Refining the System Definition

Requirements Engineering

- Team Skill 1 - Analyzing the problem
- Team Skill 2 - Understanding user needs
- Team Skill 3 - Defining the system
- Team Skill 4 - Managing scope
- Team Skill 5 - Refining the system definition
- Team Skill 6 - Building the right system
Software Requirements Specification

- Move from high level (vision stuff) to more specificity (SRS)
  - No one way
  - Still need to be understood by all stakeholders

SOFTWARE REQUIREMENTS
- Software capability needed by the user to achieve an objective
- A software capability needed to satisfy a contract, standard, specification or other formal document
Software Requirements Specification

- First place to look
  - Boundary
    - Inputs
    - Outputs
    - Functions of the system – mapping of inputs and outputs and their combinations
    - Attributes of the system (Dependability)
    - Attributes of the system environment – loads, compatibility etc.

Features and Requirements

- Features
  - Simple descriptions of useful behavior

- Software Requirements
  - Specific statements of software functionality to implement the feature – usually several SR to one feature/user need
  - What versus how
    - No project stuff
      - Schedule
      - Budget
      - Tests
    - No design specs
Software Requirements

- **Iterative**
  - Discovery, definition, design decision
    - Current requirements cause certain design decisions
    - Design decisions develop new requirements

- **Types of requirements**
  - Functional software requirements
    - How the system behaves

- **Non-functional software requirements**
  - Usability
  - Reliability
  - Performance
  - Supportability

- **Constraints**
  - Impose limitations on the design of the system, or process by which it is developed, that do not affect the external behavior

- **Sources**
  - Development infrastructure
    - Operating environments/compatibility/application standards/best practices
    - Regulations and standards

- **Segregate constraints**
  - Distinguish name DC
  - Special section
  - Fully reference
  - Document rationale
NFRs vs. FRs

- **Appearance**
  - FRs: phrased with subject/predicate constructions, or noun/verb. “The system prints invoices.”
  - NFRs: found in adverbs or modifying clauses. “The system prints invoices *quickly*.”

- **IEEE**
  - **FR**
    - Specifies a function that a SW or SW component must be capable of performing.
    - Defines behavior of the system
    - What a system will do
  - **NFR**
    - How the SW will do, for example: SW performance req., SW external interface req., SW design constraints, and SW quality attributes.
    - Difficult to test; therefore, they are usually evaluated subjectively.
NFRs

- NFRs, the qualitative aspects, are as important as functional requirements.
- The user may judge a product on how usable, convenient or secure it is.
- How well a product meets its NFRs may be the difference between an accepted, well-liked project and an unused one.

NFRs ignored

- Late in 1999, NASA’s Mars Climate Orbiter and Polar Lander were lost in Mars’ atmosphere due to inadequate attention paid to:
  - Interoperability
    - Orbiter was developed by separate companies:
      - Lockheed Martin Aeronautics (main contractor)
      - Jet Propulsion Laboratory (JPL)
    - In calculating thrust for the unit
      - LMA used English measurements
      - JPL used the metric system
  - Testability
    - During system testing, one module was excluded (there was a known problem with that module). At a later time, the excluded module was tested independently and final delivery was made.

- Cost: $319M
Examples of NFRs

- Product-Oriented Attributes
  - Performance: throughput, response time
  - Usability: easy to learn and use
  - Efficiency: minimum use of computing resource
  - Reliability: meantime between failures, availability
  - Security: prevent unauthorized access
  - Robustness: works in invalid input, fault, stress
  - Adaptability: reusable in other environment
  - Scalability: works with large data sets
  - Cost: cheap to buy, install and operate

- Family-Oriented Attributes
  - Portability: easily modified to work on different platform
  - Modifiability: easily extended to new features
  - Reusability: reuse design and components in other systems
Examples of NFRs

- Process-Oriented Attributes
  - Maintainability: easily modified
  - Readability: document and code easy to read and understandable
  - Testability: easy to test that the product meets its requirements
  - Understandability: design, architecture, and code are easy to learn
  - Complexity: level of interaction among modules

Family- & Process-Oriented NFRs

- Software developers tend to require family-oriented and process-oriented attributes.
- These attributes have to do with
  - Saving money
  - Making money
Other NFRs

- What about “fun”? 
- Is “fun” an attribute?

Measurable NFRs

- Eventually, the product will need to be tested to see if it meets its requirements.
- FRs can be defined precisely, thus, are testable.
- What about these NFRs?
Specifications of Quantified Req.

- **Performance** -- response time, throughput, capacity
- **Usability** --
  - Easy to learn: time to master certain tasks, amount of training before being able to use product to a certain standard, number of calls to help-desk
  - Users like it: rate at which users start using it, how much users use it, approval rating
  - Error rates allowed
- **Efficiency** -- minimum load on resources
- **Reliability** -- mean-time to failure

Specifications of Quantified Req.

- **Security** -- categories of users, percentage of attacks are successful
- **Robustness** -- percentage of failure on invalid input; specified degraded service (minimum performance under heavy loads, services in presence of faults, how long product must withstand stressful conditions)
- **Adaptability** -- time to learn how to apply product to new problem
- **Scalability** -- size of input data sets
- **Cost** -- maximum cost to buy, install and operate
- **Portability** -- meantime to port product to new platform
Specifications of Quantified Req.

- Modifiability -- meantime to add features.
- Reusability -- % -age of design and data structures reused
- Maintainability -- meantime to fix bugs,; meantime to add features.
- Readability -- Reading Ease Score
- Testability -- percentage of requirements that can be tested; how
- Understandability-- meantime for new developer to add certain features
- Integrity -- meantime to integrate with new adjacent systems

Quantified Req.: Performance

- Response time:
  - Stimulus: an action performed by the user/environment
  - Response: an action generated by the system.
    - s-r : “the system will generate a dial tone within 10 secs from the time the phone is picked up”
    - r-r : “the system will record that the phone is in use no later than 1 micro-second after it and generated a dial tone
    - s-s : “the user will type her password within 15 secs from typing her login name”
    - r-s : “the user will start dialing the phone number within 1 minute from getting the dial tone”
Quantified Req.: Efficiency

- Refers to the level at which a software system uses scarce computational resources, such as CPU, memory, disk.

Quantified Req.: Efficiency

- Can be characterized along a number of dimensions:
  - Capacity: max number of users/terminals/transactions
  - Degradation of service: what happens when a system with capacity $X$ widgets per time unit receives $X+1$ widgets?
    - Let the system crash
    - Let the system handle the load, work slowly
Quantified Req.: Reliability

- The ability of the system to behave consistently in a user-acceptable manner when operating within the environment for which the system was intended.

Reliability: Estimate bugs

- Reliability requirements may take this form:
  - The software shall have no more than X bugs/1K LOC
- Based on a Monte Carlo technique for statistical analysis of random events
  - Plant a known number of errors into the program, which the testing team does not know about
  - Compare the number of seeded errors the team detects with the number of total errors it detects
  - Arrive at an estimate of the total number of bugs in the program
Reliability: Estimate bugs

\[
\frac{\text{#detected seeded err.}}{\text{# detected errors}} = \frac{\text{#seeded bugs}}{\text{# bugs in program}}
\]

- Not an entirely accurate count of bugs in the program
  - the difficulty of seeded bugs
  - Bugs are not created equal
  - Fixing bugs may create more bugs
Quantified Req.: Usability

- All users will be satisfied with the usability of the product.
- 95% of the users will be able to complete representative tasks without requiring assistance (e.g., modifying exclusion date set)
- 95% of the user will be able to complete tasks X Y Z in less than 10 minutes without requiring assistance
- 95% of the users will be able to complete tasks X Y Z by the third attempt without requiring assistance

Qualified Req.

- Product-Oriented Attributes have received lots of attention.
- What about family- and process-oriented Attributes?
Ranking NFRs Requirements

- Many NFRs conflict with one another.
- It is impossible to build a product that maximizes both attributes.
  - security vs. performance
  - performance vs. reuse
  - maintainability vs. robustness
  - performance vs. portability
  - robustness vs. testability

Design Constraints

- International, National and Local laws. – e.g. “The rules and regulations of the Kingdom on ethics and practices of internet use”
- Hardware / Software limitations. – e.g. “the application must run on both new and old platforms”, “Use our C++ coding standards”
- Interfaces to other systems – e.g. “Compatibility with the legacy data base must be maintained”, “Use other corporate data bases to generate reports.”
- A word of caution: These design constraints are different from the Design issues of the Design Team. These are a part of requirements of the stakeholders which are relayed to the Design team
Parent-Child Requirements

- Parent child relationship
  - Parent feature
    - Child
    - grandchild

Refining the Use Cases

- When are Use Cases the best choice?
  - When the system is functionally oriented e.g. a B2B or B2C e-commerce project
  - OO based projects
Refining the Use Cases

- When are Use Cases not the best choice?
  - Systems with few or no users and minimal interfaces 
    e.g. scientific calculations, simulations, embedded systems, process control systems
  - Systems dominated primarily by non-functional requirements and design constraints 
    e.g. embedded systems and process control systems

Use Case Refinements

- Use Case Definition
  - Use case name
  - Brief description
  - Flow of events
  - Alternative flow of events
  - Preconditions
  - Post conditions
  - Special requirements
Criteria to Define System Requirements

- Consistent
- Complete
- Feasible
- Required
- Accurate
- Traceable
- Verifiable

The Process of Requirements Discovery

- Problem discovery and analysis
- Requirements discovery
- Documenting and analyzing requirements
- Requirements management
Documenting Requirements Using Use Cases

- A use case is a behaviorally related sequence of steps (a scenario), both automated and manual for the purpose of completing a single business task.
- An actor represents anything that needs to interact with the system to exchange information. An actor is a user, a role, which could be an external system as well as a person.
- A temporal event is a system event that is triggered by time.

Benefits of Using Use Cases

- Facilitates user involvement.
- A view of the desired system’s functionality from an external person’s viewpoint.
- An effective tool for validating requirements.
- An effective communication tool.
### Example of a High-Level Use Case

**Author:** S. Shepard  
**Date:** 03/01/200

<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th>New Member Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actors:</td>
<td>Member</td>
</tr>
<tr>
<td>Description:</td>
<td>This use case describes the process of a member submitting an order for SoundStage products. On completion, the member will be sent a notification that the order was accepted.</td>
</tr>
</tbody>
</table>

### A Modern SRS Package

- Rational RequisitePro
SRS Template

1. Introduction
  1.1 Purpose
  1.2 Scope
  1.3 Definitions, Acronyms and Abbreviations
  1.4 References
  1.5 Overview

2. Overall Description

3. Specific Requirements
  3.1 Functionality
    3.1.1 <Functional Requirement One>
  3.2 Usability
    3.2.1 <Usability Requirement One>
  3.3 Reliability
    3.3.1 <Reliability Requirement One>
  3.4 Performance
    3.4.1 <Performance Requirement One>
  3.5 Supportability
    3.5.1 <Supportability Requirement One>
  3.6 Design Constraints
    3.6.1 <Design Constraint One>
  3.7 Online User Documentation and Help System Requirements
  3.8 Purchased Components
  3.9 Interfaces
    3.9.1 User Interfaces
    3.9.2 Hardware Interfaces
    3.9.3 Software Interfaces
    3.9.4 Communications Interfaces
  3.10 Licensing Requirements
  3.11 Legal, Copyright and Other Notices
  3.12 Applicable Standards

4. Supporting Information

---

An Ambiguous Requirements Statement

**Requirement:**
Create a means to transport a single individual from home to place of work.

- **Management Interpretation**
- **IT Interpretation**
- **User Interpretation**
Requirement Quality (IEEE 830)

- Correct
- Unambiguous
- Consistent
- Complete
- Ranked for importance and stability
- Understandable
- Verifiable
- Traceable
- Modifiable

Req. Quality: Correct

- An SRS is correct if and only if every requirements stated therein represents something required of the system to be built.
- The correctness will be verified only by review and acceptance by the user.
Req. Quality: Unambiguous

- A requirement is unambiguous if and only if it can be subject to *only one* interpretation

Req. Quality: Unambiguous

- Use *formal* specification techniques
- Techniques presented by Gause and Weinberg
  - Recall what’s in the memory
    - Parts that are not clear and cannot be easily remembered are likely to be the most ambiguous
  - Keyword technique
    - Identify the *key operational words* in a statement and to list all their definitions
    - Mix and match to determine different interpretations
  - Emphasis technique
    - Emphasize individual words until as many different interpretations as possible have been discovered
  - Other techniques: graphics, pictures
Req. Quality: Consistent

- Terms mean the *same* thing every time
- No *conflicts* among requirements

Consistent: the Same Thing

- **Glossary**: Key to achieve consistency of terms
  - Commonly accepted terms often create the biggest headaches later in the development cycle.
    - Do notes, letters, and correspondence differ in the project?
    - Get agreement on the meanings and write them down.
  - The meaning of terms that seem commonly understood may change as personnel on the project change.
  - The definition accepted when a req. was written may shift after.
- **Acronyms**
  - commonly understood in the business
  - Define them for newcomers to the project.
- Avoid **vague** words
  - Vague words inject confusion, such as: manage / track / handle / flag
- Find more **descriptive** words
Consistent: Conflict

- Inconsistent when addressing the same subject
  - e.g.: one req. ask for blue backgrounds, another for red.
  - *Key word and phrase searches* can help locate requirements consistency problems, but only if the terminology is consistent and the storage method allows it.
  - The *spelling and capitalization* of terms and phrases should be consistent to ensure accurate searches.
  - The organization of req. will also affect the ability to find these problems.

- **Conceptual** inconsistency
  - e.g.: a requirement addressing staircases and another restricting the building to one level.
  - *Key word and phrase searches* may not locate these conflicts.

- **Diagrams**
  - can also be used to locate inconsistencies.

Req. Quality: Complete

- A set of requirements is complete if and only if it describes *all* significant requirements of concern to the user; including requirements associated with functionality, performance, design constraints, attributes, or external interface.
Req. Quality: Complete

- Non-functional requirements
  - create a simple checklist follows the guidelines and covers the right questions to ask in the search for design constraints.

- Functional requirements
  - ask the user
    - sometimes, the functionality is so “obvious” the user is not even consciously aware of it.

- Prototyping
  - Check the requirement with the user

Req. Quality: Understandable

- Organized in a manner that facilitates overall review

- Written in a style that facilitates review by the affected parties
Understandable: Organized

- Organized by readers
  - SRD will be read by different kinds of readers, which have different viewpoints for the same project.

- Organized by work process or scenario
  - Allows the subject matter expert to see if there is a "hole" in the work that needs to be done.

- Organized by abstraction level
  - System-level requirements will be used by everyone, while other functional requirements may be used by only one part of the development team.

- Separate NFRs from FRs

Understandable: Style

- Template
  - one way to ensure all SRD have the same style.
  - Various organizations, such as IEEE, have created standards for writing SRD.
  - A typical format is: The system shall [verb] [object of verb].

- Forbid weasel words
  - SRD is a statement of work to be performed
  - Words such as "can," "may," and "should" make implementation of it optional, potentially increasing cost and schedule or reducing quality.
  - If it isn't that important, associate a low priority rating with the requirement and save it for later.
Req. Quality: Ranking for importance and stability

- state the priority of requirement and if it’s going to change
- particularly important when resources are insufficient to implement all requirements within the allotted schedule and budget
- Assign additional attributes to each requirement

Req. Quality: Verifiable

- Able to prove the requirement can be tested
**Req. Quality: Verifiable**

- Problem words and phrases example
  - **Most**: Meaning is subject to interpretation
  - **Some**: Meaning is subject to interpretation
  - **All**: Testing forever to prove it. List the items included
  - **Not**: State from a positive viewpoint
  - **Most adverbs**: "quickly" and "user friendly." Meaning is subject to interpretation
  - **Most adjectives**: "robust". Meaning is subject to interpretation
  - **Conjunctions**: Review any requirements with "or" and "but" to see if the requirement can be interpreted in more than one way
  - **Modifiers**: Words and phrases such as "as appropriate," "if practical," and "at a minimum." Meaning is subject to interpretation
  - **Lack of specificity**: Avoid words such as: "etc.," "and/or," "TBD." Avoid passive voice and pronouns with no reference
- Either rewrite the requirement or find proxies for it

**Req. Quality: Traceable**

- Each product can be *associated* with its *originator* and vice versa
Req. Quality: Traceable

- **Compound** requirements are difficult to trace and may cause the product to fail testing.
  - For example: "the system shall calculate retirement annuities and survivor benefits."
  - If the first build of the system only includes retirement annuities, it will fail the requirement during testing.
  - Once the requirement is marked as having passed testing, the record is not immediately clear which part was passed.
- It is much simpler to *break* the requirement into two.

Req. Quality: Modifiable

- **storage method**
  - affects modifiability
    - For example, requirements in a word processor may be more difficult to modify than in a requirements management tool. But, for a very small project, the cost and learning curve for the RM tool may make the word processor a better choice
- **simplify changes**
  - Templates and glossaries for writing requirements make global changes possible.
- **avoid changes**
  - Templates should exclude information that will be associated with the requirement
    - For example, "the Calculation Module shall determine the gross amount of annuity," must be reworded if the name of the Calculation Module is changed or annuity calculating moved to another module. This change will be in addition to the changes required to the information associating the requirement with the Calculation Module
Req. Quality: Use-case Model

- Several questions may be asked to measure the quality of use case
  - Have all of the use cases been found?
  - Do the use cases meet all the functional requirements?
  - By studying the use-case model, can you form a clear idea of the system’s functions and how they are related?
  - Is each concrete use case involved with at least one actor?
  - Is each use case independent of the others?
  - Do any use cases have very similar behaviors or flows of event?
  - Should the flow of events of one use case be inserted into the flow of events of another?
  - Do customers understand the names and descriptions of the use cases?
  - Is it clear how and when the use case’s flow of events starts and ends?

Req. Quality: Use-case Model

- Questions (continue):
  - Have you found all the actors?
  - Is each actor involved with at least one use case?
  - Do any actors play similar roles in relation to the system?
  - Will a particular actor use the system in several completely different ways, or does the actor have several completely different purposes for using the use case?
  - ...
Req. Quality: Modern SRS Package

- quality features specific to the containing package
- A SRS not only organizes and captures the requirements, but also must be *easy to use*
- A great SRS contains:
  - a good table of contents
  - a good index
  - a revision history
  - a glossary

Table of Contents

- An absolutely *mandatory* feature of a good SRS
  - Nudges the author to use helpful headings
  - Guide reading
- Easy to create
- Keep it *up-to-date*, which can reflect the current package
- Might be difficult to prepare due to parts may be prepared by another tool
Index

- An *important* element of every SRS
- More difficult to create
  - The author must *identify* key elements for indexing
  - Varying views that the project team maintains
- Point the readers to *concepts* rather than to titles.
- Updated to ensure a consistent package

Revision History

- Should *be included in every* SRS to captures the relevant *changes* to each version
- At least, it should include
  - The revision number or code for each change
  - The date of each revision
  - A short summary of the revisions
  - The name of the person responsible for the changes
- Providing markers on each changed elements in the document
- Document changes after settling down
Glossary

- A special language, or a shorthand developed for certain application domain
- Helps users understand this special language.
- Think in terms of including and defining all project-specific terms, all acronyms, all abbreviations and any special phrases.

Key points

- Measurements of quality
- How to build a good SRD
- Checklists can be used to ensure the quality of use-case model
- A high-quality modern SRS package has a good TOC, a good index, a revision history, and a glossary