Chapter 3

Requirements

([FS04] ch.2)

From design, idea, to a software system...

1. Project scope
2. List of requirements: identifies the functionality that lies within the scope

Done by software engineers, directed towards developers.

Communication medium: software requirements specification document.

Tools to develop a list of requirements help ensure that it is complete, accurate, verifiable, and worded so that developers can easily understand them.

Other document: test plan.

Topics:

1. Identify what counts as software requirements
2. Establish criteria for making sense of and writing requirements
3. Determine ways to collect or discover requirements
4. Explore how to employ use cases to refine and analyze requirements
5. Find techniques to confirm that requirements are complete
6. Verify the accuracy of requirements
7. Trace changes to the requirements

3.1 What are requirements?

Concisely worded statement that describes how the software will behave when it is complete. In other words, everytime a requirement is implemented, the system *shall* behave in accordance to the conditions that the requirements establish.

- Functional requirement addresses the operations that the system performs (behavior).
- Nonfunctional requirement applies to the standards or qualities of performance that constrain the design or operations of the system.

Requirements (what) are not expressed in the language of implementation (what). Functionality of the system addressed without reference to implementation: external, or user-oriented perspective.

Example: "The user shall click a dialog button to open a new window" vs. "After the user selects the next level option, the next level appears."

Requirements engineering: translate features into functionality. Features come from a variety of sources (many people, some documents created by others).

Software Requirements Analyst (requirements engineering = requirements analysis): elicits, analyzes, specifies, verifies, and manages the functional and nonfunctional requirements of the software system.

3.2 Why requirements?

Engineering involves stating and solving a problem. Requirements is how the problem is stated. (engineer vs. hacker/performance artist)

- Solving the problem: problem must be clearly and precisely stated
- Planning: without precise specification of all behaviors, impossible to evaluate the time needed to implement the system
- Testing: without complete list of behaviors to be supported, impossible to test whether a system implemented actually behaves as it is supposed to
- Extension
- Cost (efficiency of implementation)
- Maintainability
- Process improvement (meet CMM)

3.3 Software Requirements Specification

Software Requirements Specification (SRS) document: list of formally stated requirements. May also combine other approaches to requirements specification (e.g. in [FS04] add case-driven specification).

IEEE provides a document template for SRS (IEEE standard 830), that serves as a starting point to be tailored to a specific project:

- Table of Contents
- Introduction
  - Purpose
  - Scope (general constraints)
  - Definitions, acronyms and abbreviations
  - References
  - Overview (diagram, subsystems organization and interactions)
- Overall description (optional)
  - Product perspective (refer to design document, describe genre of the game, brief description of the game mechanics)
  - Product functions (break down into different subsystems-or modules)
  - User characteristics (profile the player of the game)
  - Constraints
  - Assumptions and Dependencies (DirectX, OpenGL, Win32, etc.)
- Specific requirements
  - External interface requirements
    - User interfaces (keyboard, mouse, joystick, gamepad, etc.; screens and menus, possibly from design document)
    - Hardware interfaces (modem, joystick, audio, video, etc.)
    - Software interfaces (software dependencies: OS, libraries, etc.)
    - Communications interfaces (modem)
  - **Functional requirements** (break down requirements according to subsystems or components to which they apply)

        - Subsystem A (state requirements, number each requirement)
        - Subsystem B
        - etc.
    - Performance requirements
        - Standards (for specific markets)
        - Hardware limitations
    - Design constraints
        - Availability (for networked/distributed systems)
        - Security
        - Maintainability (how to upgrade, fix, etc.)
    - Other requirements
- Appendices (additional documents)
- Index

3.4 Engineering Requirements

Main delivery of the requirements engineering process is Software Requirements Specification (SRS).

Engineering is an iterative and incremental process, so is requirements engineering. The proposed approach (not a methodology) defines four increments (or phases), iterated as many times as necessary (or possible)..

The four phases are:

- **Elicitation**: gather information about requirements from the design document and other sources; create a first draft of the SRS.
- **Exploration**: create a candidate list of use cases (game context use case, sixty second scenario); create an initial list of requirements; update the SRS
- **Analysis**: Develop use cases based on candidate list; test initial list of requirements for completeness and validity; add use cases and requirements; generate a test case for each use case.
- **Refinement**: Prioritize the requirements; create a requirements matrix; refine requirements language; refine test cases; refine SRS.

15th IEEE International Requirements Engineering Conference (Dehli, India, October 2007). Theme: Understanding Requirements in the Global Economy

"As software development is now part of the global economy, requirements engineering is the key bridge between the customer and supplier. Understanding and translating users' needs into effective solutions has always been vital: however, as development is outsourced requirements have to reflect cultures and languages and local needs. Furthermore, understanding requirements becomes a collaborative activity across time and space."

Topics of interest include, but are not restricted to:

- requirements elicitation, analysis, documentation, validation and verification
- requirements specification languages, methods, processes, and tools
- requirements management, traceability, viewpoints, prioritization, and negotiation
- modelling of requirements (formal and informal), goals, and domains
- prototyping, simulation, and animation
- interaction between requirements and design
- evolution of requirements over time, product families, and variability

- relating requirements to business goals, products, architecture, and testing
- social, cultural, global, personal and cognitive factors in requirements engineering
- collaborative requirements engineering
- domain-specific problems, experiences and solutions

3.4.1 Elicitation

Gather information about requirements from the design document and other sources; create a first draft of the SRS.

- Game description: outer scope (genre, mechanics, etc.) - High level
- Play narratives (sixty-second-of-play narratives): inner scope (what it is like to play the game) - Detailed

3.4.2 Exploration

Create a candidate list of use cases (game context use case, sixty second scenario); create an initial list of requirements; update the SRS
develop use cases (start from game narratives)
Use cases

- narratives: informal paragraph that tells a story about how the user interacts with the system
- scenarios: more formal, narrative broken down into a numbered sequence of events
- diagrams (UML)

Template:

- Use case name
- Requirement(s) explored
- Player (actor) context (role)
- Precondition(s)
- Trigger(s)
- Main course of action
- Alternate course(s) of action
- Exceptional course(s) of action

Develop gtame context use cases (outer scope) and sixty-seconds-of-play use cases (inner scope).
Concurrently develop use cases and list of requirements.
Expess rquire,emts in active voice, use the verb”shall,” and make them pertain to only one feature at a time. Use only one of 2 subjects: “player” and “software.” Avoid wording that conveys assumptions about implementation.

3.4.3 Analysis

Develop use cases based on candidate list; test initial list of requirements for completeness and validity; add use cases and requirements; generate a test case for each use case.
Subject candidate requirements to procedures that reveal weaknesses, redundancies and gaps.
Develop more use case diagrams.scenarios and activity diagrams.

3.4.4 Refinement, Verification and Validation

Prioritize the requirements; create a requirements matrix; refine requirements language; refine test cases; refine SRS.

3.4.5 Verification

3.4.6 Management

3.5 Unified Modeling Language

Object Management Group's (OMG) Unified Modeling Language (UML; *www.uml.org*). Current UML specification: version 2.0

3.5.1 Introduction to OMG's Unified Modeling Language (UML)

(Excerpts from *www.omg.org/gettingstarted/what_is_uml.htm*)

Modeling is the designing of software applications before coding. Modeling is an Essential Part of large software projects, and helpful to medium and even small projects as well. A model plays the analogous role in software development that blueprints and other plans (site maps, elevations, physical models) play in the building of a skyscraper. Using a model, those responsible for a software development project's success can assure themselves that business functionality is complete and correct, end-user needs are met, and program design supports requirements for scalability, robustness, security, extendibility, and other characteristics, before implementation in code renders changes difficult and expensive to make. Surveys show that large software projects have a huge probability of failure - in fact, it's more likely that a large software application will fail to meet all of its requirements on time and on budget than that it will succeed. If you're running one of these projects, you need to do all you can to increase the odds for success, and modeling is the only way to visualize your design and check it against requirements before your crew starts to code.

The OMG's Unified Modeling Language (UML) helps you specify, visualize, and document models of software systems, including their structure and design, in a way that meets all of these requirements. (You can use UML for business modeling and modeling of other non-software systems too.) Using any one of the large number of UML-based tools on the market, you can analyze your future application's requirements and design a solution that meets them, representing the results using UML 2.0's thirteen standard diagram types.

You can model just about any type of application, running on any type and combination of hardware, operating system, programming language, and network, in UML. Its flexibility lets you model distributed applications that use just about any middleware on the market. Built upon fundamental OO concepts including class and operation, it's a natural fit for object-oriented languages and environments such as C++, Java, and the recent C#, but you can use it to model non-OO applications as well in, for example, Fortran, VB, or COBOL. UML Profiles (that is, subsets of UML tailored for specific purposes) help you model Transactional, Real-time, and Fault-Tolerant systems in a natural way.

You can do other useful things with UML too: For example, some tools analyze existing source code (or, some claim, object code!) and reverse-engineer it into a set of UML diagrams. Another example: Some tools on the market execute UML models, typically in one of two ways: Some tools execute your model interpretively in a way that lets you confirm that it really does what you want, but without the scalability and speed that you'll need in your deployed application. Other tools (typically designed to work only within a restricted application domain such as telecommunications or finance) generate program language code from UML, producing most of a bug-free, deployable application that runs quickly if the code generator incorporates best-practice scalable patterns for, e.g., transactional database operations or other common program tasks. (OMG members are working on a specification for Executable UML now.) Our final entry in this category: A number of tools on the market generate Test and Verification Suites from UML models.

What can you Model with UML? UML 2.0 defines thirteen types of diagrams, divided into three categories: Six diagram types represent static application structure; three represent general types of behavior; and four represent different aspects of interactions:

- **Structure Diagrams** include the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.
- **Behavior Diagrams** include the Use Case Diagram (used by some methodologies during requirements gathering); Activity Diagram, and State Machine Diagram.
- **Interaction Diagrams**, all derived from the more general Behavior Diagram, include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

3.5.2 Use case diagrams

3.5.3 Activity diagram

3.5.4 State machine diagram

3.5.5 Communication diagram

3.6 Recommendations

Care not to create incomplete, spotty, redundant, or inaccurate requirements; seek to find the essential behavior of the system and to specify this behavior in nontechnical, clearly stated terms.

- Establish the scope of the project: boundary of the engineering effort, from design document (game overview). Inner scope vs. outer scope
- Identify the customers (stakeholders): the players are the end users. But stakeholders of requirements engineering also include game designers, graphics, music and sound effects designers, voice recorders, etc. who expect that the software system will give life to what they have contributed.
- Feasibility: 3 major risks are number of features, amount of money (resources), quality of the product.
- Uncontrolled growth
  - feature creep: at requirements specification time, go beyond established scope; at implementation time, developers add unspecified functionality
  - goldplating: developers develop the technology they want to develop rather than the product the requirements specify

Eight basic qualities that make for good requirements specifications:

- Make requirements complete
- Make requirements correct
- Necessary requirements only
- Consider the feasibility
- Prioritize requirements (critical path vs. secondary)
- Eliminate ambiguity
- Verify and validate requirements
- Manage evolution