

CS1632, Lecture 3: Requirements

Wonsun Ahn

What are requirements?

- The specifications of the software
 - Often collected into a SRS, Software Requirements Specification
- That is, the finished software is required to meet the requirements
- This is how developers know what code to write, and (more importantly for this class), testers know what to test

Requirements Example – Bird Cage

- The cage shall be 120 cm tall.
- The cage shall be 200 cm wide.
- The cage shall be made of stainless steel.
- The cage shall have one dish for food, and one dish for water, of an appropriate size for a small bird.
- The cage shall have two perches.
- At least 90% of birds shall like the cage.

Problems With Our Requirements?

- What if the cage is 120.001 cm tall.. OK?
- What if the cage is 120 km tall.. OK?
- Is 120 cm tall and 200 cm wide an appropriate size for a small bird?
- How can we know birds like it?
- Do we have to ask all the birds in the world to see if 90% like it?
- Food dishes are plastic... OK?
- The perches are wood... OK?
- 2 cm gaps between cage wires... OK?
- 60 cm gaps between cage wires.. OK?
- How many birds should it support?
- Cage has no door... OK?
- Cage has 17 doors, which are opened via elaborate puzzles... OK?

Most software is more
complex than a bird cage!



What the customer said



What was understood



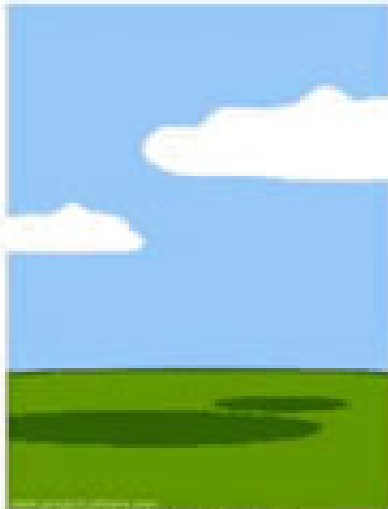
What was planned



What was developed



What was described by the business analyst



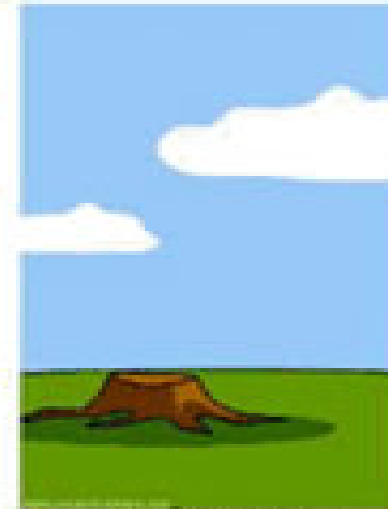
What was documented



What was deployed



The customer paid for...



How was the support



What the customer really needed

You need to understand requirements!

- Why? Because they describe the expected behavior!
 - Expected behavior vs observed behavior is the foundation of testing software
- A Software Requirements Specification (SRS) is in a sense a contract
 - A contract between user, developer, and tester
- Requirements are not (usually) set in stone and do evolve
 - But a clear understanding of them is crucial during the entire process
 - As a QA engineer, you will participate in its evolution

Requirements should say WHAT to do, not HOW to do it!

- **GOOD:** The system shall store all logins for future review.
- **BAD:** The system shall use an associative array in a singleton class called AllLoginsForReview to store all logins.

- **GOOD:** The system shall support 100 concurrent users.
- **BAD:** The system shall use a BlockingQueue in order to support 100 concurrent users.

Requirements should say WHAT to do, not HOW to do it! Why?

- User cares about what the behavior is, not how it happens
 - Generally true for (almost) all software
- Specifying how restricts developers from improving implementation
- Black-box testing is impossible if implementation details are required
 - End-user would need source code to verify requirements have passed

Verification vs Validation

- Verification – Are we building the ***software right?***
 - Ensure that requirements are met by software (testing)
 - Ensure that there are no unexpected failures, observed behavior meets the requirements, corner cases are handled, etc.
- Validation – Are we building the ***right software?***
 - Ensure that software does what customer/user actually wants
 - Ensure that requirements don't contain any inconsistencies
 - Ensure that requirements are verifiable

Requirements Validation

- Should happen before writing the first line of code
- Should happen throughout development as requirements evolve
- Aspects of requirements validation:
 - Validity check – does SRS align with user needs?
 - Completeness check – does SRS cover all aspects of software?
 - Consistency check – is SRS internally consistent? Any conflicts?
 - Realism check – is SRS realistic in terms of technology / budget / delivery date?
 - Ambiguity check – does SRS contain room for interpretation?
 - Verifiability check – are the requirements something that can be tested?

Validity Check

- Requirements should align with stakeholders needs and wants
- Common misconceptions
 - Misconception: “More is better” – more functionality, more modes, etc.
Truth: “Less is more” – ease-of-use and elegance suffers with “more”
 - Misconception: Stakeholders are limited to users
Truth: Stakeholders include users, operators, managers, investors
 - Misconception: Stakeholders know what they need and want
Truth: What looks good on paper often is a flop when seen in real life
 - ☛ Motivation to do early prototyping and demos in front of stakeholders

Completeness Check

- Requirements should cover all aspects of a system
 - Anything not covered is liable to different interpretation
 - If you care that something should occur a certain way, it should be specified

Consistency Check

- Requirements must be internally consistent
 - Requirements must not contradict each other.
- Example
 - Req 1: "The system shall shut down if the temperature reaches **-20 C**"
 - **BAD** Req 2: "The system shall turn on LOWTEMP warning light whenever temperature is **-40 C** or colder."
 - **GOOD** Req 2: "The system shall turn on LOWTEMP warning light whenever temperature is **0 C** or colder."

Realism Check

- Requirements must be realistic in technology / budget / delivery date
- Example
 - **BAD:** The system shall communicate between Earth and Mars with a round-trip latency of less than 25 ms (That's faster than the **speed of light!**).
 - **GOOD:** The system shall communicate between Earth and Mars with a round-trip latency of less than 42 minutes at apogee and 24 minutes at perigee.
- Is it unrealistic to communicate faster than the speed of light?
 - Quantum entanglement signaling is known to travel faster than speed of light
 - Then, question becomes can you deliver system within budget by delivery date

Ambiguity Check

- Requirements should not be open to interpretation
- Example
 - **BAD:** When database stores an invalid Date, it shall be set to default value.
 - **GOOD:** When database stores an invalid Date, it shall be set to 1 Jan 1970.
(1 Jan 1970 happens to be time “0” in all Unix and Linux systems)
- Example
 - **BAD:** The system shall on error shut down gracefully in a timely manner.
 - **GOOD:** The system shall on error store all pending requests in a checkpoint file configurable on the command line and then shut down within 5 seconds.

Verifiability Check

- Requirements must be testable
 - Should be documented to the level where it is testable
 - Should be testable within the given budget and delivery date
- Example
 - **BAD:** The calculator subsystem shall be awesome.
 - **GOOD:** The calculator subsystem shall include functionality to add, subtract, multiply, and divide any two integers between MININT and MAXINT.
- Example
 - **BAD:** The system shall process a 100 TB data set within 4,137 years.
 - **GOOD:** The system shall process a 1 MB data set within 4 hours.

FUNCTIONAL REQUIREMENTS AND QUALITY ATTRIBUTES (NON-FUNCTIONAL REQUIREMENTS)

- **Functional Requirements**
 - Specify functional behavior of system
 - The system shall do X [under conditions Y].
- **Quality Attributes**
 - Specify overall qualities of system, not a specific behavior
 - The system shall be X [under conditions Y].
- Note “do” vs “be” distinction!

FUNCTIONAL REQUIREMENT EXAMPLES

- **Req 1:** The system shall **return** the string "NONE" if no elements match the query.
- **Req 2:** The system shall **turn on** the HIPRESSURE light when internal pressure reaches 100 PSI.
- **Req 3:** The system shall **turn off** the HIPRESSURE light when internal pressure drops below 100 PSI for more than five seconds.

QUALITY ATTRIBUTE EXAMPLES

- **Req 1** - The system shall **be** protected against unauthorized access.
- **Req 2** - The system shall **have** 99.999 uptime and **be** available.
- **Req 3** - The system shall **be** easily extensible and maintainable.
- **Req 4** - The system shall **be** portable to other processor architectures.

SOME CATEGORIES OF QUALITY ATTRIBUTES

- Reliability
- Usability
- Accessibility
- Performance
- Safety
- Supportability
- Security

You can see why quality attributes are sometimes called “-ility” requirements!

Quality attributes are often more difficult to test than functional requirements.

- Why?
- Can be very subjective
- Often difficult to measure
- No standardized rules for considering them "met"

Solution

*Agree with stakeholders upon **quantifiable requirements** that ensure quality.*

Converting Qualitative to Quantitative

- **Performance:** transactions per second, response time
- **Reliability:** Mean time between failures
- **Robustness:** How many failures can the system cope with
- **Portability:** Number of systems targeted, or how long it takes to port
- **Safety:** Number of accidents per year
- **Usability:** Average amount of time required for training
- **Accessibility:** Percentage of population who can use system

Qualitative to Quantitative Example

- Quality attributes should be expressed in a quantitative way
 - Or else they are ambiguous
- Example
 - **BAD:** The system shall be highly usable.
 - **GOOD:** Over 90% of users shall have no questions using the software after one hour of training.
- Example
 - **BAD:** The system shall be reliable enough to be used in a space station.
 - **GOOD:** The system shall have a mean-time-between failures of 100 years.

Now Please Read Textbook Chapters 5

- If you are interested in further reading:

**IEEE Recommended Practice for
Software Requirements Specifications (IEEE Std 830-1998)**

- Can be found in resources/IEEE830.pdf in course repository