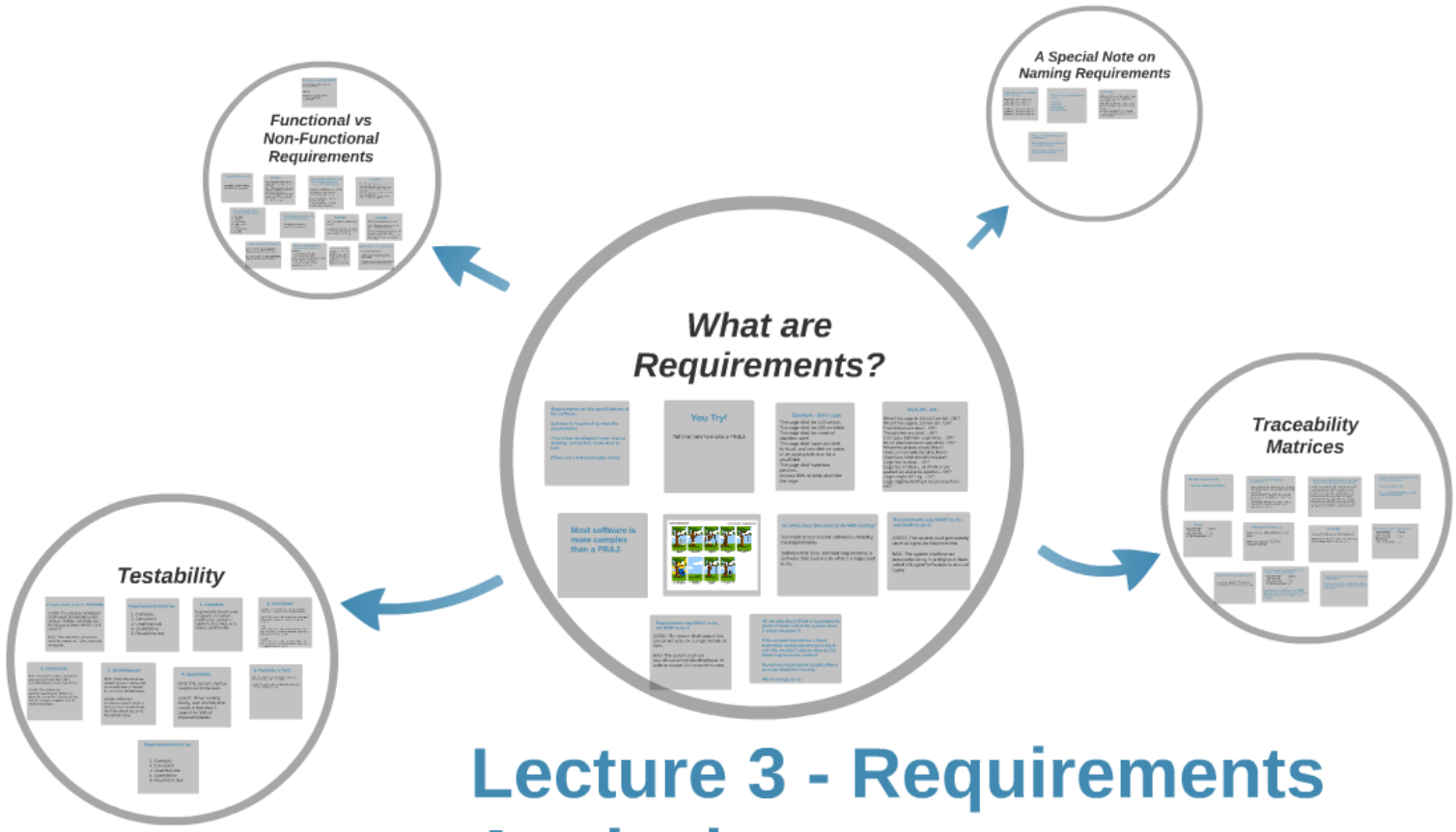
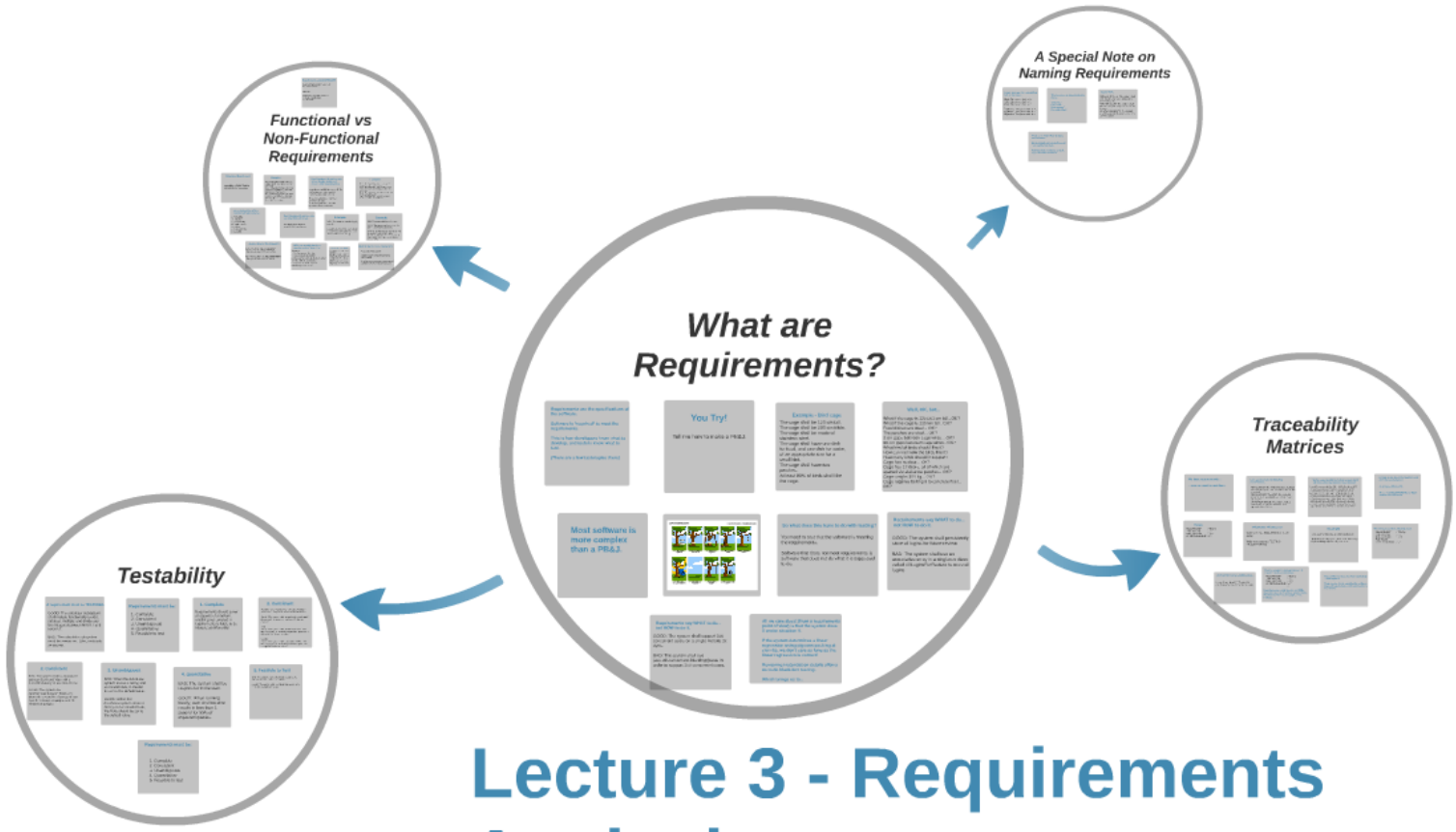


Lecture 3 - Requirements Analysis



Lecture 3 - Requirements Analysis

What are Requirements?

Requirements are the specifications of the software.

Software is *required* to meet the requirements.

This is how developers know what to develop, and testers know what to test.

(There are a few tautologies there)

You Try!

Tell me how to make a PB&J.

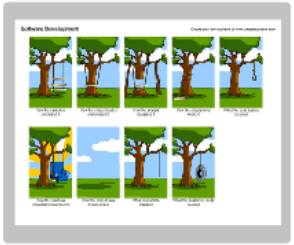
Example - Bird cage

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.

Well, OK, but...

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?
Cage requires bolting it to concrete floor... OK?

Most software is more complex than a PB&J.



So what does this have to do with testing?

You need to test that the software is meeting the requirements.

Software that does not meet requirements is software that does not do what it is supposed to do.

Requirements 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.

Requirements say WHAT to do... not HOW to do it.

GOOD: The system shall support 100 concurrent users on a single Heroku 2x dyno.

BAD: The system shall use java.util.concurrent.BlockingQueue in order to support 100 concurrent users.

All we care about (from a requirements point of view) is that the system does X under situation Y.

If the system determines a linear regression using pigeons pecking at crumbs, we don't care as long as the linear regression is correct!

Removing instantiation details allows us to do black-box testing.

Which brings us to...

Requirements are the specifications of the software.

Software is *required* to meet the requirements.

This is how developers know what to develop, and testers know what to test.

(There are a few tautologies there)

You Try!

Tell me how to make a PB&J.

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.

Well, OK, but...

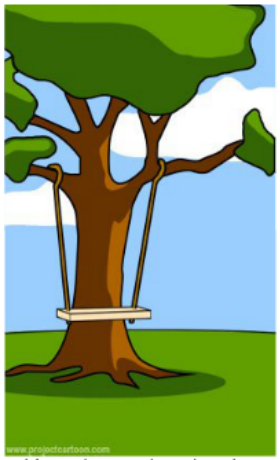
What if the cage is 120.001 cm tall.. OK?
What if the cage is 120 km tall.. OK?
Food dishes are steel... OK?
The perches are steel... OK?
2 cm gaps between cage wires... OK?
60 cm gaps between cage wires.. OK?
What kind of birds should like it?
How can we know the birds like it?
How many birds should 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?
Cage requires bolting it to concrete floor... OK?

Most software is more complex than a PB&J.

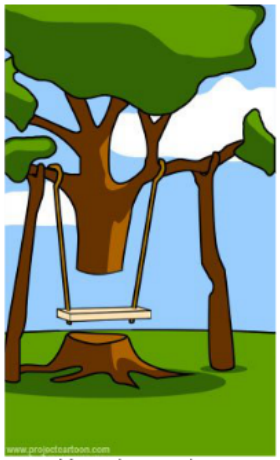
Software Development

Create your own cartoon at www.projectcartoon.com

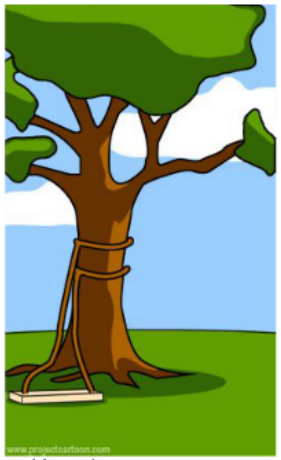
How the customer explained it



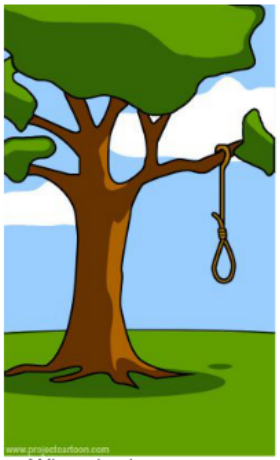
How the project leader understood it



How the analyst designed it

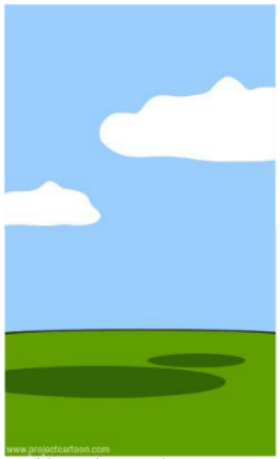


How the programmer wrote it

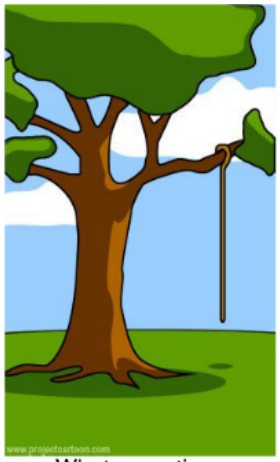


What the beta testers received

How the business consultant described it



How the project was documented



What operations installed

What the customer really needed

So what does this have to do with testing?

You need to test that the software is meeting the requirements.

Software that does not meet requirements is software that does not do what it is supposed to do.

Requirements 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.

Requirements say WHAT to do... not HOW to do it.

GOOD: The system shall support 100 concurrent users on a single Heroku 2x dyno.

BAD: The system shall use java.util.concurrent.BlockingQueue in order to support 100 concurrent users.

All we care about (from a requirements point of view) is that the system does X under situation Y.

If the system determines a linear regression using pigeons pecking at crumbs, we don't care as long as the linear regression is correct!

Removing instantiation details allows us to do black-box testing.

Which brings us to...

Testability

A requirement must be TESTABLE.

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 must be:

1. Complete
2. Consistent
3. Unambiguous
4. Quantitative
5. Feasible to test

1. Complete

Requirements should cover all aspects of a system. Anything not covered in requirements is liable to be interpreted differently!

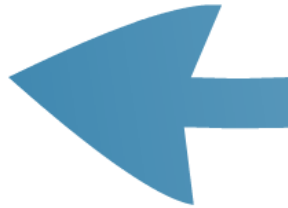
2. Consistent

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."



2. 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.

3. 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.

4. 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.

5. 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 KB data set within 4 hours.

Requirements must be:

1. Complete
2. Consistent
3. Unambiguous
4. Quantitative
5. Feasible to test

A requirement must be TESTABLE.

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 must be:

1. Complete
2. Consistent
3. Unambiguous
4. Quantitative
5. Feasible to test

1. Complete

Requirements should cover all aspects of a system. Anything not covered in requirements is liable to be interpreted differently!

2. 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."

2. 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.

3. 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.

4. 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.

5. 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 KB data set within 4 hours.

Requirements must be:

1. Complete
2. Consistent
3. Unambiguous
4. Quantitative
5. Feasible to test

Requirements sound AWESOME!

Systems Engineers often work with requirements all day.

IEEE 830

"IEEE Recommended Practice for Software Requirements Specifications"

Functional vs Non-Functional Requirements

Functional Requirement

Specifies a FUNCTION or BEHAVIOR of a system.

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.

Non-Functional Requirements (a.k.a. Quality Attributes) (a.k.a. "-ility" requirements)

Specify the OVERALL QUALITIES of the system, not a specific BEHAVIOR or FUNCTION.

Execution Qualities - how the system executes.
Evolution Qualities - how the system evolves over time.

Examples

Req 1 - The system shall be protected against unauthprized access. (Execution)
Req 2 - The system shall have 99.999 (five 9's) uptime and be available during that same time. (Execution)
Req 3 - The system shall be easily extensible and maintainable. (Evolution)
Req 4 - The system shall be portable to other processor architectures. (Evolution)

Some Categories of Non-Functional Requirements

1. Reliability
2. Usability
3. Accessibility
4. Performance
5. Safety
6. Supportability
7. Security

Non-Functional Requirements are often difficult to test.

Solution: agree upon a quantifiable requirement.

Example

BAD: The system must be highly usable.

GOOD: Over 90% of users have no questions using the software after one hour of training.

Example

BAD: The system shall be safe to use.

GOOD: The system shall cause less than one injury per year of operation.

BETTER (but Functional!): The system will immediately cease all operations, excluding WARNING bells and DANGER lights, when internal pressure is greater than 100 PSI.

Another Way to Think About It..

FUNCTIONAL REQUIREMENT
The system must DO something.

NON-FUNCTIONAL REQUIREMENT
The system must BE something.

NFRs are usually harder to specify and test than FRs.

Reasons:
1. Can be very subjective
2. May relate back to FRs
3. It's easy for contradictions to arise
4. Often difficult to quantify
5. No standardized rules for considering them "met"

Convert Qualitative to Quantitative

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

Side Note: User Stories vs Requirements

As a user of the system

I want to log in using the company LDAP server

In order to use the same username and password as my corporate account.

Functional Requirement

Specifies a FUNCTION or BEHAVIOR of a system.

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.

Non-Functional Requirements (a.k.a. Quality Attributes) (a.k.a. "-ility" requirements)

Specify the OVERALL QUALITIES of the system, not a specific BEHAVIOR or FUNCTION.

Execution Qualities - how the system executes.
Evolution Qualities - how the system evolves over time.

Examples

Req 1 - The system shall be protected against unauthorized access. (Execution)
Req 2 - The system shall have 99.999 (five 9's) uptime and be available during that same time. (Execution)
Req 3 - The system shall be easily extensible and maintainable. (Evolution)
Req 4 - The system shall be portable to other processor architectures. (Evolution)

Some Categories of Non-Functional Requirements

1. Reliability
2. Usability
3. Accessibility
4. Performance
5. Safety
6. Supportability
7. Security

Non-Functional Requirements are often difficult to test.

Solution: agree upon a quantifiable requirement.

Example

BAD: The system must be highly usable.

GOOD: Over 90% of users have no questions using the software after one hour of training.

Example

BAD: The system shall be safe to use.

GOOD: The system shall cause less than one injury per year of operation.

BETTER (but Functional!): The system will immediately cease all operations, excluding WARNING bells and DANGER lights, when internal pressure is greater than 100 PSI.

Another Way to Think About It..

FUNCTIONAL REQUIREMENT
The system must DO something.

NON-FUNCTIONAL REQUIREMENT
The system must BE something.

NFRs are usually harder to specify and test than FRs.

Reasons:

1. Can be very subjective
2. May relate back to FRs
3. It's easy for contradictions to arise
4. Often difficult to quantify
5. No standardized rules for considering them "met"

Convert 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. used
Safety: Number of accidents per year
Usability: Amount of time for training
Ease of use: Number of errors made per day by a user

Side Note: User Stories vs Requirements

As a user of the system

I want to log in using the company LDAP server

In order to use the same username and password as my corporate account.

Requirements sound AWESOME!

Systems Engineers often work with requirements all day.

IEEE 830

"IEEE Recommended Practice for Software Requirements Specifications"

A Special Note on Naming Requirements

Long, long ago... in a computing lab far, far away...

REQ1: The system shall do X.
REQ2: The system shall do Y.
REQ3: The system shall do Z.
....
REQ98420: The system shall do א.
REQ98421: The system shall do ב.
REQ98422: The system shall do ג.

This becomes an unmaintainable mess.

Ordering?
Removal?
Modification?
Remembering?

Easier Way...

FUN-SYS-FPCALC: The system shall calculate floating point value using IEEE standards.
FUN-PER-LOGIN: The system shall persist all login attempts to permanent storage.
NF-SYS-RELIABILITY: The system shall be available 99.999% of the time (five 9's uptime).

There is no "right way" to name requirements.

But naming them is usually easier than numbering them.

You may have numbers, as well, e.g., FUN-SYS-FPCALC-2.

Long, long ago... in a computing lab far, far away...

REQ1: The system shall do X.
REQ2: The system shall do Y.
REQ3: The system shall do Z.

....

REQ98420: The system shall do א.
REQ98421: The system shall do ב.
REQ98422: The system shall do ג.

This becomes an unmaintainable mess.

Ordering?
Removal?
Modification?
Remembering?

Easier Way...

FUN-SYS-FPCALC: The system shall calculate floating point value using IEEE standards.
FUN-PER-LOGIN: The system shall persist all login attempts to permanent storage.
NF-SYS-RELIABILITY: The system shall be available 99.999% of the time (five 9's uptime).

There is no "right way" to name requirements.

But naming them is usually easier than numbering them.

You may have numbers, as well, e.g., FUN-SYS-FPCALC-2.

Traceability Matrices

We have requirements...

...now we need to test them.

Let's say you have the following requirements...

FUN-LIGHTON: The WARNING light shall turn on when the system has less than 5 MB of disk space left.
FUN-LIGHTOFF: The WARNING light shall turn off when the system has 5 MB or more disk space.
NF-PERFORMANCE: The system shall be near real-time, with all indicators being updated within 250 ms.

You've come up with the following tests (we'll talk more about test planning next lecture)

1. Set free space to 100 MB. Light should be OFF.
2. Set free space to 10 MB. Light should be OFF.
3. Set free space to 2 MB. Light should be ON.
4. Set free space to 0 MB. Light should be ON.
5. Set free space to 6 MB. Light should be OFF.
6. Set free space to 5.1 MB. Reduce to 4.9 MB. Light should turn ON within 250 ms of reduction.
7. Set free space to 4.9 MB. Increase to 5.1 MB. Light should turn OFF within 250 ms of increase.

Q: How do you know that you've tested all of the requirements?

A: A traceability matrix.

These map REQUIREMENTS to TEST CASES and vice-versa.

Example

REQUIREMENT	: TEST(S)
FUN-LIGHTON	: 3,4
FUN-LIGHTOFF	: 1,2,5
NF-PERFORMANCE	: 6,7

What Does This Buy Us?

Make sure ALL REQUIREMENTS are tested

Make sure tests are TESTING REQUIREMENTS

Example

Let's assume there was another requirement -

FUN-RESET - The system shall reset itself every night at midnight (12:00 AM) local time.

We now can see we're missing a test!

REQUIREMENT	: TEST(S)
FUN-LIGHTON	: 3,4
FUN-LIGHTOFF	: 1,2,5
FUN-RESET:	:
NF-PERFORMANCE	: 6,7

Or what if we had an additional test...

8. User types "poodle". The system should respond "BARK!" and shut down.

There's no point in having this test. It doesn't match a requirement.

REQUIREMENT	: TEST(S)
FUN-LIGHTON	: 3,4
FUN-LIGHTOFF	: 1,2,5
NF-PERFORMANCE	: 6,7
	: 8

Note that some sanity checks and NFRs may not match up directly to requirements. Functional tests should always do so, however.

Traceability matrices are often reported up to management.

They are also key to ensuring the software meets all requirements and testing effort is not wasted.

We have requirements...

...now we need to test them.

Let's say you have the following requirements...

FUN-LIGHTON: The WARNING light shall turn on when the system has less than 5 MB of disk space left.
FUN-LIGHTOFF: The WARNING light shall turn off when the system has 5 MB or more disk space.
NF-PERFORMANCE: The system shall be near real-time, with all indicators being updated within 250 ms.

You've come up with the following tests (we'll talk more about test planning next lecture)

1. Set free space to 100 MB. Light should be OFF.
2. Set free space to 10 MB. Light should be OFF.
3. Set free space to 2 MB. Light should be ON.
4. Set free space to 0 MB. Light should be ON.
5. Set free space to 6 MB. Light should be OFF.
6. Set free space to 5.1 MB. Reduce to 4.9 MB. Light should turn ON within 250 ms of reduction.
7. Set free space to 4.9 MB. Increase to 5.1 MB. Light should turn OFF within 250 ms of increase.

Q: How do you know that you've tested all of the requirements?

A: A traceability matrix.

These map REQUIREMENTS to TEST CASES and vice-versa.

Example

REQUIREMENT	: TEST(S)
FUN-LIGHTON	: 3,4
FUN-LIGHTOFF	: 1,2,5
NF-PERFORMANCE	: 6,7

What Does This Buy Us?

Make sure ALL REQUIREMENTS are tested

Make sure tests are TESTING REQUIREMENTS

Example

Let's assume there was another requirement -

FUN-RESET - The system shall reset itself every night at midnight (12:00 AM) local time.

We now can see we're missing a test!

REQUIREMENT	: TEST(S)
FUN-LIGHTON	: 3,4
FUN-LIGHTOFF	: 1,2,5
FUN-RESET:	:
NF-PERFORMANCE	: 6,7

Or what if we had an additional test...

8. User types "poodle". The system should respond "BARK!" and shut down.

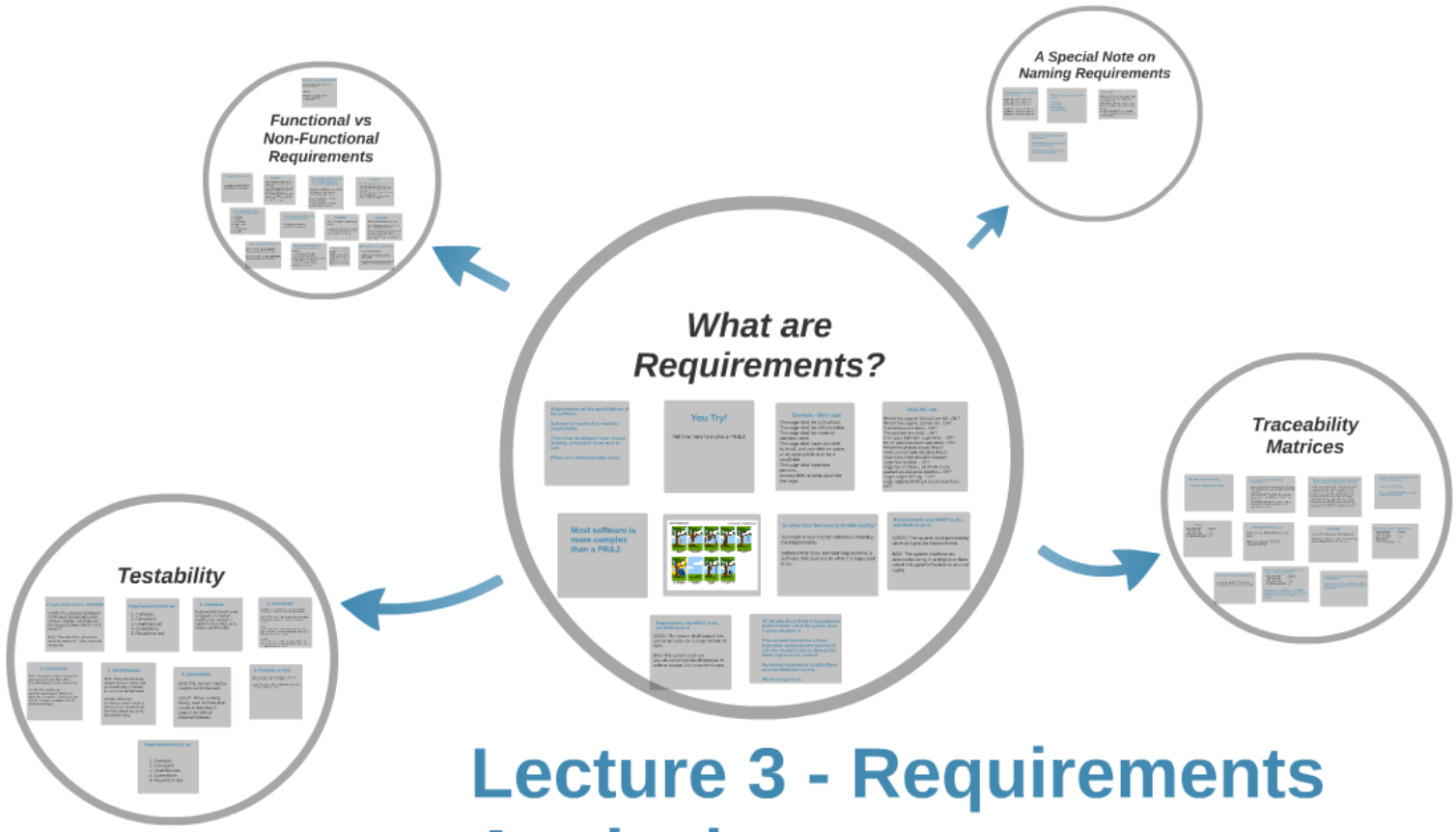
There's no point in having this test. It doesn't match a requirement.

REQUIREMENT	: TEST(S)
FUN-LIGHTON	: 3,4
FUN-LIGHTOFF	: 1,2,5
NF-PERFORMANCE	: 6,7
	: 8

Note that some sanity checks and NFRs may not match up directly to requirements. Functional tests should always do so, however.

Traceability matrices are often reported up to management.

They are also key to ensuring the software meets all requirements and testing effort is not wasted.



Lecture 3 - Requirements Analysis