CS48

Chandra Krintz:  ckrintz@ucsb.edu
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~ckrintz/

http://www.cs.ucsb.edu/~cs48
Vision Statement and Draft Project: Part of your **Software Development Process**

1. **Software specification**
   - Customers and engineers define the software that is to be produced and any constraints on its operation; vision statement; requirements analysis

2. **Software design**
   - Software spec is designed and prototyped

3. **Software implementation, validation, and testing**
   - Software is programmed and checked to ensure that it is what the customer requires

4. **Software maintenance and evolution**
   - Software is maintained (bug fixes, upgrades) and modified to reflect changing customer and market requirements
Discussion/debate on the functionality, input and output formats, types of users, etc. is called requirements analysis.

Writing precise requirements specifications can be challenging:
- English is easy to read and write, but ambiguous
- Today’s solutions employ a combination of
    - Ours is called a Draft Project
  - Should be a “living document” that evolves over time
    - Starts with a vision statement
Draft Project = Agile Requirements Specification

1. Define project specifics
2. Team goals and objectives
3. Background and strategic fit
4. Assumptions
5. User Stories or Use Cases
6. User Interaction and Design
7. Questions
8. What we’re NOT Doing

• Evolve the document over time, concurrently with development

Required reading: https://www.atlassian.com/agile/requirements
Draft Project

• Authors, Team, Project Title

• Intro – **Vision Statement** (revised): 2-3 pages
  – Define project specifics, team goals/objectives, background, and assumptions
  – Links to github, travis-ci, slack

• System architecture overview (drawings)
  – High level diagram (1 page)
  – Interfaces and interactions between components

• Requirements (functional and non-functional)
  – **User stories or use cases** → 5+ for first turn-in, 10+ for final turn-in
    -- each prioritized with *acceptance tests*
  • Links to github issues, commits, tests/demos for all subtasks
  • You can use Trello or Pivotal for free (online stories/use cases) – if you use this, then make sure TAs/instructor are invited and give **links** in draft project

• Appendix
  – List of technologies employed
Functional and Non-functional Requirements

- Functional requirements (user + system requirements)
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May also state what the system should not do.

- Domain requirements
  - Constraints on the system from the domain of operation
    - Operating environment (e.g. underwater, temp range, environmental conditions to be tolerated)

- Non-functional requirements
  - Constraints on services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.
Types of Non-functional Requirements

Non-functional requirements

Product requirements
- Efficiency requirements
- Usability requirements
- Performance requirements

Organizational requirements
- Dependability requirements
- Security requirements
- Environmental requirements
- Operational requirements

External requirements
- Regulatory requirements
- Development requirements
- Legislative requirements
- Accounting requirements
- Safety/security requirements
## Metrics for Specifying Non-functional Requirements

<table>
<thead>
<tr>
<th>Property</th>
<th>Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>Processed transactions/second</td>
</tr>
<tr>
<td></td>
<td>User/event response time</td>
</tr>
<tr>
<td></td>
<td>Screen refresh time</td>
</tr>
<tr>
<td>Size</td>
<td>Mbytes</td>
</tr>
<tr>
<td></td>
<td>Number of ROM chips</td>
</tr>
<tr>
<td>Ease of use</td>
<td>Training time</td>
</tr>
<tr>
<td></td>
<td>Number of help frames</td>
</tr>
<tr>
<td>Reliability</td>
<td>Mean time to failure</td>
</tr>
<tr>
<td></td>
<td>Probability of unavailability</td>
</tr>
<tr>
<td></td>
<td>Rate of failure occurrence</td>
</tr>
<tr>
<td></td>
<td>Availability</td>
</tr>
<tr>
<td>Robustness</td>
<td>Time to restart after failure</td>
</tr>
<tr>
<td></td>
<td>Percentage of events causing failure</td>
</tr>
<tr>
<td></td>
<td>Probability of data corruption on failure</td>
</tr>
<tr>
<td>Portability</td>
<td>Percentage of target dependent statements</td>
</tr>
<tr>
<td></td>
<td>Number of target systems</td>
</tr>
</tbody>
</table>
Draft Project: your agile requirements document

- The official statement of what is required of the system developers
- Includes a specification of both user and system requirements
  - Functional and non-function requirements
- Defines **WHAT** the system should do not **HOW** it should do it
  - Design comes later

- Agile and extreme SWE processes express requirements as
  - **Use cases** – how a system will act
  - Or as scenarios called **user stories** (describe result/benefit of it)
Use Cases

• Use cases document the behavior of the system from users’ point of view.
  – By user we mean anything external to the system
  – Consist of:
    actors – scope – goals – steps – success

• An actor is a role played by an outside entity that interacts directly with the system
  – An actor can be a human, or a machine or program
  – Actors are shown as stick figures in use case diagrams

Customer
Use Cases

• A use case describes the possible sequences of interactions among the system and one or more actors in response to some initial stimulus by one of the actors
  – Each way of using the system is called a use case
    • Sequence of interactions
  – A use case is not a single scenario but rather a description of a set of scenarios
    – For example: Creating an account or Performing transaction or Applying for a loan

• In a use case, the system is considered a black-box.

We are only interested in describing externally visible behavior
Use Cases

• To define a use case, group all transactions that are similar

• A typical use case might include a main case, with alternatives taken in various combinations and including all possible exceptions that can arise in handling them

  – Use case for an online banking app: Performing a Transaction
    • Subcases could include Making Deposits, Making Withdrawals, etc., together with exceptions such as Overdrawn or Account Closed
  – Apply for a Loan could be a separate use case since it is likely to involve very different interactions
Generalization in Use Case Diagrams

Customer

Individual Customer
Corporate Customer

Validate User

Check Password
Retinal Scan

Indicates generalization
Use Cases

• Description of a use case should include **events exchanged between objects and the operations performed by the system that are visible to actors**

• **Have preconditions and postconditions**
  - Precondition states all assumptions about state/environment of system that impacts the actor(s) in this use case
  - Postcondition is an acceptance test (how to know when implementation is complete) and describes externally visible state/environmental changes
Use case: Update Benefits


Precondition: Employee has logged on to the system and selected “update benefits” option

Flow of Events:

Basic Path:
1. System retrieves employee account from Employee Account Database
2. System asks employee to select medical plan type; uses Update Medical Plan
3. System asks employee to select dental plan type; uses Update Dental Plan
   ...

Alternative Paths:
- If health plan is not available in the Employee’s area the employee is informed and asked to select another plan (exceptional cases that must be handled)
  Employee selects cancel, logs out, or leaves page at any point prior to confirming the update (an end-early path)

Postcondition: Employee account plan type has been updated in the Employee Account Database or nothing has changed (end-early paths)

Note that code tests can be written for pre/post conditions
User Stories
User Stories

• Similar to Use Cases but not the same
  – User stories are centered on the result and the benefit of the thing you’re describing, whereas use cases are more granular, and describe how your system will act. From: http://www.boost.co.nz/blog/2012/01/use-cases-or-user-stories/

• Use cases: actors – scope – goals – steps – success
  – Details of most important requirements worked out ahead of time to ensure that everyone is on the same page
  – Useful for groups of similar stories and describing overall system
    • Use cases decompose stories into actions in the system

• User stories: scope of a feature + acceptance criteria
  – Each feature is captured as a story; stories easily prioritized
  – A story is a place holder for discussion and planning poker in a sprint

See recommended reading links for examples and suggestions
User Stories

• Stems from Behavior Driven Development (BDD)
  – Employed in XP/Agile processes
  – Improves communication/understanding of requirements by all involved

• An outside-in methodology
  – Encourage discovery: drill down on a feature set to achieve desired (business) outcomes

• See for examples
  – Dan North: “What’s in a Story?”
  – Agile Modeling: “Introduction to User Stories”
Writing Good User Stories

• It's typically difficult to get started writing good user stories
  – Here are 4 steps to make it easier

1. As a [role], I can [feature] so that [reason]
2. Use index cards and a sharpie
3. Make it testable with acceptance criteria or demo plan
4. Connect the dots

From: http://codesqueeze.com/the-easy-way-to-writing-good-user-stories/
As a [role], I can [feature] so that [reason]

- Role – a person; feature – something your project does; reason – a solution to a problem the person has
  - This is a pattern that is commonly used for stories

  As a account owner, I can check my balance online so that I can access my daily balance 24 hours a day.

- Variations
  - As a [role], I want [feature] because [reason]
  - As a [role], I can [feature]
  - As a [role], I can [feature] so that [reason]
Use index cards and a sharpie

• Although there is software out there to help you with this
  – Jira, Trello, Pivotal tracker

• Physically writing out stories facilitates keeping the story clear, concise, and of the appropriate size
  – Keep them short and sweet and unambiguous
    • Goal is to aid communication, not overly detailed or long-winded
  – It also enables you to doodle/draw the outline of the user interface

• If it doesn’t fit, break up the story into sub-stories
Make it testable with acceptance test or demo

- If they are short and sweet and without detail, how do we know when they are “done”?

**Story**: As a [role], I can [feature] so that [reason]

- Include an acceptance test (what to demo when done):

  **Scenario 1: Title**
  
  **Example**

  Given the account’s balance is below 0
  And there is not a scheduled direct deposit that day
  When the account owner attempts to withdraw money
  Then the bank will deny it
  And send the account owner a nasty letter.

- All tests should fit on back of story card (in sharpie)
  - If they don’t, break up the story into two
  - You should be able to **code** them in a few lines of code
Writing Good User Stories

• It's typically difficult to get started writing good user stories
  – Here are 4 steps to make it easier

1. As a [role], I can [feature] so that [reason]
2. Use index cards and a sharpie
3. Make it testable with acceptance criteria or demo plan
4. Connect the dots
  – Lay the stories out, determine which ones are dependent on others, prioritize them in order to provide a working system/product each sprint

From: http://codesqueeze.com/the-easy-way-to-writing-good-user-stories/