
</tbody>
</table>
Form/template-based specifications

<table>
<thead>
<tr>
<th>Insulin Pump/Control Software/SRS/3.3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong>  Compute insulin dose: safe sugar level.</td>
</tr>
<tr>
<td><strong>Description</strong>  Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.</td>
</tr>
<tr>
<td><strong>Inputs</strong>  Current sugar reading (r2); the previous two readings (r0 and r1).</td>
</tr>
<tr>
<td><strong>Source</strong>  Current sugar reading from sensor. Other readings from memory.</td>
</tr>
<tr>
<td><strong>Outputs</strong>  CompDose—the dose in insulin to be delivered.</td>
</tr>
<tr>
<td><strong>Destination</strong>  Main control loop.</td>
</tr>
</tbody>
</table>

(cont’d)
Form/template-based specifications

**Action**
CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

**Requirements**
Two previous readings so that the rate of change of sugar level can be computed.

**Pre-condition**
The insulin reservoir contains at least the maximum allowed single dose of insulin.

**Post-condition**
r0 is replaced by r1 then r1 is replaced by r2.

**Side effects**
None.
Program Description Languages (PDLs)

- Requirements are specified *operationally* using *pseudocode*.
- Shows what is required via a design example that illustrates how the requirements *could* be satisfied. (NOT how they *should* be satisfied.)
- Especially useful when specifying a process that involves an ordered sequence of actions.

† Also known as a “Design Description Language”
Part of an ATM specification

```java
class ATM {
    // declarations here
    public static void main (String args[]) throws InvalidCard {
        try {
            thisCard.read () ; // may throw InvalidCard exception
            pin = KeyPad.readPin () ; attempts = 1 ;
            while ( !thisCard.pin.equals (pin) & attempts < 4 )
            {
                pin = KeyPad.readPin () ; attempts = attempts + 1 ;
            }
            if (!thisCard.pin.equals (pin))
                throw new InvalidCard ("Bad PIN");
            thisBalance = thisCard.getBalance () ;
            do {
                Screen.prompt (" Please select a service ") ;
                service = Screen.touchKey () ;
                switch (service) {
                    case Services.withdrawalWithReceipt: 
                        receiptRequired = true ;
```
Graphical representations

- Graphical models are particularly useful in describing
  - system environments (context models)
  - data structures and flows (semantic data models / dataflow diagrams)
  - state changes and system responses to events (state machine models)
  - classification and aggregation of system entities (object models)
  - dynamic system behavior (sequence diagrams)
Example: UML “Sequence diagrams”

- These show the sequence of events that take place during some user interaction with a system.
- You read them from top to bottom to see the order of the actions that take place.
- Cash withdrawal from an ATM
  - Validate card
  - Handle request
  - Complete transaction
“Sequence diagram” of ATM withdrawal

ATM

Card

PIN request

PIN

Option menu

<<exception>>
invalid card

Withdraw request

Amount request

Amount

<<exception>>
insufficient cash

Card

Card removed

Cash

Cash removed

Receipt

Database

Card number

Card OK

Validate card

Balance request

Balance

Handle request

Debit (amount)

Debit response

Complete transaction
“Interface specifications”

- Used to specify operating interfaces with other systems.
  - Procedural interfaces (e.g., function, procedure, or method names)
  - Data structures that are exchanged
  - Data representations (if necessary)
- Also used to specify functional behaviour.
  - Formal notations (e.g., pre- and post-conditions) are effective.
PDL-based interface description

interface PrintServer {

// defines an abstract printer server
// requires: interface Printer, interface PrintDoc
// provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter

  void initialize ( Printer p ) ;
  void print ( Printer p, PrintDoc d ) ;
  void displayPrintQueue ( Printer p ) ;
  void cancelPrintJob (Printer p, PrintDoc d) ;
  void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;
} //PrintServer
Interface specification of a simple function using pre- and post-conditions

Function: Set BIG to the largest value in the non-empty array A[1..N].

**pre-condition:** \( N \geq 1 \)

**post-condition:** there exists an \( i \) in \([1,N]\) such that \( \text{BIG}=A[i] \) & for every \( j \) in \([1,N]\), \( \text{BIG} \geq A[j] \) & A is unchanged
Equivalent (pseudocode based) “operational” specification

Function: Set BIG to the largest value in the non-empty array $A[1..N]$. 

\[
\begin{align*}
    & \text{BIG} := A[1] \\
    & i := 2 \\
    & \text{while } i \leq N \text{ do} \\
    & \quad \text{if } A[i] > \text{BIG} \text{ then BIG := A}[i] \text{ end_if} \\
    & \quad i := i+1 \\
    & \text{end_while}
\end{align*}
\]
Agile methods and requirements

- Many agile methods argue that producing a requirements document is a waste of time because requirements change so quickly, it would always be out of date.

- Methods such as XP use *incremental requirements engineering* and express requirements as “user stories” (discussed in Chapter 3).

- But this may be problematic for systems that require a lot of pre-delivery analysis (e.g. critical systems) or systems developed by several teams.
Requirements document variability

- The information in a requirements document depends on the type of system and the development approach used.
- Systems developed incrementally will, typically, have less detail in the requirements document.
- Most of the requirements documents standards (e.g., the IEEE standard) are mostly applicable to large systems engineering projects.
Requirements Engineering Processes
RE processes

- **Vary widely** depending on:
  - Application domain
  - People involved
  - Organization developing the requirements

- **Generic activities common to most:**
  - Requirements elicitation and analysis
  - Requirements validation
  - Requirements management

- In practice, RE is an **iterative activity** in which these processes are **interleaved**.
Elicitation and analysis

- Involves REs working with customers to learn about the application domain, the services needed and the system’s operational constraints, etc.
- May also involve end-users, managers, maintenance personnel, domain experts, trade unions, etc. (That is, other stakeholders.)
Problems of elicitation and analysis

- Getting all, and only, the *right* people involved
- Stakeholders often:
  - don’t know what they really want
  - express requirements in their *own terms*
  - have *conflicting* or *competing* requirements
- Requirements naturally *change* as insight improves. *(Should this be thought of as a *problem*?)

(cont'd)
Problems of elicitation and analysis (cont’d)

- **New stakeholders** may emerge. *(Consider the “railroad paradox”)*.

- **Political** or **organizational** factors may affect requirements.

- The **environment may evolve** during the RE process.
Elicitation and analysis process activities

- **Requirements discovery**
  - Interacting with stakeholders to discover product and domain requirements

- **Requirements classification and organization**
  - Grouping and organizing requirements to facilitate analysis

- **Prioritization and negotiation**
  - Prioritizing requirements and resolving requirements conflicts.

- **Requirements documentation**
  - Requirements are documented and input into the next round of the *spiral*...
Elicitation and Analysis spiral

- Requirements classification and organisation
- Requirements prioritization and negotiation
- Requirements discovery
- Requirements documentation
Viewpoint-oriented elicitation

- There are many *different ways of looking at a problem* ("viewpoints").
- A multi-perspective analysis is important as there is no single correct way to analyze system requirements.
- Provides a natural way to *structure* the elicitation *process* and *organize* requirements.
Types of viewpoints

- **Interaction viewpoints**
  - People or other systems that interact directly with the system.

- **Indirect viewpoints**
  - Stakeholders who do not use the system themselves but who influence the requirements.

- **Domain viewpoints**
  - Domain characteristics and constraints that affect the requirements.
Method-based RE

- "Structured methods" to elicit, analyze, and document requirements.
- A modern example is the Volere\textsuperscript{†} Requirements Process (www.volere.co.uk)
  - Consists of requirements templates, processes, books, consulting, training, etc.
  - Process and templates work with existing tools and methods including agile methods, RUP, etc.

\textsuperscript{†} volere (Italian) – to want
Volere Requirements Process

Start here
## Volere Requirement Shell

<table>
<thead>
<tr>
<th>Requirement #</th>
<th>Unique ID</th>
<th>Requirement Type</th>
<th>Event/Use Case #</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>A one sentence statement of the intention of the requirement</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>A justification of the requirement</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Source</strong></td>
<td>Who raised this requirement?</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fit Criterion</strong></td>
<td>A measurement of the requirement such that it is possible to test if the solution matches the original requirement</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Customer Satisfaction</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Customer Dissatisfaction</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td>A list of other requirements that have some dependency on this one</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Conflicts</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Supporting Materials</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>History</strong></td>
<td>Creation, changes, deletions, etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Other Requirements</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Measure of Stakeholder Happiness</strong></td>
<td>Degree of stakeholder happiness if this requirement is successfully implemented. Scale from 1 = uninterested to 5 = extremely pleased.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Measure of Stakeholder Unhappiness</strong></td>
<td>Measure of stakeholder unhappiness if this requirement is not part of the final product. Scale from 1 = hardly matters to 5 = extremely displeased.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Interviewing

- RE’s meet with stakeholders to discuss the system currently in place and the system to be developed.

- May be:
  - Formal or informal,
  - Closed (with a pre-defined agenda), open (no pre-defined agenda), or a mix.

- Useful for learning how stakeholders might affect or be affected by the system.

(cont'd)
Interviewing (cont’d)

- May be less useful for learning about domain requirements since:
  - RE’s may not understand domain-specific terminology;
  - Stakeholders may not communicate such requirements because they are so obvious (to themselves).

Scenarios

- Depict *examples* or *scripts* of possible system behaviour.
- People often relate to these more readily than to abstract statements of requirements. “Give me an example to help tie the parts together (into a coherent whole).”
- Particularly useful in elucidating *fragmentary, incomplete, or conflicting* requirements.
Scenario elements

1. **System state** at the beginning of the scenario (if relevant)

2. **Sequence of events for a specific case** of some generic task the system is required to accomplish.

3. Any relevant concurrent activities.

4. **System state** at the completion of the scenario.
A simple scenario

\textbf{t0}: The user enters values for input array \textbf{A}. The values are \([1, 23, -4, 7, 19]\).

\textbf{t1}: The user executes program \textbf{MAX}.

\textbf{t2}: The value of variable \textbf{BIG} is 23 and the values of \textbf{A} are \([1, 23, -4, 7, 19]\).

(Compare this to the interface and operational specification examples.)
Use cases

- Graphical notations for representing abstract scenarios in the UML. (UML is the de facto standard for OO Analysis & Design.)
- Identify **actors** in an interaction and describe the interaction itself.
- A set of use-cases should describe all types of interactions with the system.
- **Sequence diagrams** may also be used to show the sequence of event processing.
Library use-cases

Library User → Lending services

Library User → User administration

Supplier → Catalog services

Library Staff → Catalog services

Library Staff → User administration

Library Staff → Lending services
Catalogue management sequence diagram

Bookshop: Supplier

Item: Library Item

Books: Catalog

Cataloguer: Library Staff

Acquire

Catalog Item

New

Dispose

Uncatalog Item

time
Ethnography

- A social scientist **observes** and analyzes **how people actually work**.
- Subjects do not have to **explain** or otherwise articulate what they do.
- **Social and organizational factors** of importance may be observed.
- Ethnographic studies have shown that **work is usually richer and more complex than** suggested by simple system models.
**Focused ethnography**

- Developed during a project studying the air traffic control process.
- **Combines ethnography with prototyping.**
- Prototype development raises issues which focus the ethnographic analysis.
- **Problem with ethnography alone:** it studies existing practices which may not be relevant when a new system is put into place.
Requirements Validation

- attributes
- techniques
Requirements validation

- Concerned with whether or not the requirements define a system that the customer really wants.
- Requirements error costs are high, so **early validation is very important**. (Fixing a requirements error **after delivery** may cost many orders of magnitude more than fixing an error during implementation.)
Requirements attributes

- **Validity**: Does the system provide the functions which best support the customer’s *needs*?
- **Consistency**: Are there any requirements conflicts?
- **Completeness**: Are all functions required by the customer included?
- **Realism**: Can the requirements be implemented given available budget and technology
- **Verifiability**: Can the requirements be *tested*? (More precisely, can the system be tested to determine whether or not the requirements will have been met?)
Requirements validation techniques

- **Requirements reviews / inspections** — systematic manual analysis of the requirements.
- **Prototyping** — using an executable model of the system to check requirements.
- **Test-case generation** — developing tests for requirements to check testability.

*How are requirements validated in Agile Software Development?*
Requirements reviews / inspections

- Regular reviews should be held **while requirements are being formulated**.
- Both *client and contractor staff* should be involved in reviews. (+ other stakeholders)
- Reviews may be *formal* (with completed documents) or *informal*…
- *Good communication between developers, customers and users can resolve problems at an early stage.*
Review check-list†

- **Verifiability**: Is the requirement *testable*?
- **Comprehensibility**: Is the requirement *understandable*?
- **Traceability**: Is the *origin* (*and rationale*) of the requirement clearly stated?
- **Adaptability**: Can the requirement be *changed* with minimum impact on other requirements? *(Especially when change is anticipated!)*

† Attributes to check for in addition to consistency, completeness, etc.
Requirements Management

- Planning considerations
- Change management process
Requirements management...

- ...is the process of understanding and controlling requirements changes -- both during system development and after it goes into use.

- Requirements evolve, priorities change, and new requirements emerge as
  - a better understanding of needs develops, and
  - the business and technical environment of the system changes.
Requirements management planning requires decisions on:

- **Requirements identification** – how requirements will be individually identified.
- **A change management process** – to be followed when analyzing the impact and costs of a proposed change.
- **Traceability policies** – the amount of information about requirements relationships that is maintained.
- **Tool support** – tools range from specialized requirements management systems to spreadsheets and simple database systems.
Change management process

- Applied to all proposed requirements changes.
- Principal stages:
  - **Problem analysis** – analyze identified requirements problem and propose specific change(s).
  - **Change analysis and costing** – assess effects of change on other requirements, etc.
  - **Change implementation** – modify requirements document (+ system design and implementation, as necessary) to reflect the change.
Change management process (cont’d)

- Identified problem
- Problem analysis and change specification
- Change analysis and costing
- Change implementation
- Revised requirements
Key points

- **Requirements** concern *what the system should do* and the *constraints on its operation and implementation*.

- *Functional requirements* are the *services* the system should provide

- *Non-functional requirements* constrain the *system* being developed or the *development process*.

(cont'd)
Key points (cont’d)

- Domain requirements – are functional or non-functional requirements *derived from the application domain rather than the specific needs of users.*
- *User requirements* are statements of system services and constraints, written primarily for *customers.*
- *System requirements* provide more *detailed descriptions* of services and constraints, and may serve as the *basis for a contract.*
Key points (cont’d)

- The **Software Requirements Document** (a.k.a. an “SRS”) is the official statement of what is required of the system developers.
- The **RE process** includes requirements elicitation and analysis, specification, and validation.
- **Elicitation and analysis** involves requirements discovery, classification and organization, prioritization and negotiation, and documentation.

(cont'd)
Key points (cont’d)

- Systems have **multiple stakeholders with different viewpoints and requirements.**
- **Social and organization factors** influence system requirements.
- Requirements **validation** is concerned with checks for **validity, consistency, completeness, realism, and verifiability.**

(cont'd)
Key points (cont’d)

- Business, organizational, and technical changes inevitably lead to **changing requirements**.
- **Requirements management** involves careful planning and a change management process.

(cont'd)
A review of Sommerville’s classifications of requirements

- “Functional” vs. “Non-Functional”
- Within the “Non-Functional” category:
  - “Product” vs. “Organizational” vs. “External” (3 different sources)
  - “Goals” vs. verifiable (non-functional) requirements
- “Domain requirements” (a 4th source; may be “functional” or “non-functional”)
- “User” vs. “System” specifications (different levels and intended uses)
- “Operational” vs. “Interface” specifications
Chapter 4

Requirements Engineering