Lecture 18: Specifications

- What is a Specification
  - Purpose
  - Audience
  - Different specs for different project types
- Criteria for good specifications
  - clarity, consistency, completeness
  - measurable & traceable
  - operational vs. definitional specs
- Standards for specifications
  - IEEE standard for SRS

What is a Specification?

- A specification is an agreement…
  - …between the producer of a service…
  - …and the consumer of that service

	Agreement between	
	Producer ⟺	Consumer
Requirements Specification	Development Contractor ⟺	Purchaser
Design Specification	Implementor ⟺	System architect
Module specification	Programmer writing the module ⟺	Programmer using the module

- Software must be specified *precisely*…
  - …if there is a danger of misunderstanding (or forgetting) the consumer's needs
  - …if more than one person's needs are represented
  - …if more than one person will be developing the software

Uses of specifications

- Statement of user needs
  - communicates understanding of those needs to everyone involved
  - acts as a check that the *real* needs have been captured
    - must be understandable by the owners of those needs!
- Statement of implementation constraints
  - a point of reference for the developers
  - can be used to justify development goals and resources
- Documentation of a product
  - a point of reference for product maintainers
    - must be updated when the product is updated
  - baseline for change requests
- A legal contract
  - a point of reference for verification and certification
    - must be possible to determine whether the specification was met
    - must be updated whenever changes are negotiated

Choosing appropriate Spec types

Source: Adapted from Blum 1992, p154-5

- Consider two different projects:
  - A) Small project, 1 programmer, 6 months
    - programmer talks to customer, then writes up a 5-page memo
  - B) Large project, 50 programmers, 2 years
    - team of analysts model the requirements, then document them in a 500-page SRS

	Project A	Project B
Purpose of spec?	Crystalizes programmer's understanding; feedback to customer	Build-to document; must contain enough detail for all the programmers
Management view?	Spec is irrelevant; have already allocated resources	Will use the spec to estimate resource needs and plan the development
Readers?	**Primary**: Spec author; **Secondary**: Customer	**Primary**: all programmers + V&V team; **Secondary**: managers, customers

Desiderata for Specifications

Source: Adapted from the IEEE-STD-830-1993. See also van Vliet 1999, pp225-226

- **Valid (or "correct")**
  - expresses actual requirements
- **Complete**
  - Specifies all the things the system must do
  - ...and all the things it must not do!
  - Responses to all classes of input
  - Structural completeness, and no TBDs!!
- **Consistent**
  - doesn't contradict itself (i.e. is satisfiable)
  - Uses all terms consistently
  - Note: timing and logic are especially prone to inconsistency
- **Necessary**
  - doesn't contain anything that isn't "required"
- **Unambiguous**
  - every statement can be read in exactly one way
  - define confusing terms in a glossary
- **Verifiable**
  - a process exists to test satisfaction of each requirement
  - *"every requirement is specified behaviorally"*
- **Understandable (Clear)**
  - by non-computer specialists
- **Modifiable**
  - Carefully organized, with minimal redundancy
  - Traceable!

Restrictiveness vs. Generality

- **Specificand Sets**
  - A specification describes a set of acceptable behaviours
  - The set of all implementations that meet a specification are its specificand set
  - There are always an infinite number of possible implementations
  - E.g.

```
procedure foo(x: int) returns y:int
effects: x = y
```

This specification is trivial, but its specificand set is still infinite!

- **Restrictiveness:**
  - a specification should rule out any implementation that is unacceptable to its users
- **Generality:**
  - a specification should be general enough so that few of the acceptable implementations are excluded.
    - In particular the more desirable (e.g. elegant, efficient) implementations should not be excluded.
    - Examine every condition in the spec and ask if it's really needed

Ambiguity

- **Is this ambiguous?**

> "The system shall report to the operator all faults that originate in critical functions or that occur during execution of a critical sequence and for which there is no fault recovery response."

- **Can test by trying to translate it:**

Originate in critical functions	F	T	F	T	F	T	F	T
Occur during critical seqeunce	F	F	T	T	F	F	T	T
No fault recovery response	F	F	F	F	T	T	T	T
Report to operator?	?	?	?	?	?	?	?	?

  - if you get different answers from different people, then it is ambiguous.

Consistency and Completeness

- **Consistency:**
  - an inconsistent specification contradicts itself and therefore cannot be satisfied
  - inconsistency may depend on context:

> The text should be kept in lines of equal length specified by the user. Spaces should be inserted between words to keep the line lengths equal. A line break should only occur at the end of a word

  - In practice, inconsistency is hard to test for.
- **Completeness**
  - internally complete:
    - all terms are defined
    - no TBDs
  - Complete with respect to the requirements
    - i.e. describes all services needed by the users
  - In practice, completeness is nearly impossible to achieve
    - aim for balance between generality and restrictiveness

Operational vs. Definitional

- An *Operational* Specification…
  - describes an abstraction in terms of its intended behaviour
    - by describing how it might work

e.g.

```
procedure search(list a, int x) returns int
effects: examines each element of a in turn and returns
  the index of the first one that is equal to x.
signals: NOT_IN if it reaches the end of the list
  without finding x.
```

- A *Declarative* Specification…
  - describes an abstraction in terms of the desired properties of the implementation
    - by describing some properties it must obey

e.g.

```
procedure search(list a, int x) returns int
effects: returns i such that a[i]=x;
signals: NOT_IN if there is no such i.
```

- Declarative specifications are better
  - More general (less implementation bias)
  - Easier to verify

Modifiability and Traceability

- Modifiability
  - well-structured, indexed, cross-referenced, etc.
  - redundancy reduces modifiability
    - avoid or clearly mark as such
  - An SRS is not modifiable if it is not traceable...
- Traceability
  - Backwards: each requirement traces to a source
    - e.g. a requirement in the system spec; a stakeholder; etc
  - Forwards: each requirement traces to parts of the design that satisfy that requirement
  - Note: traceability links are two-way; hence other documents must trace into the SRS
    - Every requirement must have a unique label.
- Useful Annotations
  - E.g. relative necessity and relative stability

IEEE Standard for SRS

Source: Adapted from IEEE-STD-830-1993. See also, van Vliet 1999, pp226-231

1 Introduction
- Purpose
- Scope
- Definitions, acronyms, abbreviations
- Reference documents
- Overview

2 Overall Description
- Product perspective
- Product functions
- User characteristics
- Constraints
- Assumptions and Dependencies

3 Specific Requirements

Appendices

Index

- Identifies the product, & application domain
- Describes contents and structure of the remainder of the SRS
- Describes all external interfaces: system, user, hardware, software; also operations, site adaptation, and hardware constraints
- Summary of major functions
- Anything that will limit the developer's options (e.g. regulations, reliability, criticality, hardware limitations, parallelism, etc)
- All the requirements go in here (I.e. this is the body of the document). IEEE STD provides 8 different templates for this section

IEEE STD Section 3 (example)

Source: Adapted from IEEE-STD-830-1993. See also, Blum 1992, p160

3.1 External Interface Requirements
- 3.1.1 User Interfaces
- 3.1.2 Hardware Interfaces
- 3.1.3 Software Interfaces
- 3.1.4 Communication Interfaces

3.2 Functional Requirements

this section organized by mode, user class, feature, etc. For example:

- 3.2.1 Mode 1
  - *3.2.1.1 Functional Requirement 1.1*
  - …
- 3.2.2 Mode 2
  - *3.2.1.1 Functional Requirement 1.1*
  - …
- …
- 3.2.2 Mode n
  - …

3.3 Performance Requirements

Remember to state this in measurable terms!

3.4 Design Constraints
- *3.4.1 Standards compliance*
- *3.4.2 Hardware limitations*
- *etc.*

3.5 Software System Attributes
- 3.5.1 Reliability
- 3.5.2 Availability
- 3.5.3 Security
- 3.5.4 Maintainability
- 3.5.5 Portability

3.6 Other Requirements

References

van Vliet, H. "Software Engineering: Principles and Practice (2nd Edition)" Wiley, 1999.

Section 9.2 covers most of the material in this lecture, and gives a good introduction to the IEEE standards.

IEEE-STD-830-1993

Is the current IEEE standard that covers software specifications. It is available electronically through the IEEE electronic library (access via U of T library website for the campus-wide subscription)

Blum, B. "Software Engineering: A Holistic View". Oxford University Press, 1992

Provides some additional insights into how to write good specifications.