

# 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



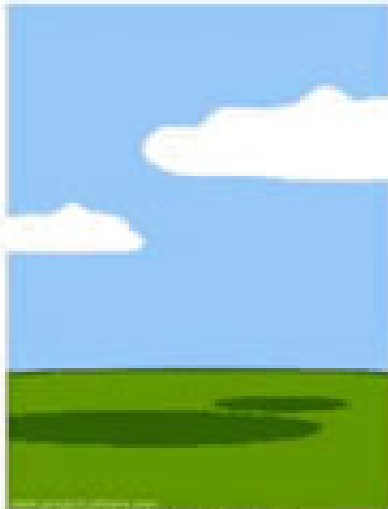
What was planned



What was developed



What was described by the business analyst



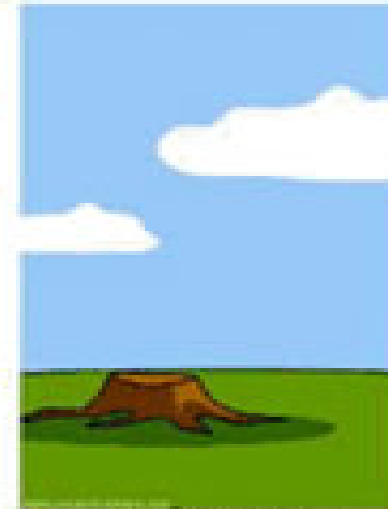
What was documented



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 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

# Verification vs Validation

- Verification – Did we build the ***software right?***
  - Ensure that requirements are met
  - Ensure that there are no unexpected failures, output is correct, edge cases handled, etc. (implicit requirements)
- Validation – Did we build the ***right software?***
  - Ensure that software does what customer/user actually wants



# 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.

From a requirements perspective, we care about WHAT the system does, not HOW

- We want to know – does the system do X in situation Y, under circumstances Z?
- Black-box testing can be impossible if we need to know implementation details
- Specifying implementation details restricts designers and developers from implementing better solutions

# TESTABILITY

- Requirements should be testable. What this means, exactly, will vary, but we have some guidelines.
- GOOD: The calculator subsystem shall include functionality to add, subtract, multiply, and divide any two integers between MININT and MAXINT.
- BAD: The calculator subsystem must be awesome. Like, seriously awesome.

*Requirements should be...*

- Complete
- Consistent
- Unambiguous
- Quantitative
- Feasible to test

# COMPLETE

- Requirements should cover all aspects of a system. Anything not covered in requirements is liable to be interpreted differently!
- If you care that something should occur a certain way, it should be specified in the requirements

# CONSISTENT

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

# Internally and Externally Consistent

- **BAD:** The system shall communicate between Earth and Mars with a round-trip latency of less than 25 ms.
- **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.

# UNAMBIGUOUS

- **BAD:** When the database system stores a String and an invalid Date, it should be set to the default value.
- **GOOD:** When the database system stores a String and an invalid Date, the Date should be set to the default value (1 Jan 1970).



# QUANTITATIVE

- BAD: The system shall be responsive to the user.
- GOOD: When running locally, user shall receive results in less than 1 second for 99% of expected queries.

# FEASIBLE TO TEST

- **BAD:** The system shall complete processing of a 100 TB data set within 4,137 years.
- **GOOD:** The system shall complete processing of 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 (five 9's) uptime and **be** available during that same time.
- **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 would take 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

- **BAD:** The system must be highly usable.
- **GOOD:** Over 90% of users have no questions using the software after one hour of training.

Now Please Read Textbook Chapters 5