# CS1632, Lecture 3: Requirements

Wonsun Ahn

## What are requirements?

- The specifications of the software
    - Often collected into an SRS, Software Requirements Specification
    - The SRS comes in legal binders hundreds of pages long
    - And, yes, the SRS is legally binding (pun intended)

- This is how developers know what code to write

- This is also how testers know what to test

## Requirements Evolve

- Requirements are not (usually) set in stone and do evolve
  - Means your code implementation must evolve with the requirements
  - Means your testing infrastructure must also evolve with the requirements
  - A clear understanding of the requirements is crucial at any given point
- *Requirements engineering*: Managing and documenting requirements
  - Also part of QA since low quality requirements result in low quality software
  - Bad requirements engineering can cause *requirements creep*

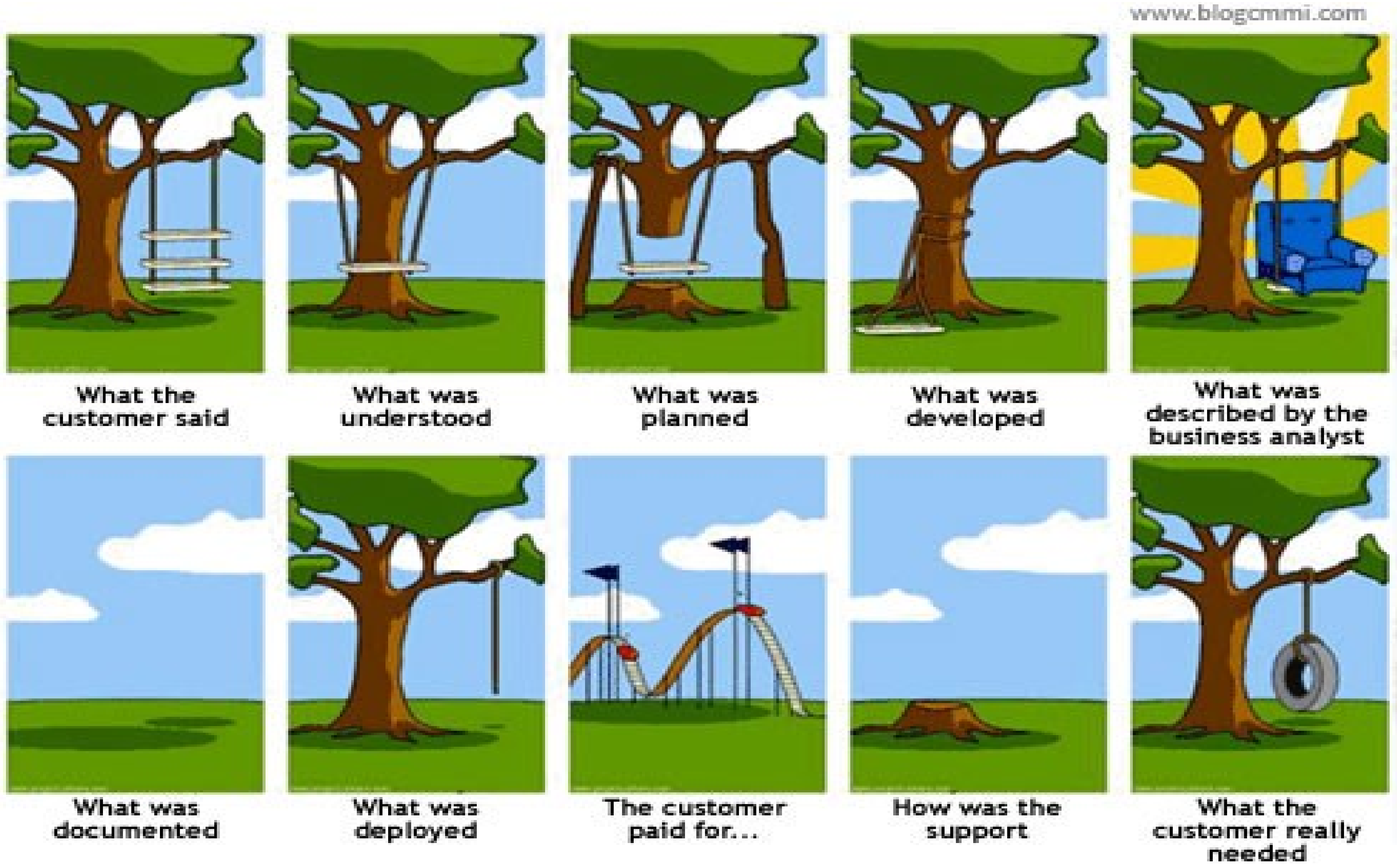
## Requirements Example – Bird Cage

- 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!


www.blogcmmi.com
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

## Role of Software QA Engineer

- *Requirements Verification*: Are we building the **software right**?
  (a.k.a. **testing**)
  1. Derive expected behavior from requirements for each test case
  2. Compare expected behavior with observed behavior

- *Requirements Validation*: Are we building the **right software**?
  1. Pore over requirements to make sure they make sense
  2. Interview stakeholders to see if requirements match actual needs
  3. Interview developers to see if requirements are technically feasible
  4. Interview testers to see if requirements are verifiable

## Requirements Validation

- Aspects of requirements validation:
  - Does the SRS make sense internally?

    *Completeness check*: does SRS cover all aspects of software?

    *Consistency check*: does SRS contain any logical conflicts?

    *Ambiguity check*: does SRS contain room for interpretation?
  - Does the SRS make sense externally?

    *Validity check*: does SRS align with user needs?

    *Realism check*: is SRS something that can be feasibly implemented?

    *Verifiability check*: is SRS something that can be feasibly tested?
- Happens throughout product cycle as requirements evolve

## 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."

## 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.

## Validity Check

- Requirements must 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

## Realism Check

- It must be realistic to **implement** the requirements
  - Must be realistic in terms of current technology
  - Must be realistic within the given budget and delivery date
- Example (unrealistic in terms of technology)
  - **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.

## Verifiability Check

- It must be feasible to **test** the requirements
  - Should be documented to the level where it is testable
  - Should be testable within the given budget and delivery date
- Example (not documented in enough detail)
  - **BAD**: The calculator subsystem shall perform all arithmetic.
  - **GOOD**: The calculator shall include functionality to add, subtract, multiply, and divide any two integers between MININT and MAXINT.
- Example (not testable within delivery date)
  - **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.

## 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 to store users, with a maximal size of 100.

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

- Users care about what the software does, not how it happens (usually)
- Specifying how restricts developers from improving implementation
- Specifying how precludes blackbox testing
  - Means customer cannot verify requirements just by trying out the software
  - Means developer must provide source code to customer in order to verify

## Functional and Non-Functional Requirements

- **Functional Requirements**
  - Specify functional behavior of system
  - The system shall **do** X on input Y.

- **Non-Functional Requirements (Quality Attributes)**
  - Specify overall qualities of system, not a specific behavior
  - The system shall **be** X during operation.

- Note “do” vs “be” distinction!

# Functional Requirement Examples

- **Req 1:** System shall return "NONE" if no elements match the query
- **Req 2:** System shall turn on HIPRESSURE light at 100 PSI
- **Req 3:** System shall throw an exception on illegal parameters

# 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.

## Quality Attribute Categories

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

*You can see why quality attributes are sometimes called "-ility" requirements!*

## Quality Attributes are Difficult to Test

- Why? Because they are qualities.

- Often difficult to measure
- Can be very subjective

## 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 simultaneous failures can system cope with
- **Portability:** Number of systems targeted, or how long it takes to port
- **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