REQUIREMENTS, TEST PLANS, AND TRACEABILITY MATRICES

Bill Laboon

Twitter: @BillLaboon

Email: laboon@cs.pitt.edu

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

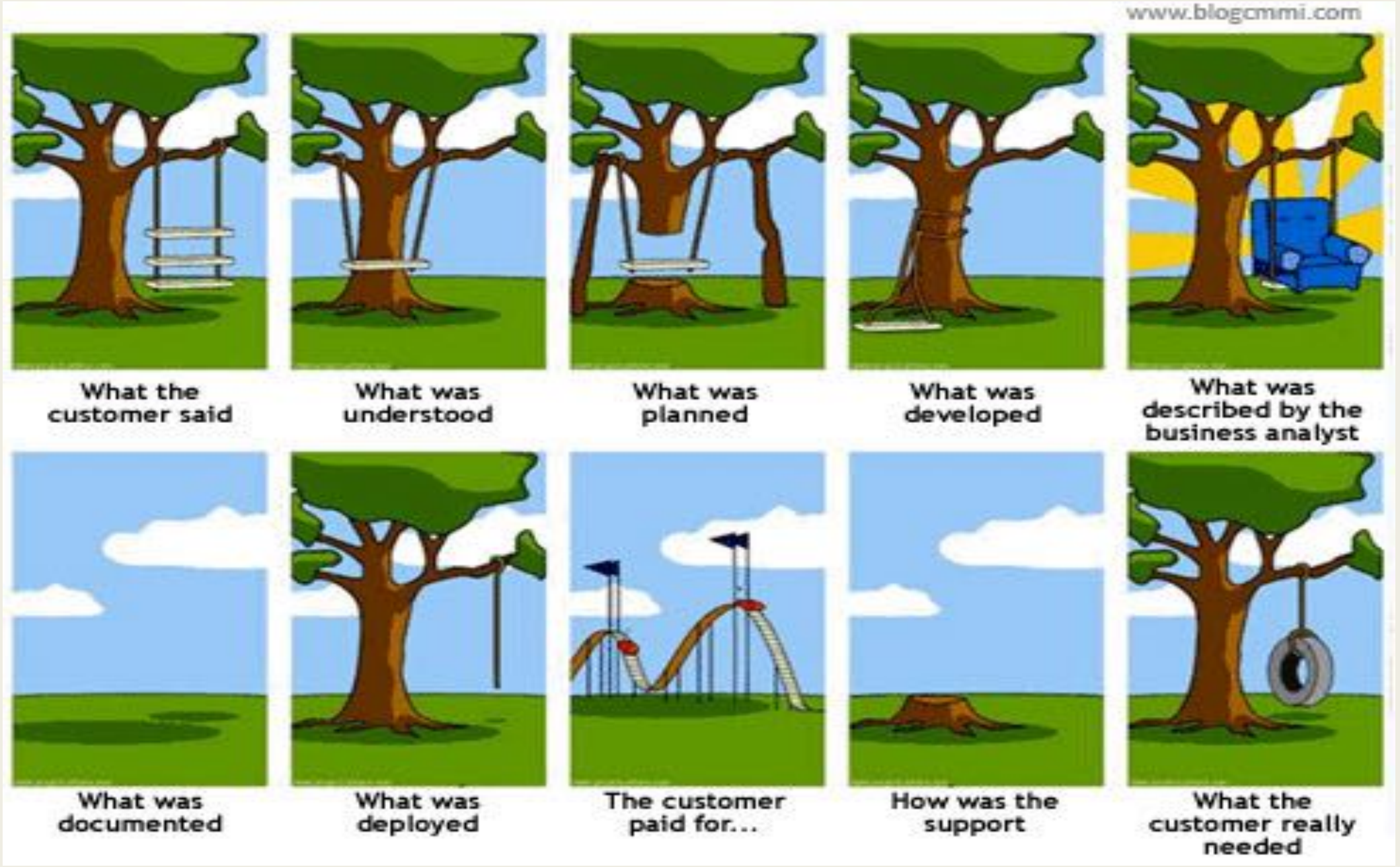
Requirements Example – Bird Cage

- 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?
- Food dishes are steel... OK?
- The perches are steel... OK?
- 2 cm gaps between cage wires... OK?
- 60 cm gaps between cage wires.. OK?
- What kind of birds should like it?
- How can we know the birds like it?
- How many birds should it support?
- Cage has no door... OK?
- Cage has 17 doors, all of which are opened via elaborate puzzles... OK?
- Cage weighs 100 kg... 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

Testers need to understand requirements

- Why? Because they (or something like them) describe the expected behavior!
  - *Expected behavior vs observed behavior is the foundation of testing software*
- Software that does not meet requirements does not do what it is supposed to do!

Requirements Formality / Specificity

- Levels of formality and specificity of the requirements vary
- User stories, notes on a napkin, a very detailed software requirement specification document... all of these can be considered a way of specifying requirements
- I will use traditional "The system shall..." requirements in these slides but the ideas are universal no matter how the requirements are specified

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

- **GOOD:** The system shall persistently 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 testing/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 the functional behavior of the system
  - *The system shall do X [under conditions Y].*
- **Quality Attributes** – Specify the overall qualities of the 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.

Solution: agree upon quantifiable requirements.

Why?

- Can be very subjective
- May relate back to functional requirements
- It's easy for contradictions to arise
- Often difficult to quantify
- No standardized rules for considering them "met"

Converting Qualitative to Quantitative

- **Performance:** transactions per second, response time
- **Reliability:** Mean time between failures
- **Robustness**: Amount of time to restart
- **Portability:** Number of systems targeted, or how long it would take to port
- **Size:** Number of kilobytes, megabytes, etc.
- **Safety:** Number of accidents per year
- **Usability:** Amount of time for training
- **Ease of use**: Number of errors made per day by a user

EXAMPLE

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

You’ve got requirements.
You’re looking for defects.

How?

Develop a test plan!

Formality

- This could be as formal or informal as necessary.
- Think about what you are testing – what level of responsibility / tracking is necessary?

Necessary testing formality varies...

- Throw-away script?
- Development tool?
- Internal website?
- Enterprise software?
- Commercial software?
- Operating system?
- Avionics software?

Testing is context-dependent

- How you test
- How much you test
- What tools you use
- What documentation you provide

- ...All vary based on software context.

Formal Test Plans

- A test plan is a sequence of test cases.
- A test case is the fundamental “unit” of a test plan.

A test case mainly consists of…

- Preconditions
- Execution Steps
- Postconditions

See IEEE 829, "Standard for Software Test Documentation", for more details

Example

Assuming an empty shopping cart, when I click "Buy Widget", the number of widgets in the shopping cart becomes one.

Preconditions: Empty shopping cart
Execution Steps: Click "Buy Widget“
Postconditions: Shopping cart displays one widget

Example

Assuming that the SORT_ASCENDING flag is set, calling the sort method with [9,3,4,2] will return a new array sorted from high to low, i.e., [2,3,4,9].

Precondition: SORT_ASCENDING flag is set
Execution steps: Call .sort method with argument [9,3,4,2]
Postconditions: [2,3,4,9] is returned

We also want to add:

- Identifier: A way to identify the test case
  - Could be a number
  - Often a label, e.g. INVALID-PASSWORD-THREE-TIMES-TEST
- Description: A description of the test case, describing what it is supposed to test.

Test Plan

- These do not always test an entire system
- They may test a subsystem or related piece of functionality
  - Examples:
  - Database Connectivity Test Plan
  - Pop-up Warning Test Plan
  - Pressure Safety Lock Test Plan
  - Regression Test Plan

Pressure Safety Lock Test Plan

LOW-PRESSURE-TEST
HIGH-PRESSURE-TEST
SAFETY-LIGHT-TEST
SAFETY-LIGHT-OFF-TEST
RESET-SWITCH-TEST
RESET-SWITCH2-TEST
FAST-MOVEMENT-TEST
RAPID-CHANGE-TEST
GRADUAL-CHANGE-TEST
MEDIAN-PRESSURE-TEST
LIGHT-FAILURE-TEST
SENSOR-FAILURE-TEST
SENSOR-INVALID-TEST

Test Run – Execution of a test plan.

- Analogy time: class vs object, test plan vs test run
  - The test plan is the structure, but you need to actually execute
- During the test run, the tester executes each test case and sets the status

Defects

- If the test case fails, a defect should be filed
  - Unless the test case has already failed, of course.
  - You usually don’t need to re-file a duplicate defect every time you re-run a failing test case
- Note the level of formality involved will vary based on the domain

Creating a test plan…

- Start top-down: what is a good way to subdivide the system into features (test plans)?
- For a given feature (test plan), what aspects do I want to test?
- For each aspect, what test cases do I want that will hit different equivalence classes / success or failure cases / edge or corner cases / etc.?
- How deep should I go down?
- Try to have test cases be independent of each other, and reproducible!

Traceability Matrices

- How can we verify that our test plan actually tests all of the requirements?
- Simply list all requirements and all test cases which test those requirements
- This is a good indicator of test coverage ... or superfluous tests!

Good Traceability Matrix

REQ1: TEST_CASE_1, TEST_CASE_2
REQ2: TEST_CASE_3
REQ3: TEST_CASE_4, TEST_CASE_7
REQ4: TEST_CASE_5, TEST_CASE_9
REQ5: TEST_CASE_6, TEST_CASE_10

All requirements have at least one test case associated with them; all test cases map to a requirement.

Possibly Problematic Traceability Matrix

REQ1: TEST_CASE_1, TEST_CASE_2
REQ2:
REQ3: TEST_CASE_4, TEST_CASE_7
REQ4: TEST_CASE_5, TEST_CASE_9
REQ5: TEST_CASE_6, TEST_CASE_10

No test case is testing requirement 2!

Possibly Problematic Traceability Matrix

REQ1: TEST_CASE_1, TEST_CASE_2
REQ2: TEST_CASE_3
REQ3: TEST_CASE_4, TEST_CASE_7
REQ4: TEST_CASE_5, TEST_CASE_9
REQ5: TEST_CASE_6, TEST_CASE_10
?????: TEST_CASE_11

What is test case 11 checking?

REQUIREMENTS, TEST PLANS, AND TRACEABILITY MATRICES

Bill Laboon

Twitter: @BillLaboon

Email: laboon@cs.pitt.edu