Managing Software Requirements in DSD

Stuart Faulk
University of Oregon

What is a “software requirement?”

• A description of something the software must do or property it must have
• The set of system requirements denote the problem to be solved and any constraints on the solution
  – Ideally, requirements specify precisely what the software must do without describing how to do it
  – Any system that meets requirements should be an acceptable implementation

Importance of Getting Requirements Right

1. The majority of software errors are introduced early in software development
2. The later that software errors are detected, the more costly they are to correct

Outline

• Liaison: work with Zach Chen (PKU TA) to help keep us all on track
  – Volunteers?
• First Exercise
  – Observations and issues
  – Completion?
• Lecture: Software Requirements
  – Importance of requirements (review of concepts)
  – Requirements risks in DSD
  – Informal specification
  – Formal specification

© S. Faulk 2010
Requirements Phase Goals

• What does “getting the requirements right” mean in the systems development context?
• Only three goals
  1. Understand precisely what is required of the software
  2. Communicate that understanding to all of the parties involved in the development (stakeholders)
  3. Control production to ensure the final system satisfies the requirements
• Sounds straight-forward but hard to do in practice
• Understanding what makes these goals difficult to accomplish helps us understand how to mitigate the inherent risks

"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 as establishing the detailed technical requirements...No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later."


What makes requirements difficult?

• Comprehension (understanding)
  -- People don’t (really) know what they want (...until they see it)
  -- Superficial grasp is insufficient to build correct software
• Communication
  -- People work best with regular structures, conceptual coherence, and visualization
  -- Software’s conceptual structures are complex, arbitrary, and difficult to visualize
• Control (predictability, manageability)
  -- Difficult to predict which requirements will be hard to meet
  -- Requirements change all the time
  -- Together can make planning unreliable, cost and schedule unpredictable
• Inseparable Concerns
  -- Many requirements issues cannot be cleanly separated (i.e., decisions about one necessarily impact another)
    -- E.g., performance vs. security, safety vs. time to market
    -- Difficult to apply “divide and conquer”
    -- Must make tradeoffs where requirements conflict: requires negotiation among stakeholders with competing interests

Additional Risks of DSD

• DSD tends to aggravate existing difficulties
• Comprehension
  -- Different languages, cultures, expectations
  -- Greater risk of ambiguity, misunderstanding
• Communication
  -- Reduced communication: limited bandwidth, language, culture, invisible stakeholders
  -- More difficult to negotiate common understanding, may miss problems
• Control
  -- Less feedback, often delayed
  -- Easy to get out of synch and not know it
  -- Easy to lose track of the effects of changes
• Inseparable concerns
  -- Difficulty making clean divisions, allocation of responsibility among sites
  -- Conversely, easy to have inadvertent side effects on other code
Sources of Errors in DSD

In Planning: View as Risks

- Risk: requirements are misunderstood
  - Problem: There is a misunderstanding of exactly what the stakeholders require. Or, the distributed teams have a different understanding of one or more requirements.
  - Mitigations: ?
- Risk: missing requirements
  - Problem: One or more requirements are not identified (requirements are incomplete). Or, the distributed teams have somewhat different sets of requirements.
  - Mitigation: ?
- Risk: requirements change
  - Problem: During development, one or more requirements change (equivalently, our understanding changes).
  - Mitigation: ?

Risk Mitigation Strategies

- Build risk mitigation into the project’s software process
  - Requirements changes occur throughout development
  - Must address at all stages of process
- Requirements are missing, misunderstood
  - Requirements exploration with stakeholders (customer)
  - Early modeling: prototypes, mockups
  - Precise documentation
  - Careful review
  - Incremental delivery
  - Clear responsibilities for requirements tasks, products
- Requirements change
  - Consider the effects of changes in advance
  - Software design for robustness, ease of change
  - Explicit processes for managing change

Requirements Process Components

- Process components – integrate into project plan
- Activities: deploy explicit requirements activities
  - Requirements exploration and understanding
  - Requirements negotiation (and explicit signoff)
  - Requirements specification
  - Requirements verification and validation (feedback)
  - Change management for distributed team
- Artifacts: provide vehicles for capture, communication & assessment, e.g.,
  - Prototypes, mock-up, story board, use cases
  - Common requirements specification
  - Reviews
- Roles: create clear responsibilities for activities, artifacts, and communication, e.g.,
  - Analyst: exploration, customer interaction, negotiation
  - Requirements Engineer: specification
  - Reviewer: verification and validation
Process View

<table>
<thead>
<tr>
<th>Process</th>
<th>Activities</th>
<th>Artifacts</th>
<th>Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Elicitation &amp; negotiation</td>
<td>Prototypes, use cases, etc</td>
<td>Customer Analyst</td>
</tr>
<tr>
<td></td>
<td>Specification</td>
<td>ConOps</td>
<td>Reviewer</td>
</tr>
<tr>
<td></td>
<td>Verification and validation</td>
<td>Formal SRS</td>
<td></td>
</tr>
<tr>
<td>Architecture</td>
<td>Design for distribution</td>
<td>Module structure</td>
<td>Architect</td>
</tr>
<tr>
<td></td>
<td>Design for change</td>
<td>Module interface specs.</td>
<td>Reviewer</td>
</tr>
<tr>
<td></td>
<td>Verification</td>
<td>Active review</td>
<td></td>
</tr>
<tr>
<td>Detailed Design/code</td>
<td>Verification and validation</td>
<td>Test cases</td>
<td>Coder</td>
</tr>
<tr>
<td>System Integration/test</td>
<td>Validation</td>
<td>Test cases</td>
<td>Tester</td>
</tr>
<tr>
<td>Change Management</td>
<td>Change Request</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Requirements Elicitation & Negotiation

- Focuses on the question: “Are we building the right system?”
- Process activities
  - Requirements analysis: interact with customer and other stakeholders
    - Identify stakeholders
    - Identify both functional and “non-functional” requirements (performance, maintainability, etc.)
    - Identify likely changes (evolution)
  - Negotiate delivered capabilities with stakeholders
    - Find consensus among distributed teams
    - Explicit agreement on requirements (signoff)
- Products
  - Anything useful to ensuring a correct understanding of the problem domain and stakeholder desires
  - Examples: mock ups, use cases, demonstration prototypes

Requirements Communication

- Focus on the question: “Do all of the stakeholders share a common understanding of what must be built?”
  - Particularly critical for DSD
- Process activities
  - Requirements specification: capture complete requirements in sufficient detail to serve the needs of all the stakeholders
  - Requirements verification: check that the requirements spec is of adequate quality (complete, consistent, testable, etc.)
  - Requirements communication: check for consistent understanding
    - Usually implicit in co-located projects
    - Should be incorporated as part of verification (e.g., distributed reviews, validation)
- Products
  - Requirements documentation
  - Reviews

Needs of Different Audiences

- Customer/User
  - Focus on problem understanding
  - Use language of problem domain
  - Technical if problem space is technical
- Development organization
  - Focus on system/software solutions
  - Use language of solution space (software)
  - Precise and detailed enough to write code, test cases, etc.
Two Kinds of Software Requirements

- Communicate with customers: i.e., stakeholders who understand the problem domain but not necessarily programming (solution domain)
  - Do not understand computer languages but may understand technical domain-specific languages
  - Must develop understanding in common languages
- Communicate with developers: sufficiently precise and detailed to code-to, test-to, etc.
  - Stated in the developer’s terminology
  - Addresses properties like completeness, consistency, precision, lack of ambiguity
- For businesses, these may be two separate documents

Documentation Approaches

- ConOps: Informal requirements to describe the system’s capabilities from the customer/user point of view
  - Purpose is to answer the questions, “What is the system for?” and “How will the user use it?”
  - Tells a story: “What does this system do for me?”
  - Helps to use a standard template
- SRS: More formal, technical requirements for development team (architect, coders, testers, etc.)
  - Purpose is to answer specific technical questions about the requirements quickly and precisely
    - Answers, “What should the system output in this circumstance?”
    - Reference, not a narrative, does not “tell a story”
  - Goal is to develop requirements that are precise, unambiguous, complete, and consistent

Informal Techniques

- Most requirements specification methods are informal
  - Natural language specification
  - Use cases
  - Mock-ups (pictures)
  - Story boards
- Benefits
  - Requires little technical expertise to read/write
  - Useful for communicating with a broad audience
  - Useful for capturing intent (e.g., how does the planned system address customer needs, business goals?)
- Drawbacks
  - Inherently ambiguous, imprecise
  - Cannot effectively establish completeness, consistency
  - However, can add rigor with standards, templates, etc.

Mock-up Example
1. Use Case: Manage Reports

1.1 Description

This Use Case describes operation for Creating, Saving, Deleting, Printing, Exiting and Displaying reports.

1.2 Actors

User: Project database

1.3 Triggers

Program Manager selects operations from menu.

1.4 Flow of events

1.4.1 Basic Flow

1. User chooses desired report by selecting “Report” -> “Open” from the menu bar
2. System displays report to screen
3. User selects desired report layout using Use Case Specify Report
4. Steps 2 and 3 are repeated until user is satisfied
5. User can Save or Print report using use case Save Report or Print Report
6. User Exits report by selecting “Exit” from the “File” menu

1.4.2 Alternative Flows

1.4.2.1 Create New Report

1. User selects “Create New Report” from file menu
2. ...

1.4.2.2 Delete Report

1. ...

1.4.3 Preconditions

A systematic approach to use cases

- Uses a standard template
- Easier to check, read
- Still informal

Technical Specification

The SRS

The role of rigorous specification

Requirements Documentation

- Is a detailed requirements specification necessary?
- How do we know what “correct” means?
  - How do we decide exactly what capabilities the modules should provide?
  - How do we know which test cases to write and how to interpret the results?
  - How do we know when we are done implementing?
  - How do we know if we’ve built what the customer asked for (may be distinct from “want” or “need”)?
  - Etc...
- Correctness is a relation between a spec and an implementation
- Implication: until you have a spec, you have no standard for “correctness”

Audience and Purpose

- Provides the detailed technical specification of the system’s behavioral and quality attributes
- Audiences and purposes
  - Developers: provides the definitive specification of system behavior. Any implementation that satisfies the technical requirements should be an acceptable system
  - Managers: defines precisely what needs to be done, guidance for planning
  - Testers: specifies the range of acceptable outputs for any given inputs.
SRS Contents

- **Audience and purpose**
  - Also describes how to use the document to find specific kinds of information

- **Requirements**
  - System context
    - Precise specification of system inputs and outputs
    - Note that the format (interface design) is a distinct issue
  - Behavioral Requirements: requirements and constraints on the observable behavior of the running system
    - Functional - acceptable outputs for any possible input
    - Accuracy constraints
    - Performance constraints
    - Safety, privacy, usability, etc.
  - Developmental quality requirements: any constraints on the static construction of the system or the development process
    - Modifiability (ease of change)
    - Reusability
    - Structuring constraints (e.g. for distributed development)

SRS Contents (2)

- **Use Case Models**
  - Technically an informal spec.
  - Here to support ease of communication

- **Fundamental assumptions**
  - Assumptions about the software or environment that you assume will not change over the life of the system.
  - Example: that you will always use a web-service for face recognition or an android mobile platform might be fundamental assumption

- **Anticipated changes**
  - Requirements or constraints that are likely to change over the life of the system
  - Result of thinking in advance about how the system will evolve
  - How can this be used?

- **Appendices**
  - Definitions and acronyms
  - Requirements sources
  - References
  - Template provided in Assembla wikis

Technical Requirements

- **Focus on developing a technical specification**
  - Should be straightforward to determine acceptable inputs and outputs
  - Preferably, can systematically check completeness consistency

- **A little rigor in the right places can help a lot**
  - Adding formality is not an all-or-none decision
  - Use it where it matters most to start (critical parts, potentially ambiguous parts)
  - Often easier, less time consuming than trying to say the same thing in prose

- **E.g. in describing conditions or cases**
  - Use predicates (i.e., basic Boolean expressions)
  - Use mathematical expressions
  - Use tables where possible

Control Production

- **Control production to ensure the final system satisfies the requirements: this means:**

- **Team members use the requirements specification as the basis of development**
  - Designing the architecture: requirements define all external design goals, e.g.: expected changes, subsets, extensions, etc.
  - Designing modules to provide required capabilities

- **Team members use the requirements spec as a basis for verification**
  - Verify designs against requirements
  - Basis of system test planning (as opposed to module)

- **If the spec is not adequate for these purposes, then fix the spec!**
Summary

• Requirements characterize “correct” system behavior
• Being in control of development requires:
  – Getting the right requirements
  – Communicating them to the stakeholders
  – Using them to guide development
• Requirements activities must be incorporated in the project plan
  – Requirements baseline
  – Requirements change management

Assignments

• Complete first meeting, team-page exercise
• First iteration of project plan
  – Choose team roles (both sides)
  – Plan first increment
    • What are the tasks?
    • Who will do them?
    • When will they be complete?

Questions?