Requirements

Software Engineering
Computer Science 520/620
Spring 2013
Prof. Leon Osterweil

Today’s Problem
Using some kind of approach like one of the following
“Classical” Waterfall

- Earliest design and test

- order -- “what shall we do next?”
- transition criteria -- "how long shall we do it?"

Modified Waterfall

- Recognition of feedback loops
  - Confined to successive stages
- “Build it twice”
  - Early prototyping
The Rational Unified Process

New semantics to show roles of agents

Boehm's Spiral Model

new semantics to show roles of agents
Different Traversals of the Spiral Model to address different risks

- Application requirements low risk; budget, schedule = high risk
- Stable application requirements & budget; errors = high risk
- Application requirements high risk; budget, schedule = low risk

Some Extreme Programming (XP) Examples

- Test-first programming
- Pair programming
- Scrum
- Etc.
Our Approach

- What is the goal/role of each component type?
- What is the nature of it?
  - Eg. what internal structure does it have?
- What sorts of stakeholders are interested in it?
- What sorts of questions do they generally have about it?
- What sorts of relations must it participate in?
  - Internal relations
  - External relations
- What sorts of processes deal with it?
Requirements Spec.

Characteristics of System to be built must match required characteristics

Test Results must match required behavior

Test Plan

Design

Hi Level consistent views

Low level

Hi level design must show HOW requirements can be met

Code

Code must implement design

Test plan exercises this code

Test Plan

This is the anchor
Some Papers in Requirements Engineering


Requirements: Goals and Purposes

Why “do requirements”?

- Clarify needs before plunging into design
  - Customer “knows” what is wanted
  - But usually doesn't know how to say it
  - Weak sense of what can be achieved
- Clarify acceptance criteria
  - How to know it really delivers what was wanted
- Serve as guide to developers, testers, customers, maintainers
  - “Baselining” requirements
Requirements Spec.

Characteristics of
System to be
built must
match required
characteristics

Test Results must
match required behavior

Design

Test Plan

Requirements Spec.

Functional
Safety
Robustness
Performance
Accuracy

Design

Modules
Components
Constraints
Design Decisions
Components
Characteristics of system to be built must match required characteristics.
Requirements Specification Driven By Stakeholders and their Questions

- Customers
  - What must it do?
- Developers (eg. designers)
  - What do I have to get it to do?
- Testers
  - What is it supposed to be doing?
  - How would I know it if I saw it?
- Users
  - What is it supposed to do?
- Regulators
  - Is enough being required?
- Others???
Requirements Specification Parts

Help stakeholders organize their thoughts about needs by decomposing requirements specification into categories needs and desires.

Some examples:

- Functional

Some examples:

- Introduction/Background
- Functional
- Environmental
- Performance
- Accuracy
- Robustness
- Security
- Safety
Background/Introduction

Purpose: Give background/context out of which the problem arises, and directions in which it is likely to go

Should contain glossary, references

Should give intuition about problem, domain, existing solutions, components

Probably best written mostly in natural language

Example: UMass has 20,000 students, slow growth next few years
Semester system
Existing system that works, but is not great
Define: FTE, fulltime load, etc.

Functional

Purpose: Indicate the functional transformations that the system will have to compute

Likely to be large and complex, therefore aids to easier and clearer comprehension are needed (eg. hierarchy)

Important to state WHAT the functions are and not HOW they are to be computed

Promising formalisms: dataflow diagrams, FSM’s

Usually the chief focus of a requirements specification, and of requirements formalisms—but non-functional requirements are often at least as important
Environmental

Purpose: Indicate the environment in which the software will have to operate

- On which hardware and software will the software run?
- With which other applications will it have to interface?
- What will be the nature of the user community it will have to support?
- With what other manual and automated systems will it have to interface correctly?

Example: System is to be interactive
Most users to be students
Must run using cellphones, PDAs
Must print reports on existing forms (?)
Must interface successfully with existing student and administrative databases

Performance

Purpose: Specify how much computer and human resource can be allocated to support the execution of the software

How much computer memory can the software use?

How fast must response time be?
--average case
--worst case

How long will users wait for batch runs to terminate?

Example: 2 second response time
overnight printing of all reports
128 Mbytes available on PDAs
500 GBytes of disk available
**Accuracy**

**Purpose:** Specify how much tolerance (if any) is acceptable in the results

Most important in numerical computations, but...

Often where "optimality" is defined
eg: what is a "good" game of chess?

**Example:** Reject scheduling constraints that cause more than 10% of all student requests to be denied

---

**Robustness**

**Purpose:** Specify what sorts of abuse the software will have to resist, and how it will respond

What kinds of "illegals" inputs might be expected, and what should be done about them?

Must the system fail safe, fail soft? When?

What abnormal environmental conditions might be expected?

**Example:** System must never corrupt any database --even after a crash
System must deny illegal requests politely
System must not crash due to --lack of storage
--user overload
Security

Purpose: Specify which users can do which things, and when

Usually there are classes of users--what are they?

How to distinguish among the users?

Matrix (?) to specify what accesses and permissions different classes and users will have?

Example: Students cannot:
--change course assignments
--cancel courses
--access data on others

Faculty cannot:
--cancel courses
--change course assignments

Faculty can: access any student data

Administrators can: ... do pretty much anything...

Safety

Purpose: Specify what hazards must be avoided

Specify what the software must NEVER be allowed to do

Has some elements of an inverse or negated set of requirements

Example: System must never corrupt student information database, or faculty personnel database

System must never divulge address or data to unauthorized parties
Not All Go Into All Requirements Specs.

- Some of these may be omitted; some emphasized/deemphasized
- Other sorts of requirements may be added/substituted eg: reliability, flexibility, portability
- Requirements specification provides information needed to satisfy needs of all stakeholders
- Different stakeholder mixes determine choices of what goes into the requirements spec.

SOME EXAMPLES OF THESE UNDERLYING NEEDS:

- Communication
- Testability
- Precision
- Clarity
- Completeness
- Changeability

Requirement Specification Challenges

- Is it **Complete**? (to the extent required)
  - Ultimately impossible to be sure about this
- Is it **Consistent**? (no internal contradictions)
  - Many possible interpretations of this
- Is it un**ambiguous**? (possible multiple interpretations)
- Is it sufficiently **precise**?
  - It is possible to be too precise too
- Is it **Feasible**?
  - If it asks the impossible it would be good to know it
- Is it **Even**? (consistent levels of detail)
- Is it **Understandable**? (what does that mean?)
  - by all stakeholder groups!
- Is there an implementation bias?
- Is there a good basis for proceeding to design?
A Requirement Specification Is Never Perfect in All (Any?) Aspects

- Imperfections are often understandable, tolerable, unavoidable
- Look at real underlying stakeholder needs for the requirements specification (communication, clarity, precision, modifiability....??)
- Plan requirements content, structure, relations to meet these needs
- Requirements specification medium is crucial in helping assure needs are met
- Select requirements specification medium to address needs

Evaluation of Requirements Specification Media

Representative Evaluation Criteria:

- UNAMBIGUOUSNESS
- CLARITY
- COMPLETENESS
- VERIFIABILITY
- CONSISTENCY
- MODIFIABILITY
- LIFECYCLE UTILITY
### Evaluation of Requirements Specification Media

**Representative Evaluation Criteria:**
- UNAMBIGUOUSNESS
- CLARITY
- COMPLETENESS
- VERIFIABILITY
- CONSISTENCY
- MODIFIABILITY
- LIFECYCLE UTILITY

**Example Specification Media:**
- NATURAL LANGUAGE
- STRUCTURED NATURAL LANGUAGE
- DIAGRAMS/CHARTS (DFD’s, FSA’s, Petri Nets)
- FORMAL APPROACHES
- COMBINATIONS OF THE ABOVE

---

**How Do They Match Up with Each other?**

---
Natural Language Prose
Requirements Specification

• Write requirements in "plain English"
• Build upon universal base of understanding of natural language
• Possible to augment with defined terms
• Use of punctuation for clarification
• Text and word processing systems help automate/maintain/alter

Examples:

All input data sets will be terminated with an end of file record
System will respond to service requests within 2 seconds
System will have a friendly user interface
System will never go into an infinite loop

Problem: How to reason about a natural language reqts. spec?
   How to determine: completeness, unambiguity, etc.?

Disciplined Use of Natural Language

• Natural response to problems of:
  --imprecision
  --ambiguity
  --consistency (especially when due to size)

• Familiar approaches:
  --Restricted use of defined terms
  --Introduction of structuring (paragraph numbering, outline form, templates, etc.)

• Other, earlier examples of disciplined use of natural language:
  --Legal documents
  --Recipes
Data Base Approaches

- Requirement items stored as database entries
- Queries to retrieve information
- Database tools to check for consistency

PSL (Relational Database Organization)

DESCRIPTION:
this process performs those actions needed to interpret
time cards to produce a pay statement for each hourly employee.;

KEYWORDS: independent;

ATTRIBUTES ARE:
 complexity-level high;

GENERATES: pay-statement, error-listing;

RECEIVES: time-card;

SUBPARTS ARE: hourly-paycheck-validation, hourly-emp-update,
h-report-entry-generates, hourly-paycheck-production;

PART OF: payroll-processing;

DERIVES: pay-statement;

USING: time-card, hourly-employee-record;

DERIVES: hourly-employee-report;

USING: time-card, hourly-employee-record;

DERIVES: error-listing;

USING: time-card, hourly-employee-record;

PROCEDURE: <<not usually included in a requirements spec.>>

HAPPENS: number-of-payments TIMES-PER pay-period;

TRIGGERED BY: hourly-emp-processing-event;

TERMINATION-CAUSES: new-employee-processing-event;

SECURITY IS: company-only;
Examples:

- Use of structure and reserve words
  
  User interaction functions;
  **Timing:** All functions must execute in < 2 seconds
  
  **Subfunctions:**
  - Query
  - Browse
  - Enter

- Disciplined use of naming

  ....*input_value*: *pay_rate*

  *pay_rate*: *input_to* .....
Functional Decomposition Rqts. DAG

Requirement Element: An Example Structure
Example: Using this to specify requirements for a Multifunction Watch

- Watch functions include
  - Telling time
  - Alarms
  - Telephone directory
  - Appointment book
  - Memo pad
- With various additional constraints
  - Speed
  - Accuracy
  - Robustness
  - Etc.

Partial Formalization of Functional Decomposition Requirement Specification

A Requirements Specification, R, is a set
R={r | r is a requirement element}

r, a requirement element, is a tuple,
r = (children(r), parent(r), timing(r),
  functionality(r), robustness(r),
  inputs(r), outputs(r) )

ETC.
Intermediate Function Decompositions

Physiology of a Requirement Element

<table>
<thead>
<tr>
<th>NAME</th>
<th>ROBUSTNESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>INPUTS</td>
</tr>
<tr>
<td>PARENTS</td>
<td>ACCURACY</td>
</tr>
<tr>
<td>LOCAL DATA</td>
<td>TIMING</td>
</tr>
<tr>
<td></td>
<td>FUNCTION DEFINITION</td>
</tr>
</tbody>
</table>
Using this for Hierarchy Specification

children(r) = \{ s \in R | s \text{ is a subfunction of } r \}

Question: What do we mean by “subfunction”?  
-- Is included textually within  
-- Requires in order to execute properly  
  (ie. is a module that is used)  
-- ??

parent(r) = The s \in R such that r \in children(s)

ancestors(r) = the transitive closure of r under the parent relation  
= r \cup parent(r) \cup parent(parent(r)) \cup ....

Timing Requirement Specification

timing(r) is a Boolean function

Intuition: To help clarify, define timing(r) to be the function pass_r(testresult), which maps the set of results of executing testcases for r onto the Boolean values \{True, False\}

Thus, pass_r is a function that defines what it means for a testresult to “pass” (ie. to satisfy a timing constraint)

Examples:  
timing(datebook) =  
  pass_datebook(testresult) = testresult < 0.1 seconds  
  pass_datebook(testresult) = testresult < 0.1 + size(datebook_cont)/1000

Note obvious proximity to testing concerns
Functionality Specification

functionality(r) = n, a node in a graph, \( G \)

Intuition: The functionality being specified here is being specified with the help of a graph, whose semantics provide rigor to the definition of the function being required.

The semantics of the graph will turn out to be vitally important in supporting reasoning about the requirements specification and its consistency with design specifications.

Using a DFD

functionality(r) = n, a node in the DFD, \( D = (N, E) \)

Intuition: The functionality being specified here is being specified with the help of a DFD.
Input/Output Specification

inputs(r) = { i | i is an object needed for the execution of the function denoted by n, the DFD node used to define the functionality of r }

outputs(r) = { o | o is an object created by the execution of the function denoted by n, the DFD node used to define the functionality of r }
An FSM can help too

- Use FSM to define how a (set of) function(s) must be related to others.
- Create an “Accessed By” field?
- Value is a pointer into FSM that describes the workings of the system
Each node (state) pointed to by the appropriate field in the appropriate node of the requirements definition

Children of these nodes point to these “substates”
Multirepresentation Systems

- Have seen that different representations are of different uses
- One diagram may be useful in different ways to different stakeholders
- But most stakeholders require a variety of diagrams
- Several different diagrams can be expected to be needed to satisfy the different stakeholders
- Problems with different views/diagrams
  - Are they all representing the same software product?
  - How to assure that they are all consistent with each other?
  - If the product changes, then ALL views must change correspondingly

STATEMATE supports Requirements Specification

- Key feature is maintenance of consistency among views
  --Done by projecting views of abstract model
  --Change abstract model through changes to views
- Use of Hierarchical Decomposition for understandability
- Different diagrammatic views
  --Statecharts (Enhanced FSMs)
  --Enhanced Data Flow Diagrams
- Statecharts, FSMs both support specification of different requirements types simultaneously
  --Functional
  --Robustness
  --Safety
**Initialize**
do: Initialize course object

cancelCourse

**Unassigned**
do: Assign professor to course

cancelCourse

**Open**
entry: Register a student

cancelCourse

**Canceled**
do: Send cancellation notices

**Closed**
do: Report course is full

**RegistrationComplete**
do: Generate class roster

**addStudent**
numStudents = 0

**RegistrationComplete**
do: Generate class roster

**Add student**
numStudents = 0

**registration closed**
numStudents < 3

**registration closed**
numStudents > 3

**numStudents = 10**

**numStudents = 10**

**addStudent**
numStudents = 10

**numStudents = 3**

**numStudents < 3**

**numStudents > 3**

**Statechart with Nested States**

**Initialize**

**Unassigned**
do: Assign professor to course

**Open**
entry: Register a student

**Canceled**
do: Send cancellation notices

**Closed**
do: Report course is full

**RegistrationComplete**
do: Generate class roster

**superstate**

**substate**

**Register**

**Unassigned**
do: Assign professor to course

**Open**
entry: Register a student

**Add student**
numStudents = 0

**registration closed**
numStudents < 3

**registration closed**
numStudents > 3

**numStudents = 10**

**numStudents = 10**

**addStudent**
numStudents = 10

**numStudents = 3**

**numStudents > 3**
Robustness Specification

\[ \text{robustness}(r) \equiv \text{a first order logic implication} \]
\[ A_r \implies B_r \] such that whenever an execution of the functionality defined by \( r \) “satisfies” the predicate \( A_r \), then it necessarily also “satisfies” the predicate \( B_r \).

Example: \( A_r: \) The clock stops ticking
\( B_r: \) The datebook keeps working

Security Specification

- Who is allowed to use this function?
- Under what circumstances?
- With what restrictions?
- Etc.
Security Decomposition

- Focus on who can do what
- Hard to infer that from the functional decomposition
- Makes most sense when security is the key focus of the system
- May make it harder to infer other things
  - Presumably they are less important

Some Possible Relations Involving Such Nodes

- Parents of Children = Children of Parents
- Speed of Children > Speed of Parents
- Functionality related to testcase inputs/outputs
- Same function as one in a DFD
- Data object also used in a DFD
Other Requirements Structures

- Functional decomposition often works well as the structure of a requirements specification
  - Key feature: “all” requirements are clustered by function
- Other decompositions are sometimes better
- Goal decomposition is increasingly interesting
- vanLamsweerde’s KAOS system
  - Goals decomposed into subgoals
  - Special attention to robustness with scoped exception specification

A Very Different Approach: Using Use Cases in Tokeneer

- Tokeneer project to build high-security software
  - To control access to a facility
- Software has little functionality
- But stringent security requirements
- Developed to assure that it works “right”
- Requirements developed using
  - Use Cases
  - Then Z
(Recall) Typical Stakeholders and their Needs

- Customer: needs to
  - communicate what the system must be like
  - have a basis for determining if it works right
- Developer: needs to
  - understand what the system must be like
  - what the design must implement
- Tester: needs to
  - know how to evaluate the final system
- User: needs to
  - see that real underlying needs are going to be met

Evaluate These Requirement Specification Approach Against Requirements Problems

- Incompleteness
- Inconsistency
- Ambiguity
- Infeasibility
- Unevenness
- There are others....
Requirements Incompleteness

- **EXAMPLES:** Missing requirements, missing details

- **CAUSES**
  - Customer may be unavailable, inaccessible, a group
  - Customer asks for too little
    - doesn’t understand computer’s capabilities
    - doesn’t understand interfaces to larger system
  - Customer doesn’t think of *everything*
    - desired function not mentioned
    - special case forgotten
  - The world changes, new things are required

- **REMEDIAL APPROACHES**
  - Cross checking requirements with each other
  - Facilitated by building in relations and redundancy

Requirements Inconsistency

- **Many types of inconsistency: inconsistent wrt what?**
  - Timing: function/subfunction mismatch
  - Functionality: subfunctions don’t “flow right”
  - Robustness: no specification of error recovery

- **CAUSES**
  - Customer may be a group that disagrees
  - Different people may negotiate different parts
  - Early identification of inconsistency can be a big benefit

- **REMEDIAL APPROACHES**
  - “Cross-checking” related requirements elements’
    - What to check against what? How? For what?
Requirements Ambiguity

• More than one possible set of inferences
• CAUSES
  – Customer may be a group where no one sees the whole picture—at least at first
  – Difficult to spot ambiguity in large, complex applications
  – So many parts, related to each other in so many ways
• REMEDIAL APPROACHES
  – Materialize the relations
  – Use them to identify inconsistencies
• PROBLEM: How to control which inferences are possible, and which are not allowable?

Requirements Infeasibility

• CAUSES
  – Customer asks for too much
    » no conceivable algorithm
    » unrealistic requests
  – Still useful to know what ultimate desires are
    » Enables early expectation management
    » Suggests incremental development planning
• REMEDIAL APPROACHES
  – This is a very hard problem
  – Need to determine consistency of requirements specification with designs and implementations
  – Multiple notions of “consistency”
Requirements Unevenness

- **CAUSES**
  - Different sources of information
  - Different people write different parts
  - Different parts of specification are more difficult than others

- **REMEDIAL APPROACHES**
  - Relate details at different levels to other details at similar levels

Approaches to these problems

- Complement requirements with test planning
- Use of requirements processes