

*M. Weintraub and
F. Tip*

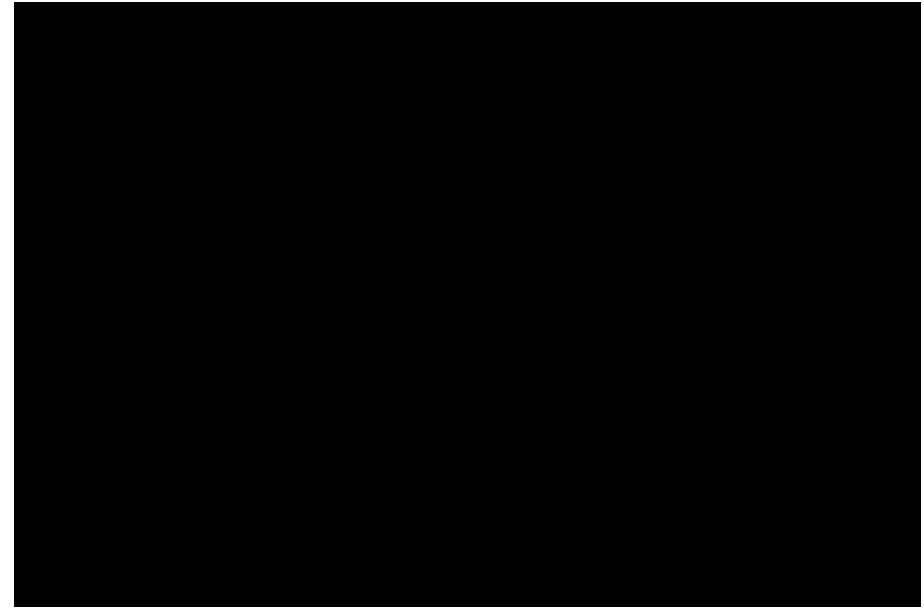
TESTING STRATEGIES

Thanks go to Andreas Zeller for allowing
incorporation of his materials

SOFTWARE QUALITY ASSURANCE (AKA TESTING)

Processes and procedures
aiming to **assess the quality,
performance, or reliability
of the software**

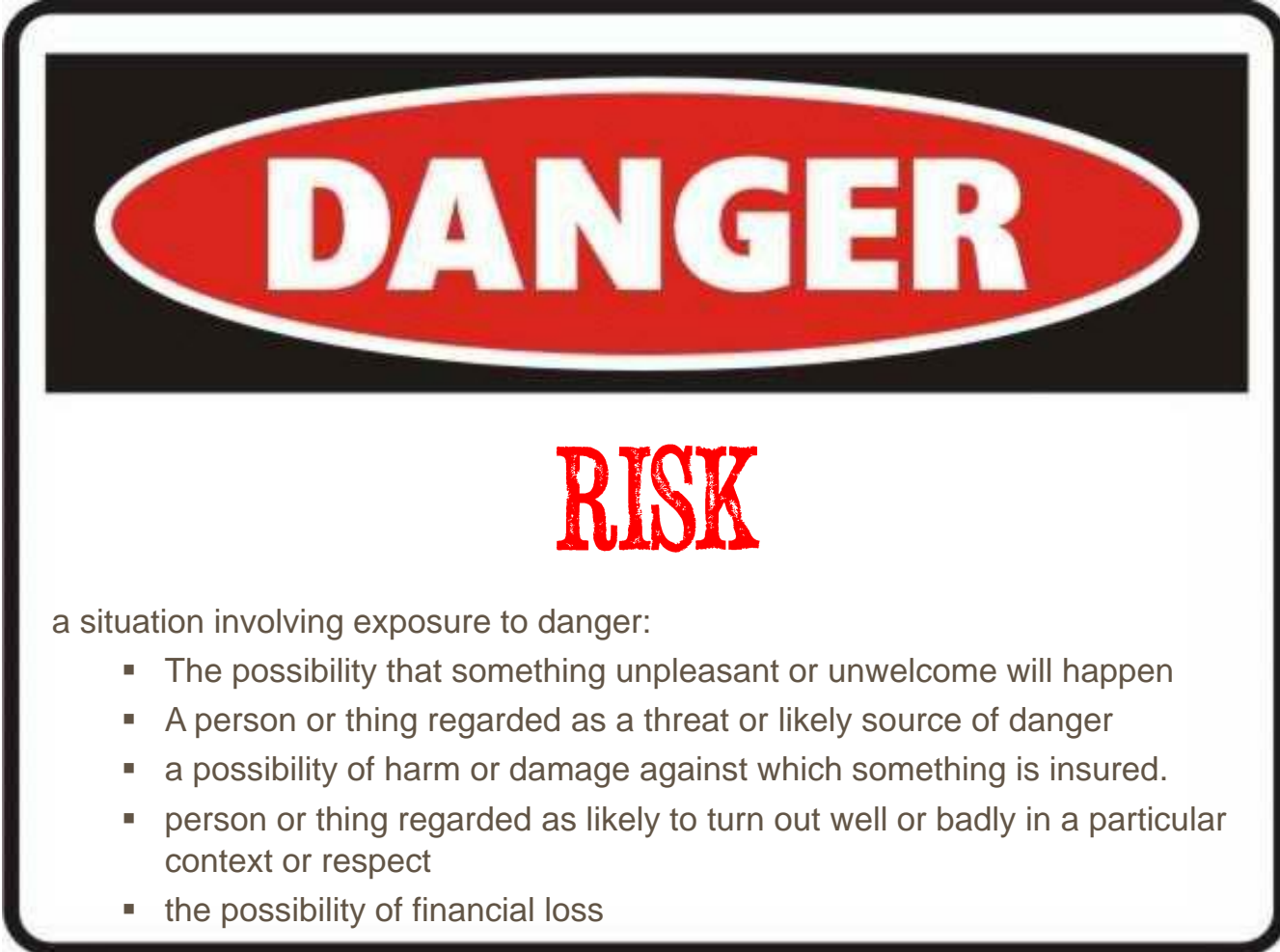
ideally before it is delivered.



Доверяй, но проверяй - *Trust, but verify*

(otherwise, you are just gambling)

WHY SQA? TO MINIMIZE RISK



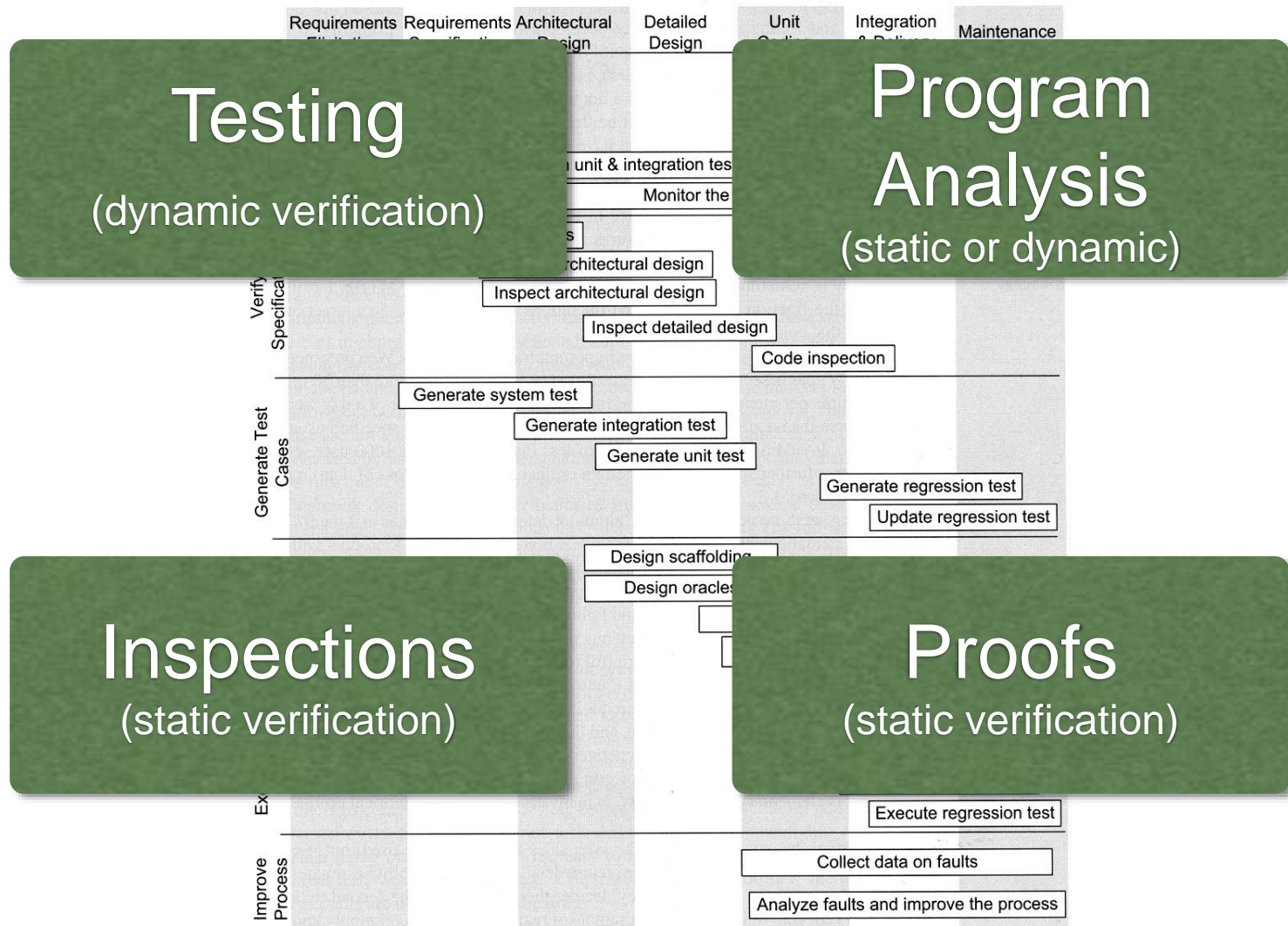
DANGER

RISK

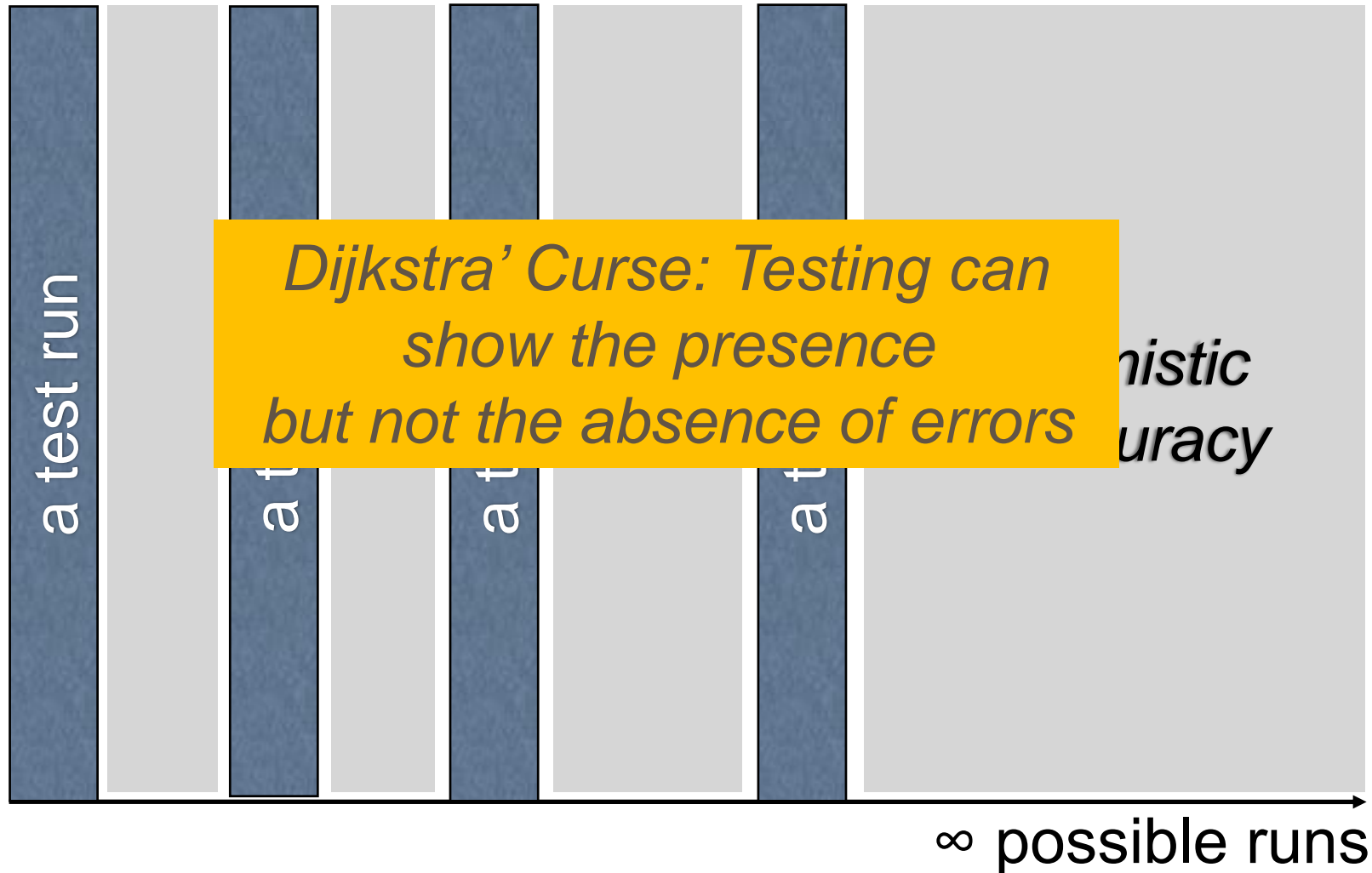
a situation involving exposure to danger:

- The possibility that something unpleasant or unwelcome will happen
- A person or thing regarded as a threat or likely source of danger
- a possibility of harm or damage against which something is insured.
- person or thing regarded as likely to turn out well or badly in a particular context or respect
- the possibility of financial loss

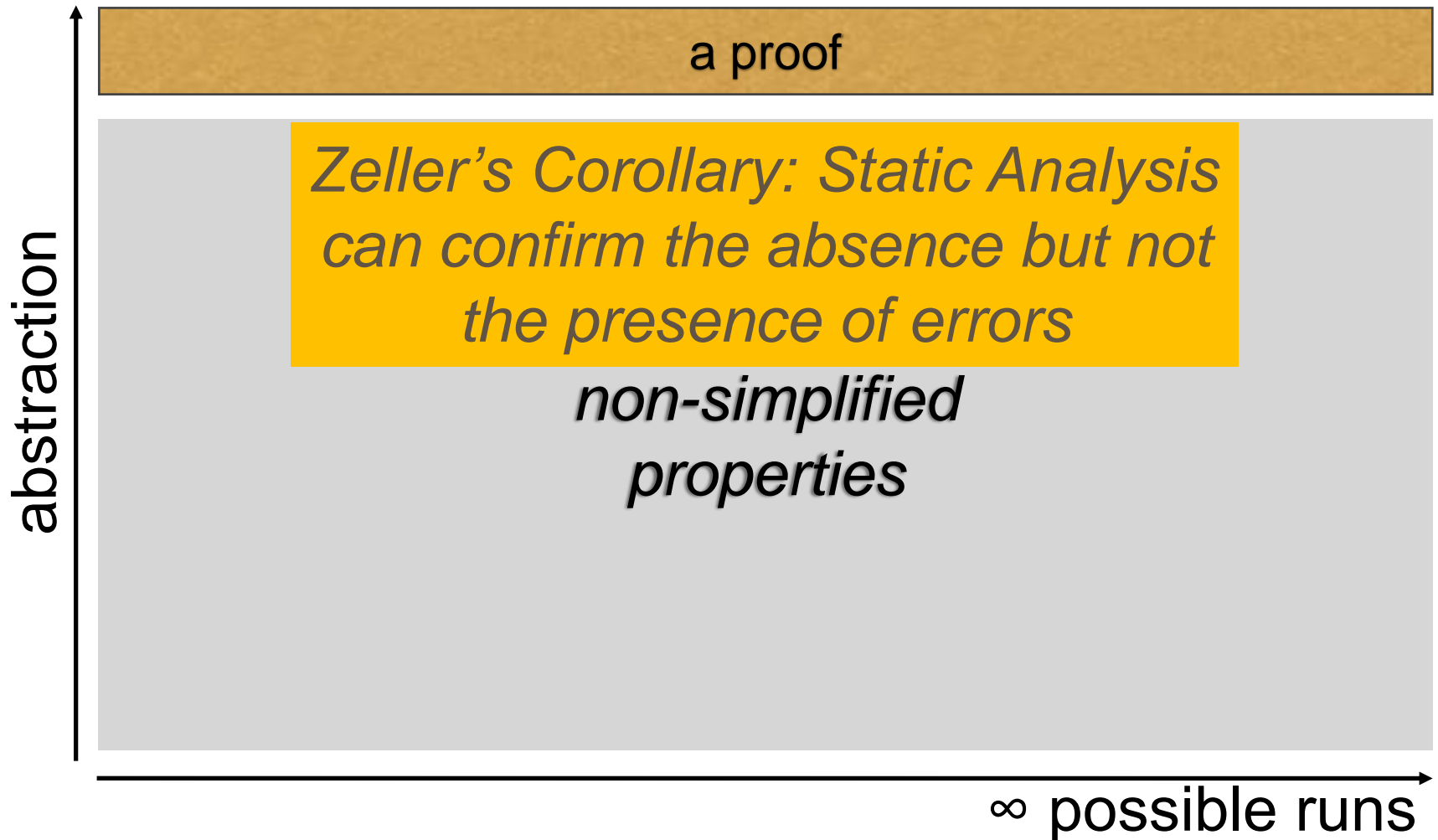
RECALL FROM BEFORE – THESE ARE OUR TECHNIQUES FOR EVALUATING SOFTWARE



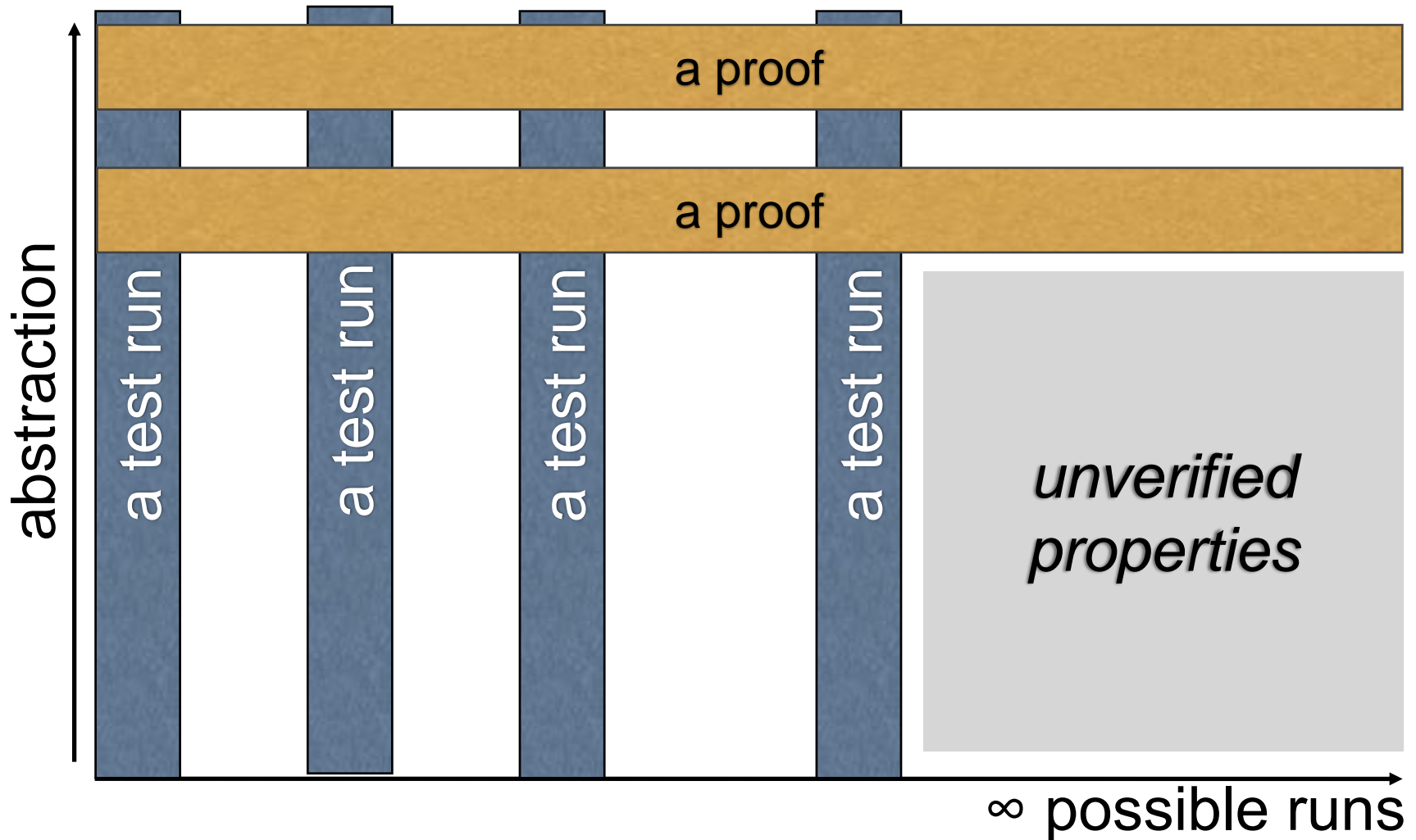
THE CURSE OF FUNCTIONAL TESTING



ITS STRUCTURAL TESTING COROLLARY



COMBINING METHODS



WHY IS SOFTWARE VERIFICATION HARD?

- Many different quality requirements
- Evolving (and deteriorating) structure
- Inherent non-linearity
- Uneven distribution of faults



```
/*===== info output on file handle */
OUTPUT_BYTE(order1);
OUTPUT_BYTE(order2);
//output upper byte then upper byte
OUTPUT_BYTE(h>>BYTE_SIZE);
OUTPUT_BYTE(h);
OUTPUT_BYTE(w>>BYTE_SIZE);
OUTPUT_BYTE(w);

order1 = order>>4;
order2 = order & 15;

#ifdef TRACE
if (!(fpm = fopen("ppmenc.doc", "wb"))
{
    fprintf(stderr, " \n Error: Ca
    exit(2);
}
#endif

/* allocate 'order+1' elements
... is used to stor
```


WHY IS SOFTWARE VERIFICATION HARD?

- Many different quality requirements
- Evolving (and deteriorating) structure
- Inherent non-linearity
- Uneven distribution of faults



If an elevator can safely carry a load of 1000 kg, it can also safely carry any smaller load

A photograph of a spiral-bound notebook with handwritten code snippets. The code includes comments in C, such as "/*some info output on file handle*/", and a function definition for "OUTPUT_BYTE". It also shows a conditional compilation block for "TRACE" that uses "fopen" and "fprintf" to write to a file named "ppmenc.doc".

```
/*some info output on file handle*/
OUTPUT_BYTE(handle);
OUTPUT_BYTE(handle);
OUTPUT_BYTE(handle); //line of symbols in source
//output upper byte then upper byte
OUTPUT_BYTE(h>>BYTE_SIZE);
OUTPUT_BYTE(h);
OUTPUT_BYTE(w>>BYTE_SIZE);
OUTPUT_BYTE(w);

order1 = order>>4;
order2 = order & 15;

#ifdef TRACE
if (!(fpm = fopen("ppmenc.doc", "wb"))
{
    fprintf(stderr, " \n Error: Ca
    exit(2);
}
#endif

/* allocate 'order+1' elements
... is used to stor
```

WHY IS SOFTWARE VERIFICATION HARD?

- Many different quality requirements
- Evolving (and deteriorating) structure
- Inherent non-linearity
- Uneven distribution of faults

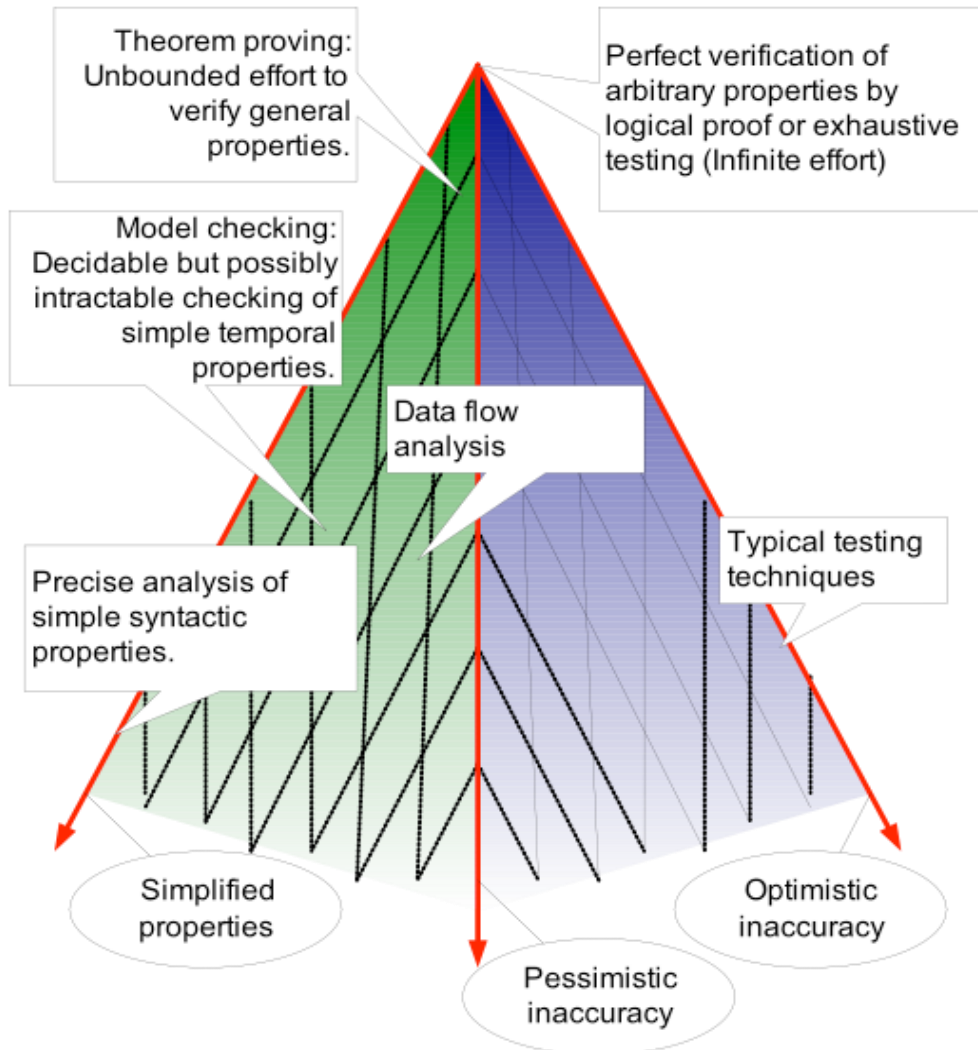


If an elevator can safely carry a load of 1000 kg, it can also safely carry any smaller load



If a procedure correctly sorts a set of 256 elements, it may fail on a set of 255 or 53 elements, as well as on 257 or 1023

A TESTING PROGRAM INVOLVES TRADE-OFFS



- We can be inaccurate (optimistic or pessimistic)
- or we can simplify properties...
- but you cannot have it all!

TYPICAL STRATEGY FOR JUDGING WHEN YOU ARE DONE

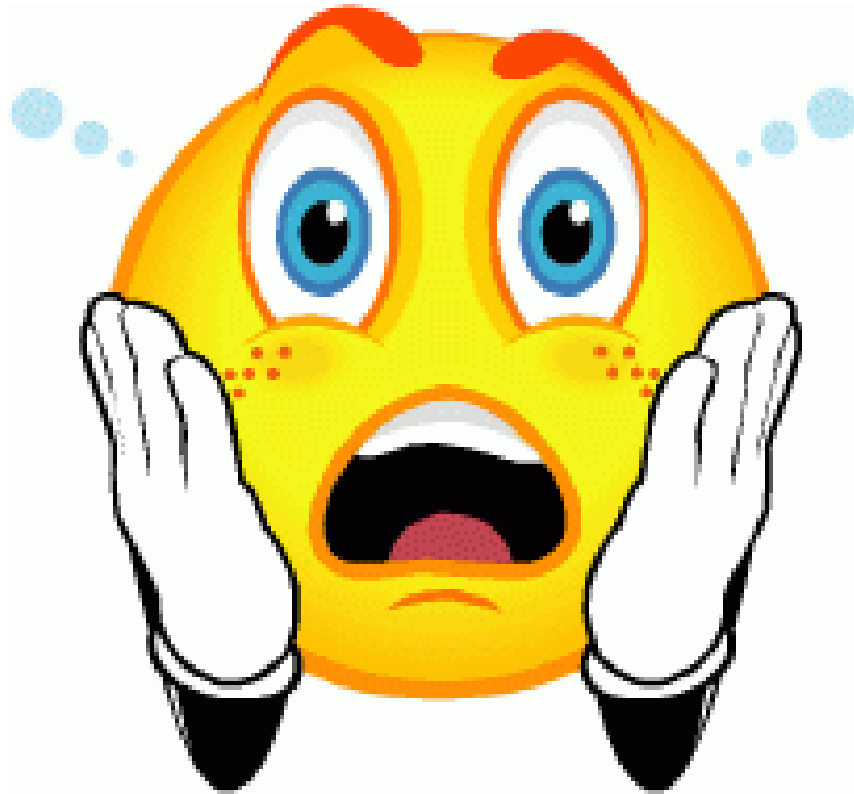
TYPICAL STRATEGY FOR JUDGING WHEN YOU ARE DONE



We built it!
ERGO,
FACTUM!

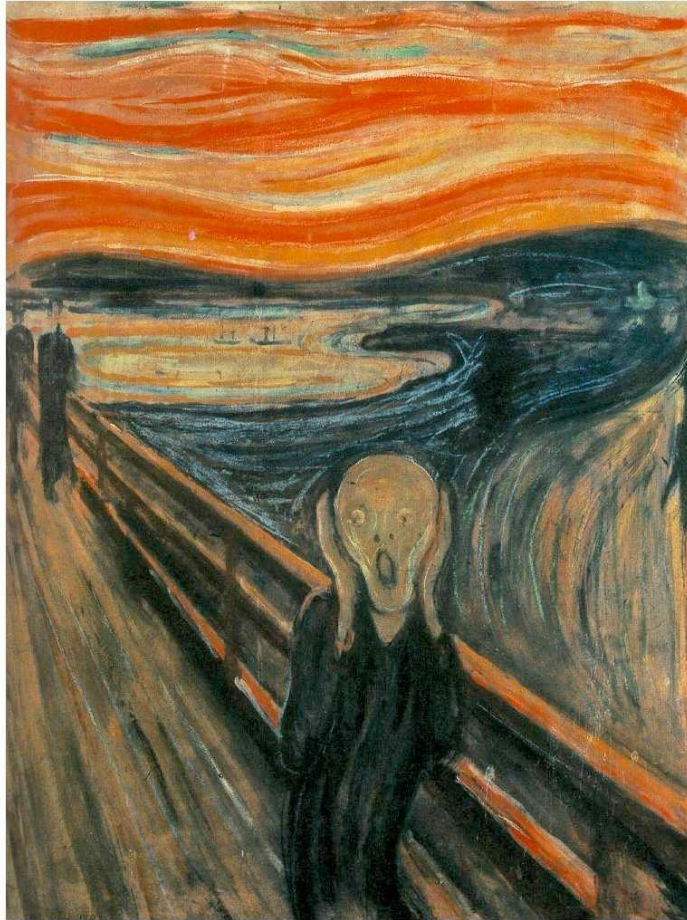
factum: Latin for it's done

**IF YOUR JOB/MONEY DEPENDED ON SUCCESSFUL
DEPLOYMENT, SHOULD YOU DEPLOY THE SYSTEM?**

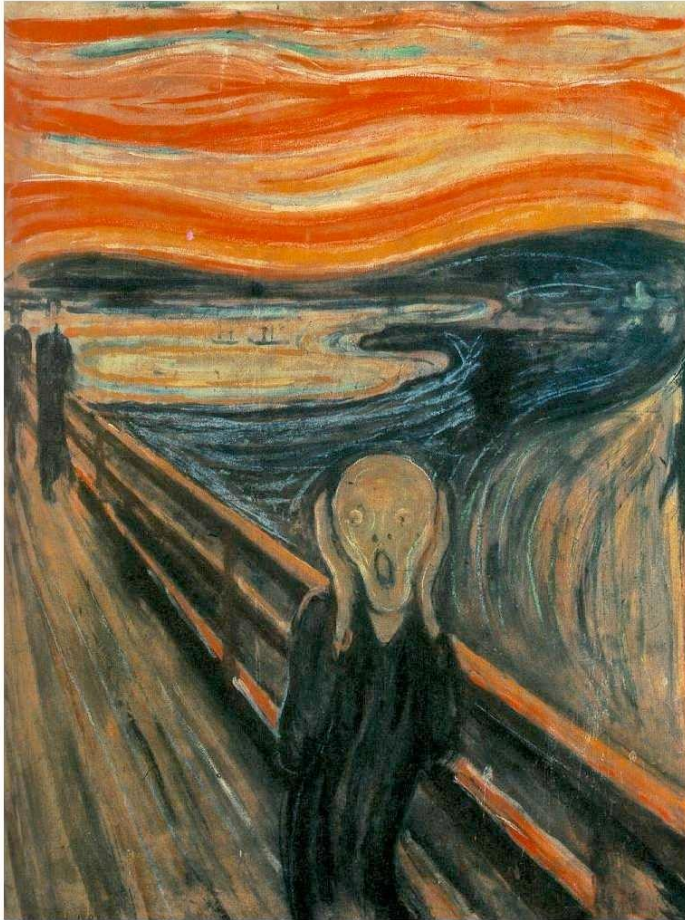


ERR. WELL...

DEPLOYMENT SHOULDN'T BE A HORROR SHOW



BUT IT STILL CAUSES FUD IN MANAGEMENT



FUD = Fear, Uncertainty, and Doubt

SIDEBAR: QUALITY AND RELIABILITY ARE ACTUALLY DIFFERENT

Quality Assurance

Whether a software component or system produces the expected/correct/accepted behavior or output relationship given a set of inputs

Assessing features of the software (UX)

Reliability

Probability of failure-free software operation for a specified duration in a particular environment

Cool phrases

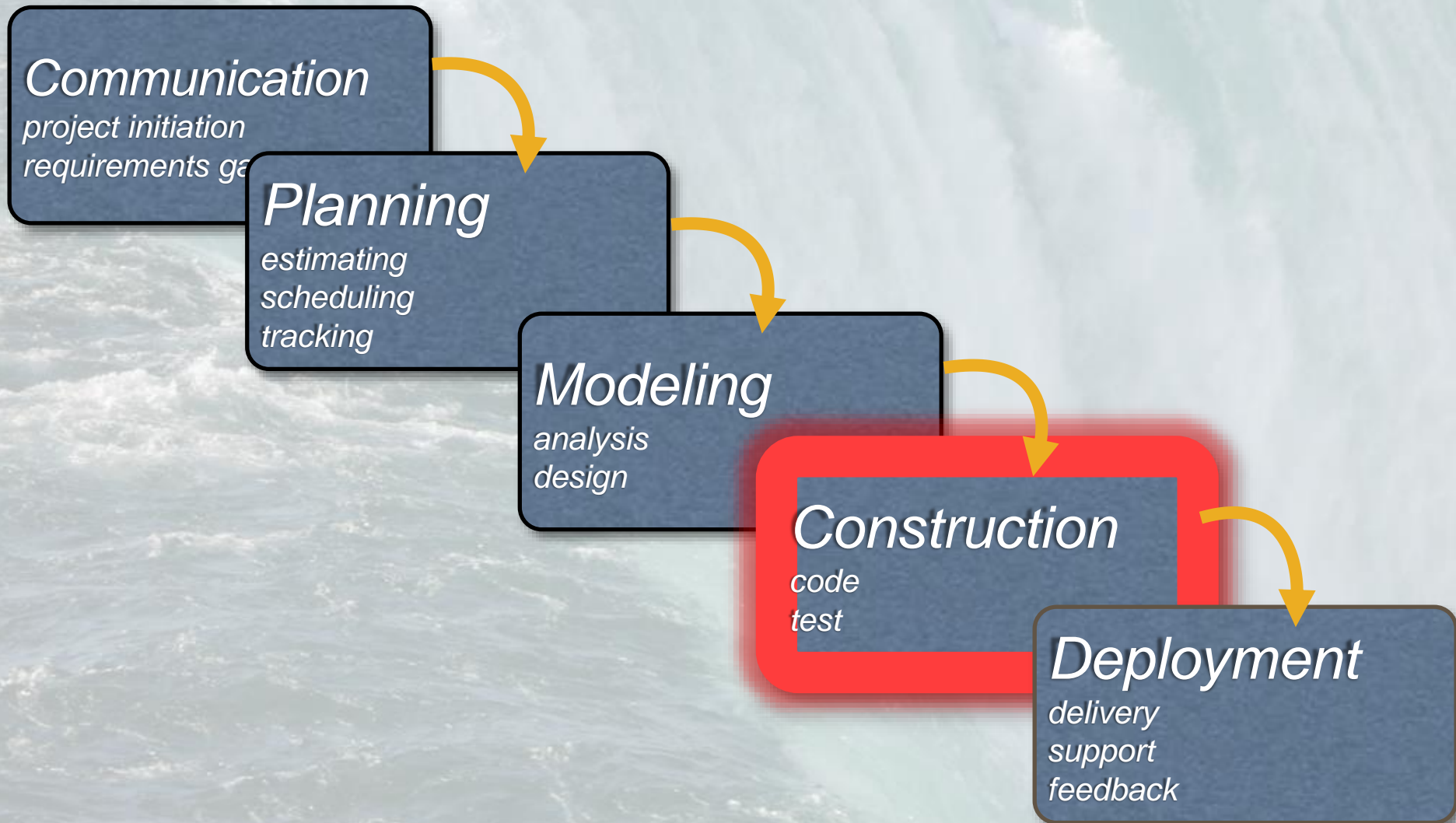
Five 9's (5.25 min/yr downtime)

No down-time

Continuous operation



TO ANSWER THIS QUESTION, WE NEED TO FOCUS ON THE CONSTRUCTION PHASE



IMPORTANT TERMS FOR JUDGING A SYSTEM: VALIDATION AND VERIFICATION (AKA V & V)

Validation

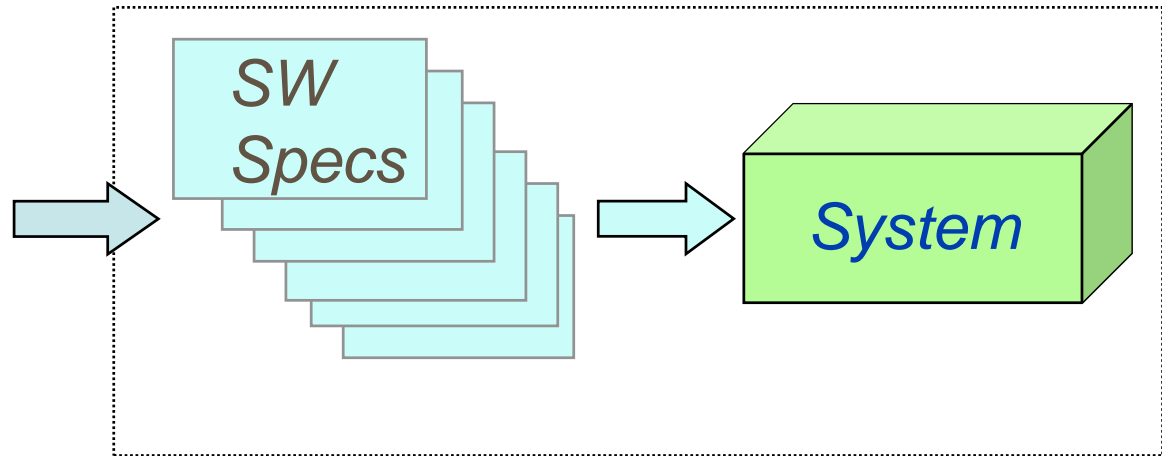
Ensuring that software has been built according to customer requirements

*Are we building
the right product
or service?*

Verification

Ensuring that software correctly implements a specific function

*Are we building
the product or
service right?*

[illegible]

*Actual
Requirements*

Validation

Involves usability testing, user feedback, & product trials

Verification

Includes testing, code inspections, static analysis, proofs

VALIDATION

“if a user presses a request button at floor i , an available elevator must arrive at floor i soon”



*not verifiable, but can be validated
by interviewing subjects*

VALIDATION VERSUS

“if a user presses a request button at floor i , an available elevator must arrive at floor i soon”



VERIFICATION

“if a user presses a request button at floor i , an available elevator must arrive at floor i **within 30 seconds**”



*Can be verified through
objective testing*

CORE QUESTIONS

When does V&V start? When is it done?

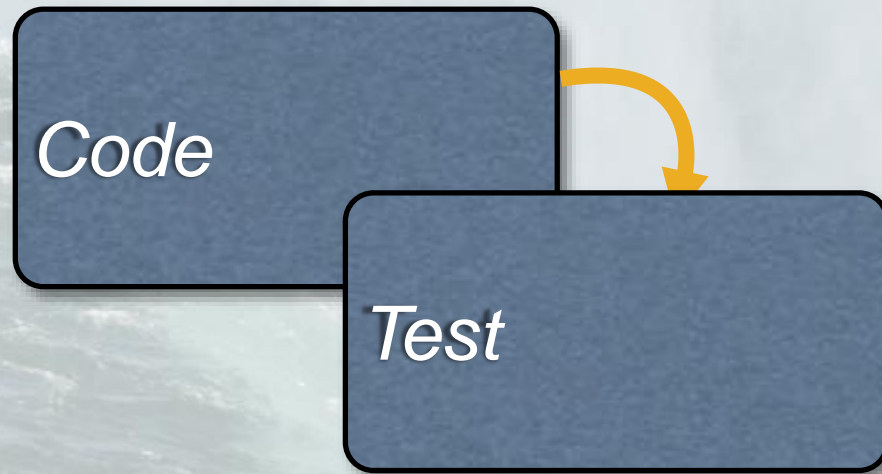
Which techniques should be applied?

How do we know a product is ready?

How can we control the quality of successive releases?

How can we improve development?

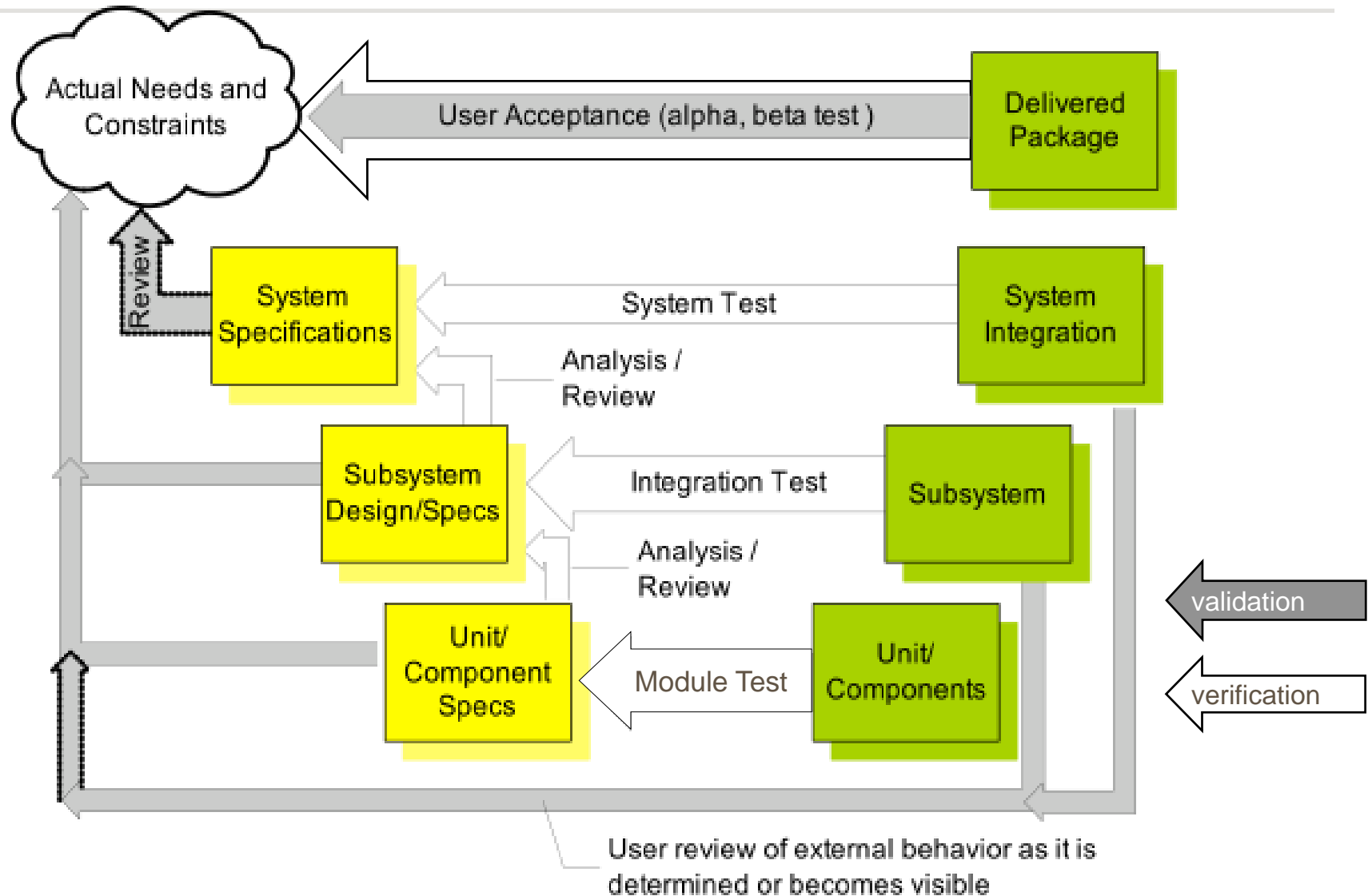
WATERFALL SEPARATED CODING FROM TESTING



FIRST CODE, THEN TEST

1. Developers on software should “test as they go.”
2. Software should be “thrown over a wall” to strangers who will test mercilessly.
3. Testers should be involved with the project only when testing is about to begin.

V MODEL

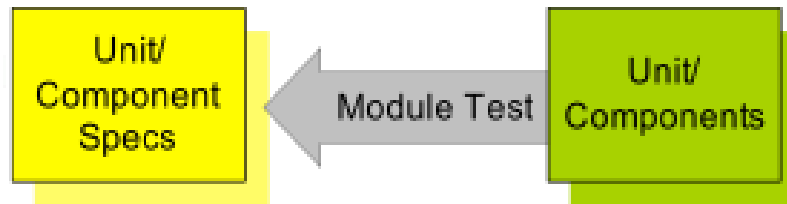


UNIT TESTS

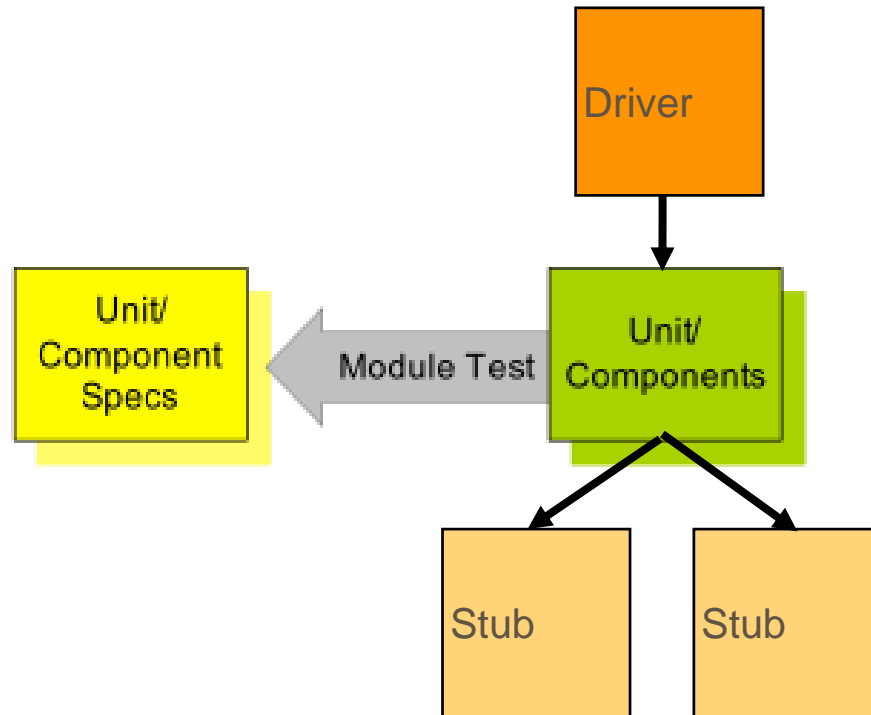
Aims to uncover errors at module boundaries

Typically written by programmer herself

Should be completely automatic
(which enables regression)



TESTING COMPONENTS: STUBS AND DRIVERS



A driver exercises a module's functions

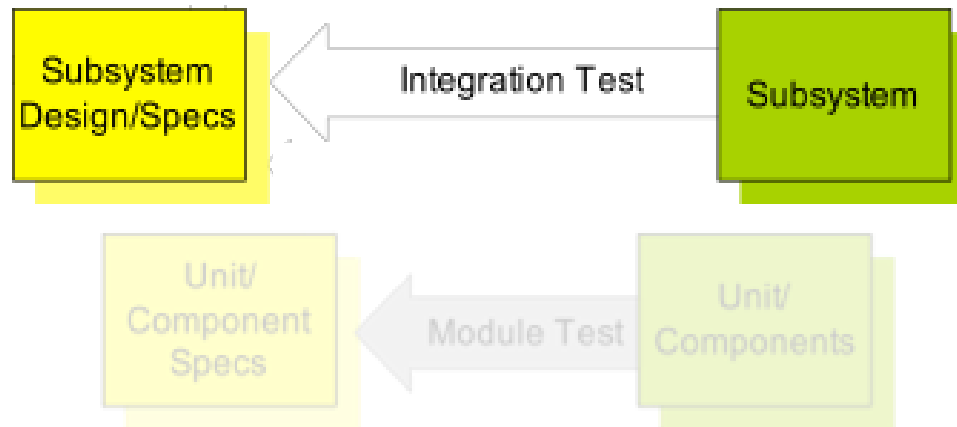
A stub simulates not-yet-ready modules

Frequently realized as mock objects

PUTTING THE PIECES TOGETHER: INTEGRATION TESTS

General idea: Construct software while conducting tests

Choices: Big Bang or Incremental Construction



BIG BANG APPROACH

All components are combined
in advance

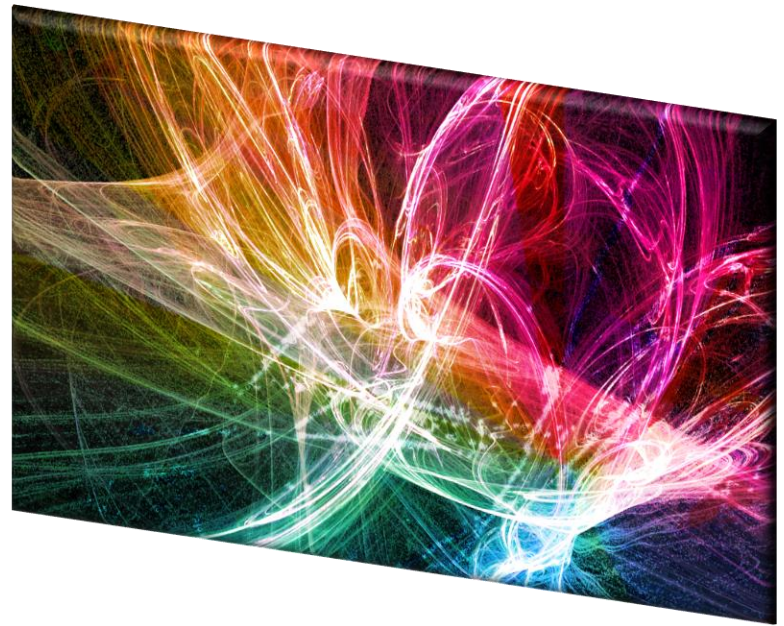
The entire program is tested
as a whole



BIG BANG APPROACH

All components are combined
in advance

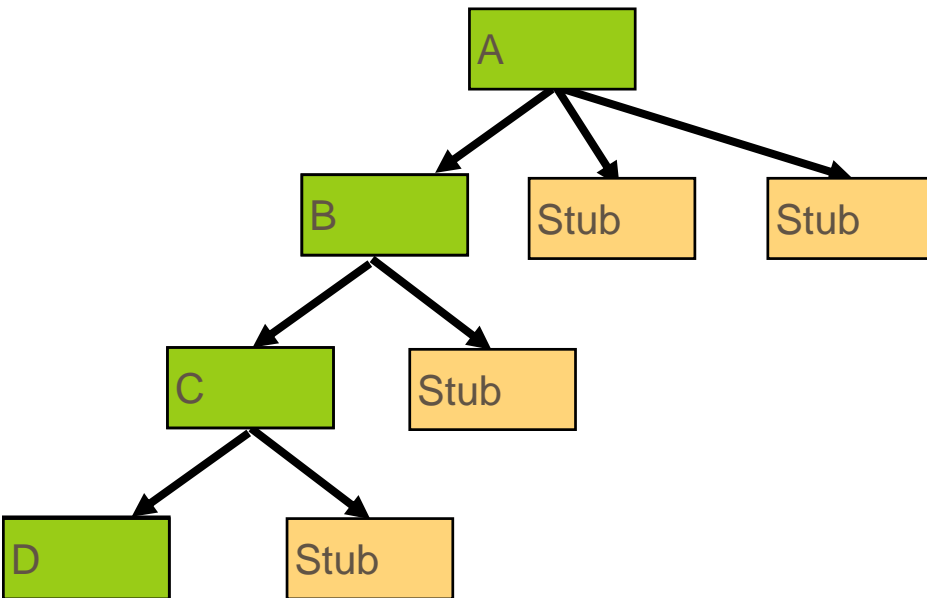
The entire program is tested
as a whole



CHAOS RESULTS!

*For every failure, the entire program must be taken into
account*

TOP-DOWN INTEGRATION



Top module is tested with stubs (and then used as driver)

Stubs are replaced one at a time (“depth first”)

As new modules are integrated, tests are re-run

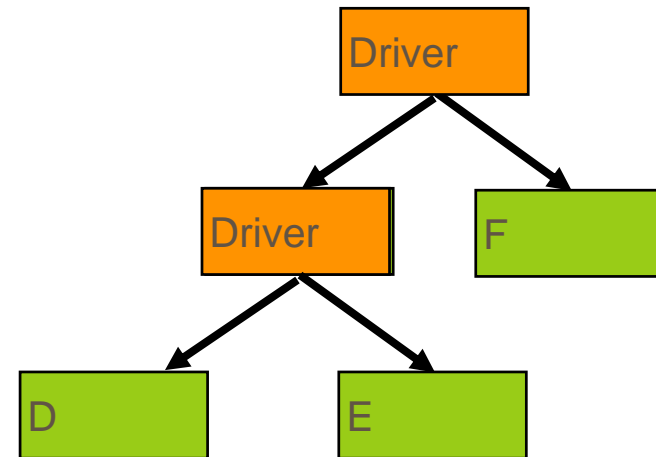
Allows for early demonstration of capability

BOTTOM-UP INTEGRATION

Bottom modules implemented
first and combined into
clusters

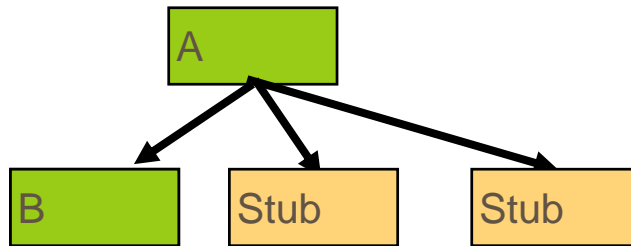
Drivers are replaced one at a
time

Removes the need for
complex stubs

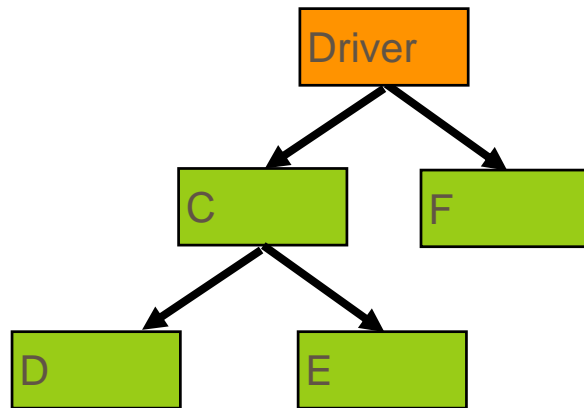


*Allows for early demonstration of capability,
but it may be hard to see the forest*

BEST OF BOTH: SANDWICH INTEGRATION



Combines bottom-up and top-down integration



Top modules tested with stubs, bottom modules with drivers

TETO PRINCIPLE

*Test **E**arly, Test **O**ften*

WHO TESTS THE SOFTWARE?



Developer

understands the system

but will test gently

driven by delivery



Independent Tester

must learn about system

will attempt to break it

driven by quality

ACTUALLY EVERYONE IS A TESTER!

Experienced Outsiders and Clients

Good for finding gaps missed by developers, especially domain specific items

Inexperienced Users

Good for illuminating other, perhaps unintended uses/errors

Mother Nature

*Always finds the hidden flaw!
(usually waits for the boss or an
important client/customer demo
before announcing herself)*



THE IDEAL TESTER

*A good tester should be
creative and destructive
– even sadistic in places.*

- Gerald Weinberg, “The psychology of computer programming”



CONFLICT BETWEEN DEV AND TEST



There is a natural tension

Dev is incented by release

Test often is a barrier to release

Both actually (usually?) want
the same thing

MUTUAL RESPECT



There is a natural tension

Dev is incented by release

Test often is a barrier to release

Both want the same thing

**QUALITY DEPLOYED
CODE**

SCREEN BLINDNESS

A developer is unsuited to test his or her code.

As humans want to be honest with themselves, developers are blindfolded with respect to their own mistakes.

“seen again and again in every project” (Endres/Rombach)

From Gerald Weinberg, “The psychology of computer programming”



DEVELOPERS SHOULD BE TESTERS



Functional testing enables developers to be testers

- Develop tests *before* code
 - *No code: no bias, no ego*

Reviews!

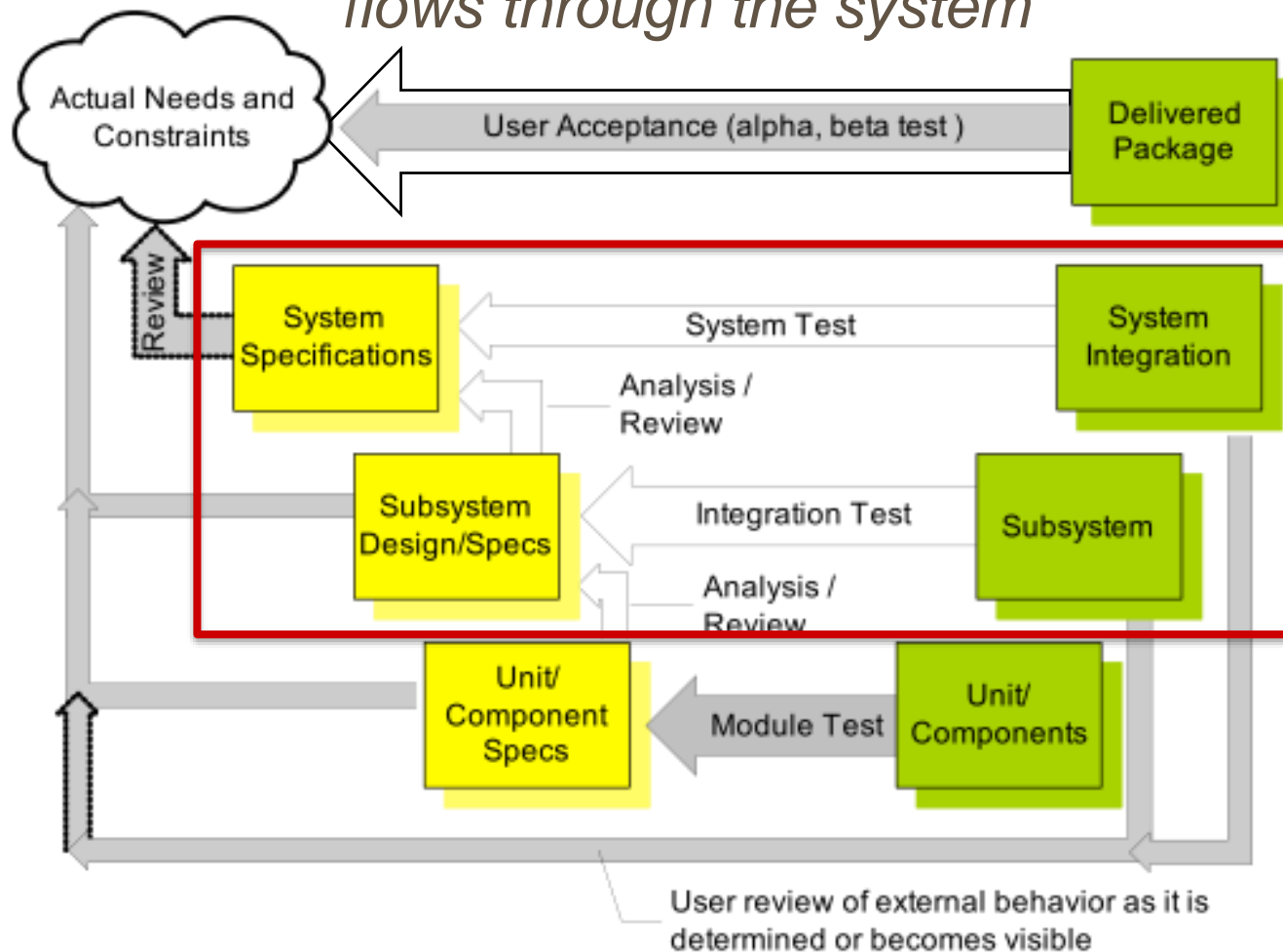
- Paired Programming
- Formal reviews

Quality code as a culture

- Especially when time is tight

INTEGRATION AND SYSTEM TESTING

Emphasizes interactions between components/systems and flows through the system



A red rectangular warning sign with rounded corners and a white border. It is held in place by two silver screws at the top and bottom center. The word "WARNING!" is written in large, bold, white, sans-serif capital letters at the top. Below it, a message is written in a smaller, white, italicized, sans-serif font.

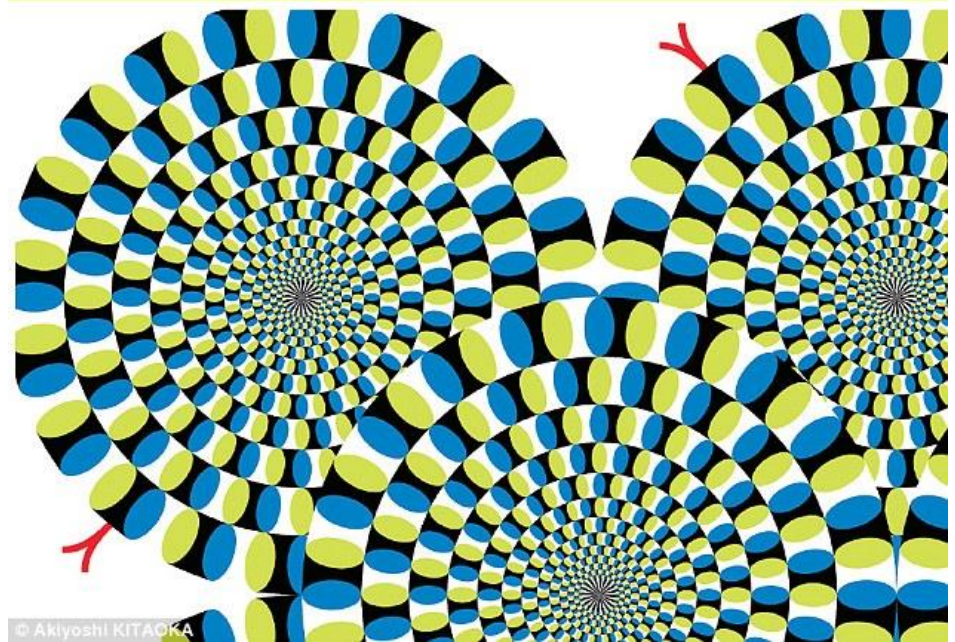
WARNING!

*Will be a complete and
utter waste if
components are not
thoroughly tested*

ONE DIFFERENCE FROM UNIT TESTING: EMERGENT BEHAVIOR

Sometimes, behavior is only clear when you put components together

This has to be tested too, although it can be very hard to plan in advance!



*Usually this is identified after the fact,
causing test suites/cases to be refactored.*

A SECOND (NON-TECHNICAL) DIFFERENCE: INTEGRATION TESTING IS A TEAM SPORT

Integration testing is often more complicated than merely putting the pieces together and then evaluating them in an orderly manner

While this should be the case,
this is a group activity.

BUT not everyone is always engaged or even
knows integration is going on



ERRORS MAY CAUSE THE BLAME GAME



IT CAN BE LIKE HERDING CATS

How to maintain momentum when *not everyone is at the table, partners don't share your priorities, or no-one "owns" the code?*

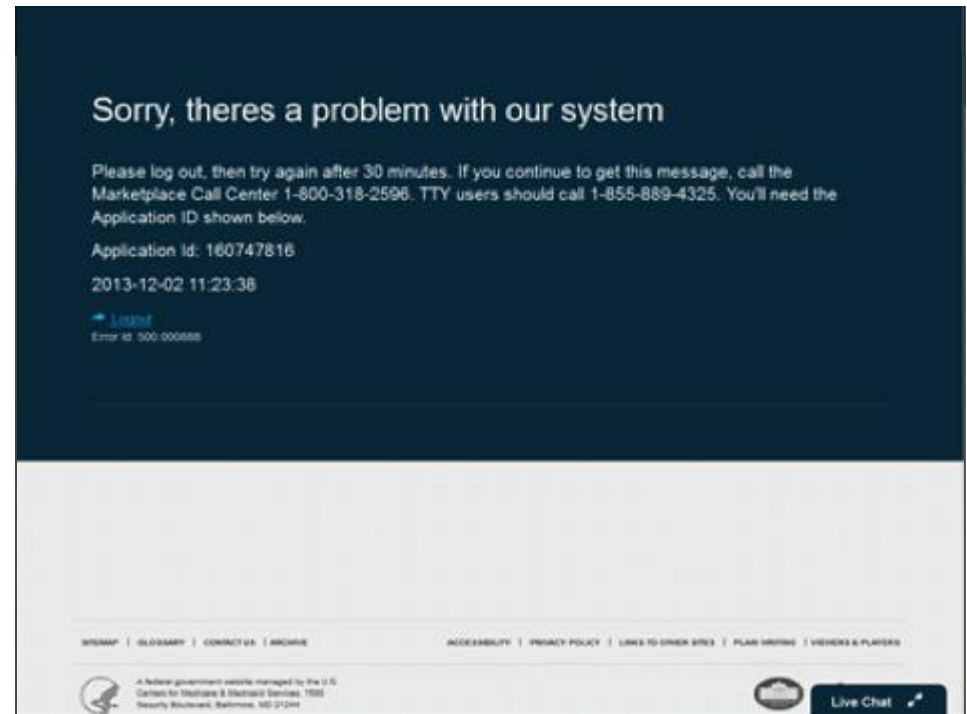


PERFORMANCE TESTING

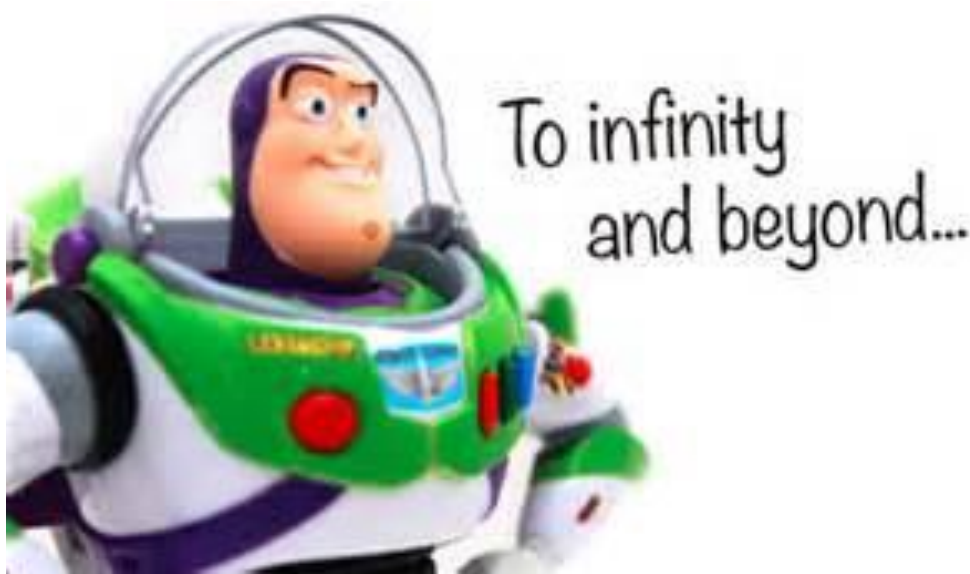
Measures the system's capacity to process a specific load over a specific time-span, usually:

1. *number of concurrent users*
2. *specific number of concurrent transactions*

Involves defining and running operational profiles that reflect the expected use



FOUR MAJOR TYPES OF PERFORMANCE TESTING



1. Load

Aims to assess compliance with non-functional requirements

2. Stress

Identifies system capacity limits

3. Spike

Testing involving rapid swings in load

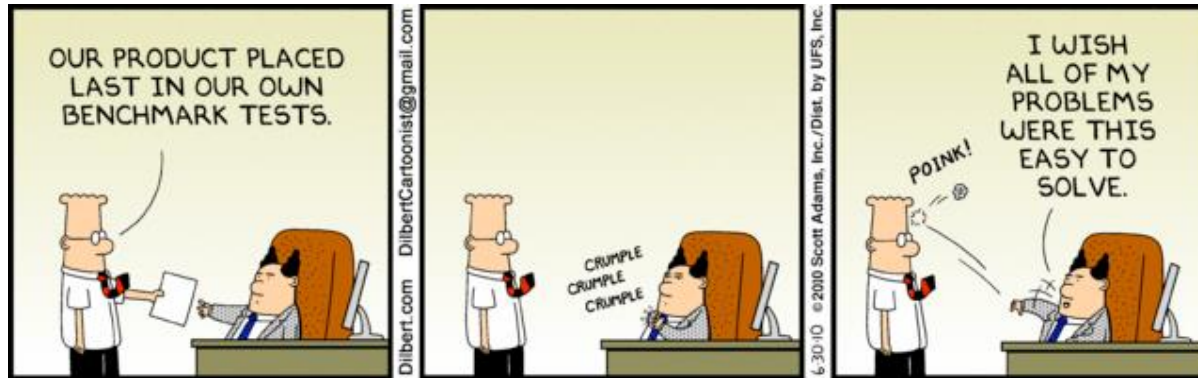
4. Endurance (or Soak)

Continuous operation at a given load

Ideally the system should degrade gracefully rather than collapse under load

Under load, issues like protocol overhead or timing issues take center stage

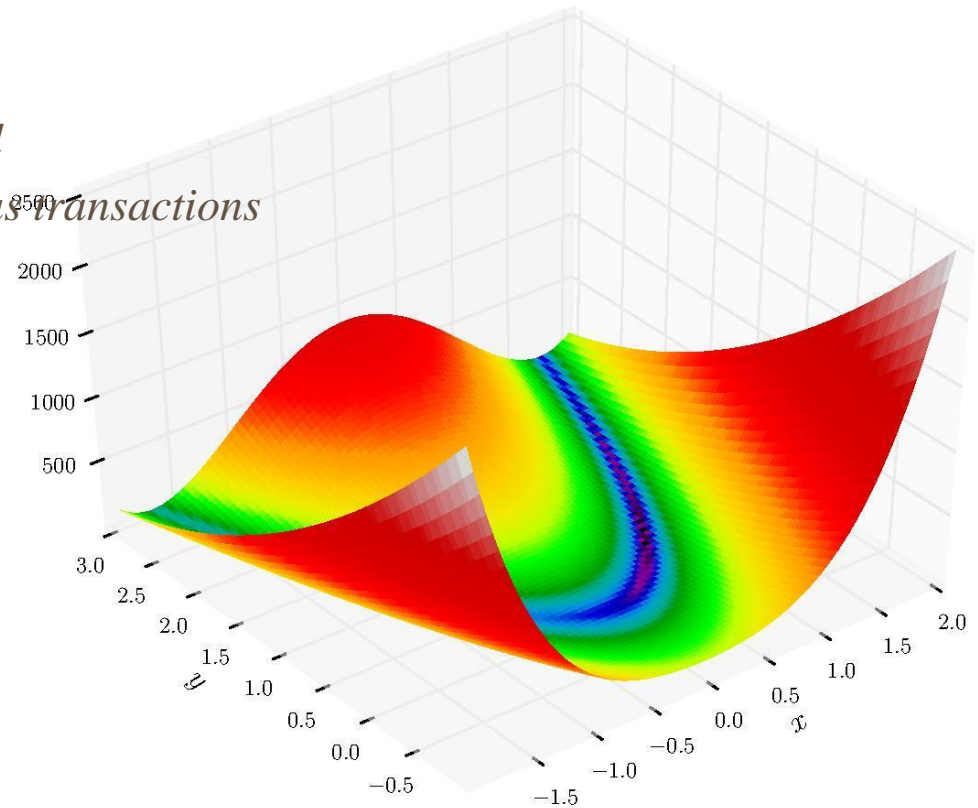
WHY DO PERFORMANCE TESTING?



1. The requirements demand it!
There are non-functional requirements specifying performance
2. It can compare two systems to find which performs better
3. May identify which parts of the system are the “weak links”
4. It can identify workloads that cause the system to perform badly

WHAT TO OPTIMIZE?

- For Throughput or Concurrency?
 - *Getting the most data processed*
 - *Greatest number of simultaneous transactions*
- For Server response time?
- For Service request round-trip time?
- For Server utilization?
- For End-User Experience?
- For Cost?



SECURITY TESTING – ASSESSING IF THE SYSTEM IS ADEQUATELY PROTECTED

It's more than penetration testing (PEN tests). This assesses:

1. Confidentiality

Information protection from unauthorized access or disclosure

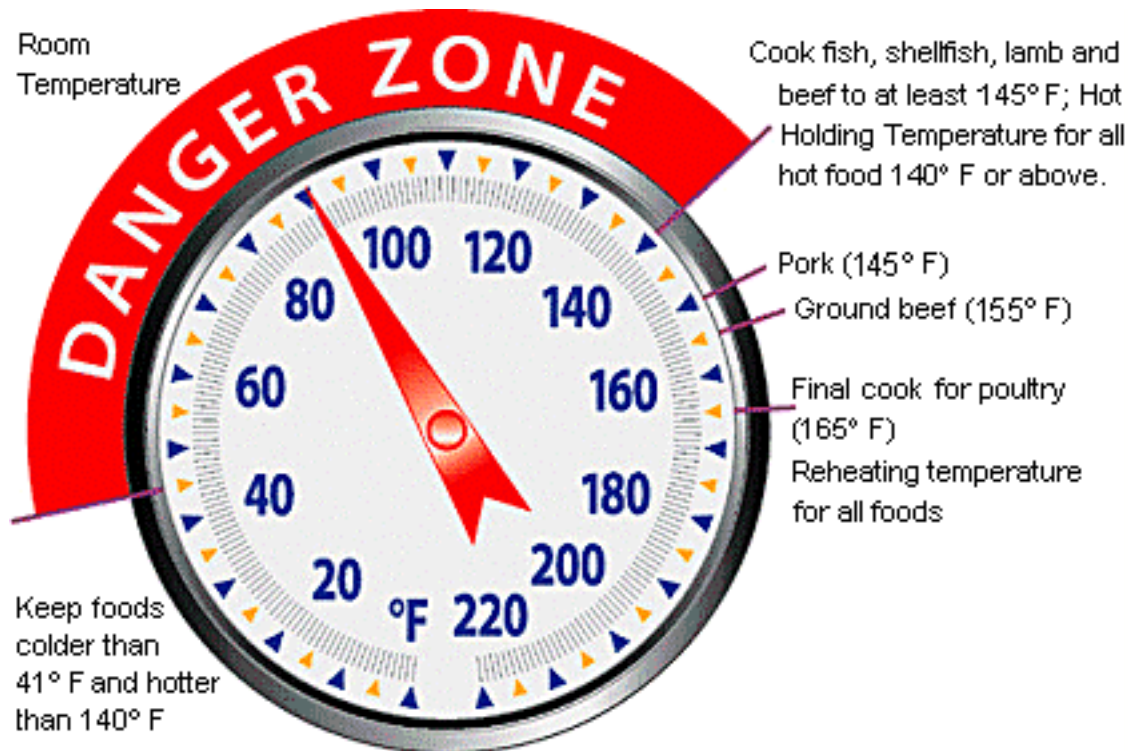
2. Integrity

Information protection from unauthorized modification or destruction

3. Availability

System protection from unauthorized disruption

HOW DO WE KNOW WHEN A PRODUCT IS READY?



FIRST APPROACH: THROW IT AGAINST THE WALL AND SEE IF IT STICKS



Let the customer test it and if
they don't complain or return
it, it's done!

Or

If we get a lot of *likes*, it's
done



SECOND APPROACH: KEEP GOING UNTIL YOU RUN OUT OF TIME OR MONEY

We're running out of
time!

We're running out of
money!

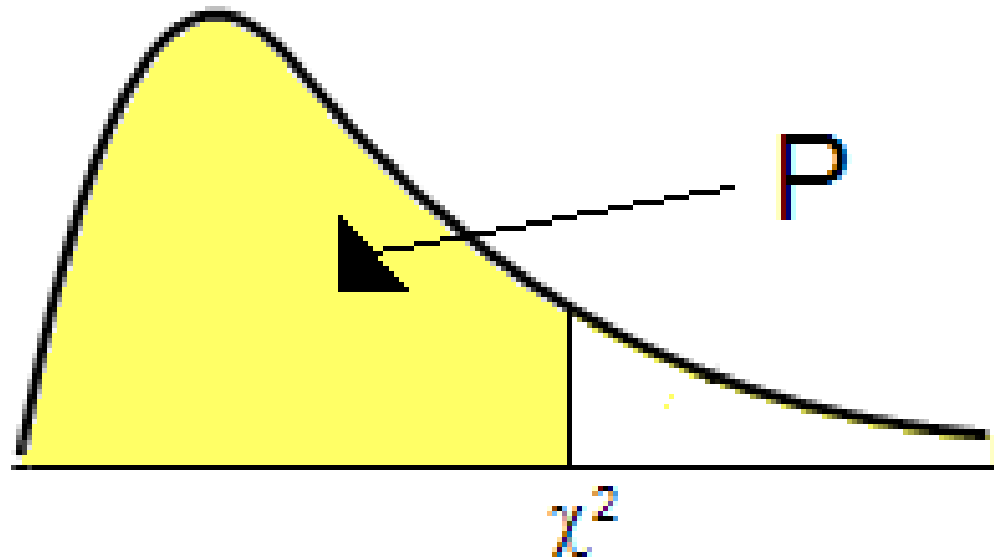
Ship it!



THE IDEAL



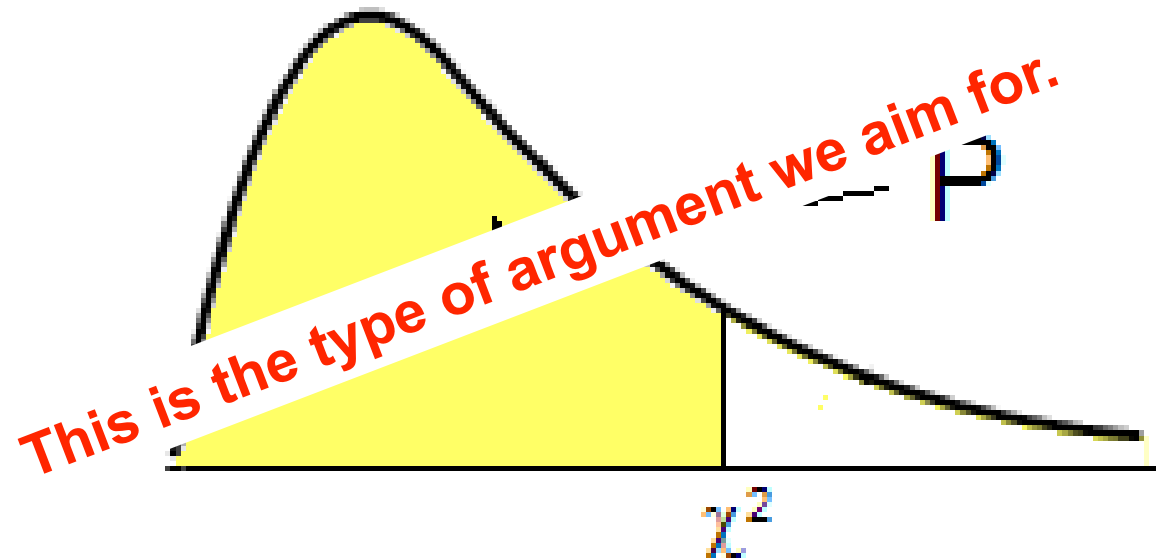
Relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95% confidence that the probability of 1,000 CPU hours of failure-free operation is ≥ 0.995 .



THE IDEAL



Relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95% confidence that the probability of 1,000 CPU hours of failure-free operation is ≥ 0.995 .



LOTS OF REGRESSION TESTING IS KEY

Reasserting that changes to system haven't broken previously working parts

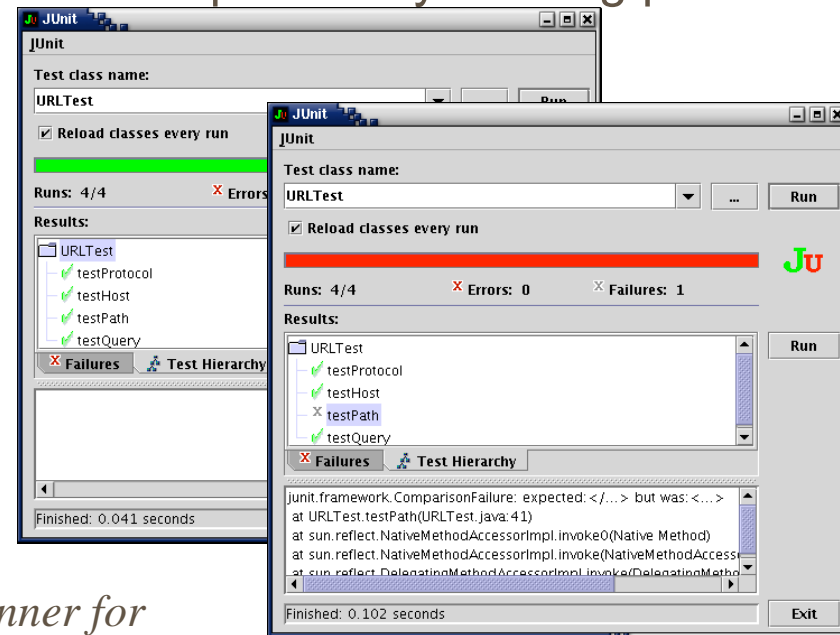
1. Retest everything

2. Retest using only selected tests

- *Tester specifies what test cases/suites get run*

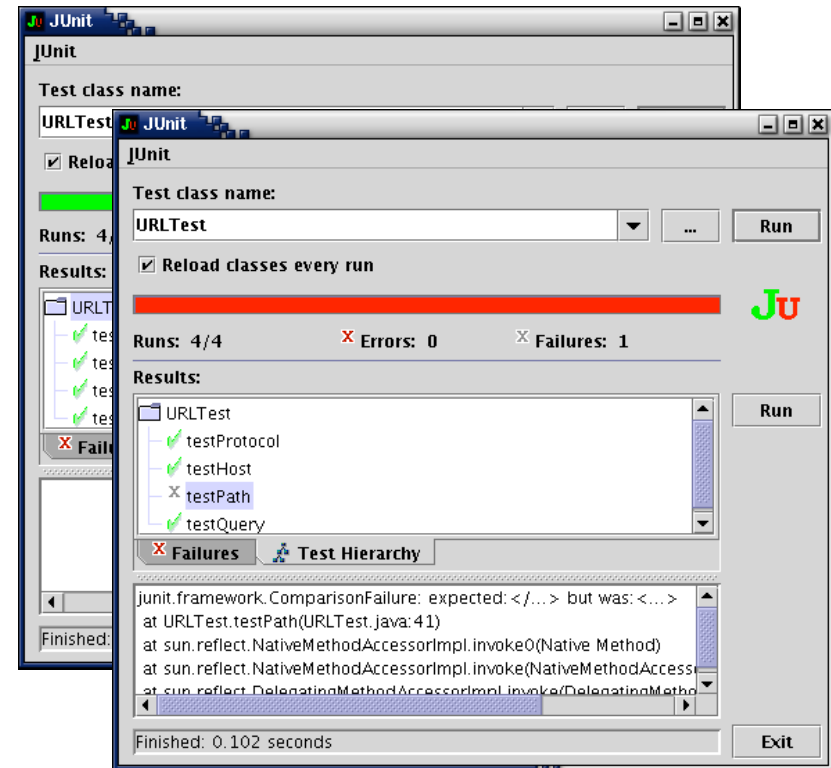
3. Retest using priority orders

- *Tester defines priorities against test cases/suites. Tests are run in a greedy manner for some time period.*
- *Priorities may be set up against specific system or module versions.*



REGRESSION TESTS: ASSERTING CHANGES HAVEN'T BROKEN WHAT WAS WORKING

1. Set up automated tests
 - using, e.g., JUnit
2. Ideally, run regression tests after each change
3. If running the tests takes too long:
 - a. *prioritize and run a subset*
 - b. *apply regression test selection to determine tests that are impacted by a set of changes*



AN OBVIOUS ISSUE WITH REGRESSION TESTING

Becomes very expensive as test suites grow and code change velocities are high.

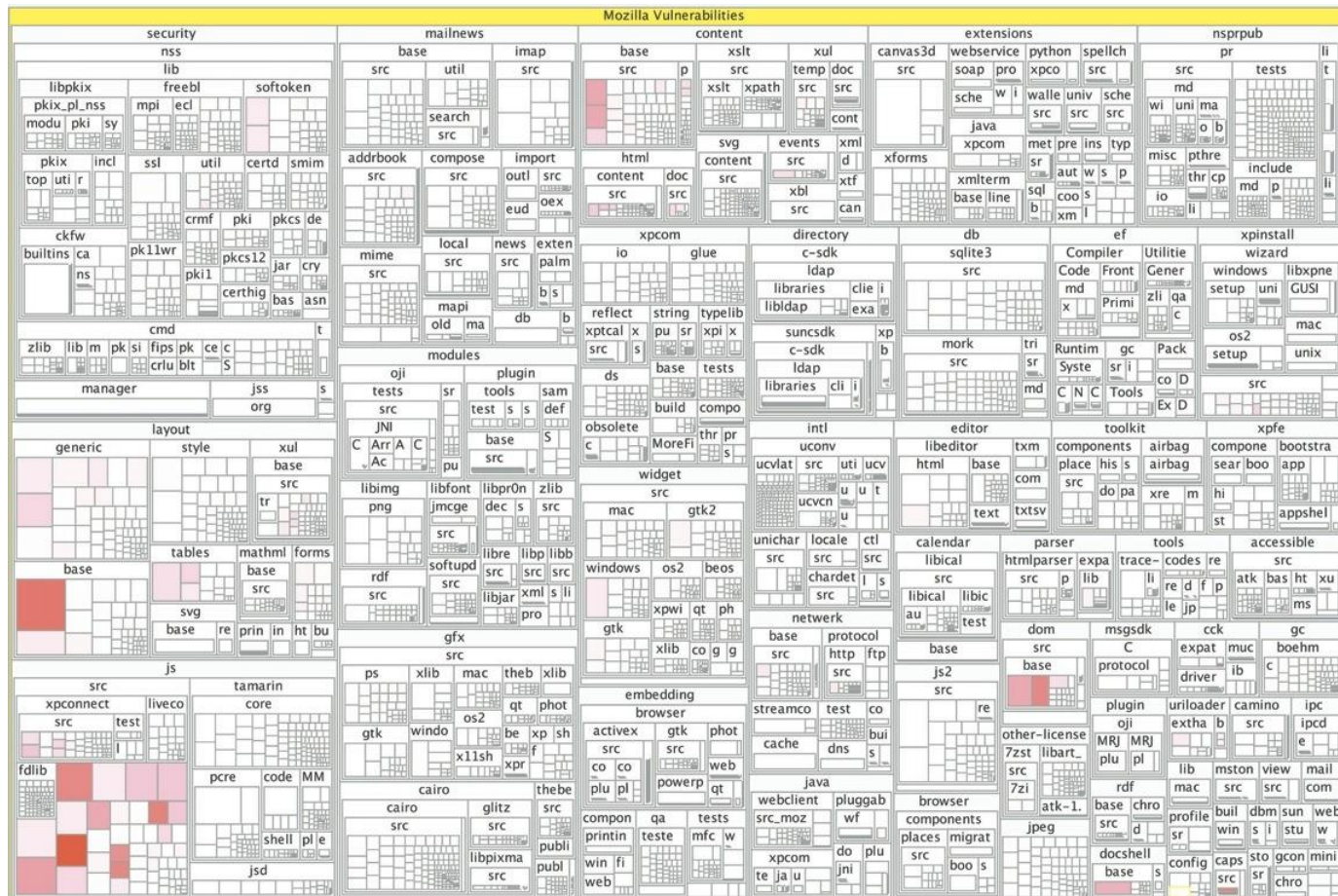
It may be that regression test cycles keep getting disrupted by continual code changes

Is a great way to put idle machine cycles to work
especially if you own the machines (versus renting them)

“Also as a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire batch of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice, such *regression testing* must indeed approximate this theoretical idea, and it is very costly.”

- F. Brooks, *The Mythical Man Month*

HOW TO CONTROL QUALITY OF SUCCESSIVE RELEASES AND HOW TO IMPROVE DEVELOPMENT?



REMEMBER PARETO'S LAW

**Approximately 80% of defects
come from 20% of modules**

BEST PRACTICES

1. Specify requirements in a quantifiable manner
2. State testing objectives explicitly
3. Understand the users of the software and develop a profile for each user category
4. Develop a testing plan that emphasizes “rapid cycle testing”

BEST PRACTICES

5. Build “robust” software that is designed to test itself
 - *Remember to build in the “little red/green light”*
6. Use effective formal technical reviews as a filter prior to testing
7. Conduct formal technical reviews to assess the test strategy and test cases themselves
8. Develop a continuous improvement approach for the testing process

DESIGN FOR TESTING

1. OO design principles also improve testing

Encapsulation leads to good unit tests

2. Provide diagnostic methods

Primarily used for debugging, but may also be useful as regular methods

3. Assertions are great helpers for testing

Test cases may be derived automatically

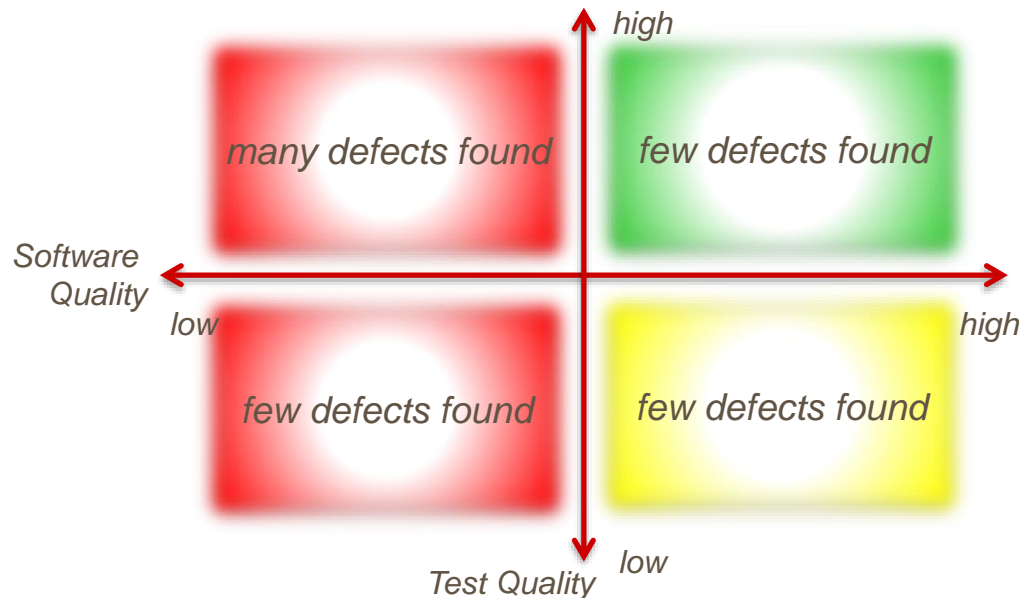
HOW DO YOU KNOW IF YOUR TESTING PROGRAM IS ANY GOOD?



YOUR TESTING IS GOOD ENOUGH UNTIL A PROBLEM SHOWS THAT IT IS NOT GOOD ENOUGH

It is hard to know when you should feel enough confidence to release the system

Confidence comes on the sub-test of possible tests selected and perception of the team



AN APPROACH: MUTATION TESTING

Introduce small changes to the code and see if testing catches it



INTENT IS TO IDENTIFY REGIONS OF CODE THAT WEREN'T TESTED AND TO CATCH “DUMB MISTYPING” BUGS

Statement Mutation – changes to lines of code (adds, removes, changes)

Value Mutation- changes parameter values are modified

Decision Mutation- changes to control statements

Original		Mutant
<code>c = a*b;</code>	\rightarrow	<code>c = a+b;</code>
<code>If (a == 10) {</code>	\rightarrow	<code>If (a == 11) {</code>
<code>If (a == b) {</code>	\rightarrow	<code>If (a != b) {</code>

WHEN DO TEST SUITES NEED REFACTORING?

Generate a set of mutants (M_T)

If testing catches the mutant, it is considered *killed* (M_K)

If testing does not catch the mutant, it is considered an *mutant equivalent* (M_E)

- *Different code, same accepted result*

Measuring effectiveness (Mutation Score) $MS = \frac{M_K}{M_T - M_E}$

- *Anything less than $MS = 1$ is bad
and means test suites need refactoring to cover M_E*
- *M_E approaching M_T is bad*

PUTS AND TAKES ON MUTATION TESTING

Advantages

- + Can nicely cover the original source
- + Finds ambiguities in the source code
- + May detect all the faults in the testing regime

Originally proposed in the early 2000's

Back in vogue because of the explosion in compute capacity

Disadvantages

- Mutation testing is extremely costly and time consuming to pursue
 - Requires generating many mutant programs
 - Each mutation runs the original test suite(s), which may involve many test cases or run tests that take a long time – or both
 - *Creates a potentially huge number of test suites to run.*
- Requires additional tech to manage mutation generation and detection

ANOTHER APPROACH: DEFECT DENSITY

- Using the past to estimate the future
- Judges code stability by comparing past number of bugs per code measure to present measured levels

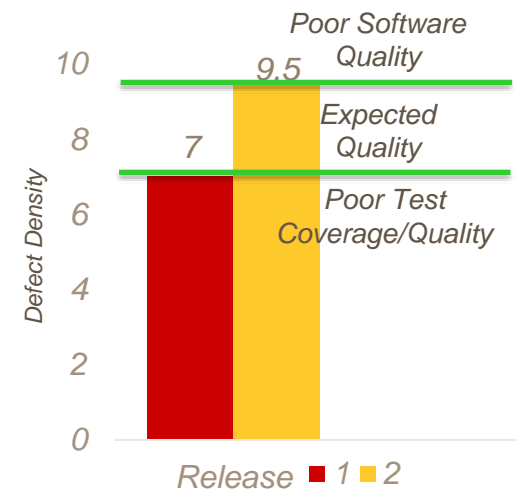
$$BugDensity_{release(i)} = \frac{BugsFound(prerelease_i) + BugsFound(postrelease_i)}{CodeMeasure(release_i)}$$

Release_i is a candidate for release if

$\min(BugDensity_{release(1..i-1)}) \leq BugDensity_{release(i)} \leq \max(BugDensity_{release(1..i-1)})$ holds

If density for the next release's additional code is within ranges of prior releases, it is a candidate for release

provided test or development practices haven't improved



*In other words, this version is as bad as previous versions,
but it's ok because we released those*

MANY, MANY WAYS TO MEASURE CODE

Approach 1: Count lines of Code

1. Lines of code
2. Count only executable lines.
3. Count executable lines plus data definitions.
4. Count executable lines, data definitions, and comments.
5. Count executable lines, data definitions, comments
6. Count lines as physical lines on an input screen.
7. Count lines as terminated by logical delimiters.
8. Count only shipped lines
9. Count only new/changed lines

Approach 2: Count function points

1. Using a standard notion of a function/object/component, use a weighted function over
 1. Number of external interfaces
 2. Number of inputs
 3. Number of outputs
 4. Internal complexity
 5. Configuration complexity
- The weights may consider:
 1. Data communications
 2. Performance
 3. Transaction rate
 4. Reusability
 5. Installation ease
 6. Operational ease

A THIRD APPROACH: CAPTURE-RECAPTURE

Uses a technique for predicting wild-life populations to estimate the number of defects

Example: Estimating Turtle Population (assuming turtles don't migrate)

You catch and tag 5 turtles. Then, you release them.

You later catch 10 turtles, and two have tags.

$$\frac{\text{Total \# of turtles}}{5 \text{ turtles}} \approx \frac{10 \text{ turtles}}{2 \text{ turtles}}$$

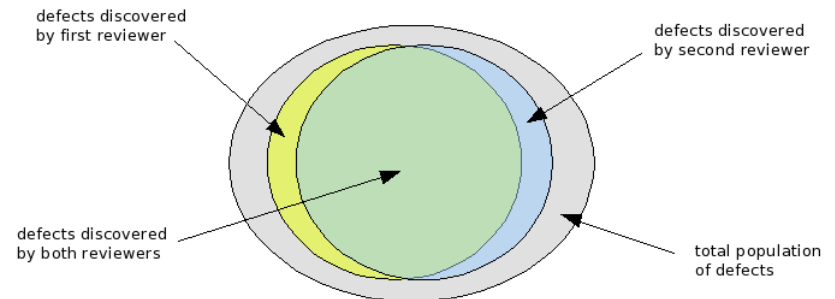
$$\text{Total \# of turtles} = \frac{10 \text{ turtles} * 5 \text{ turtles}}{2 \text{ turtles}} \cong 25$$



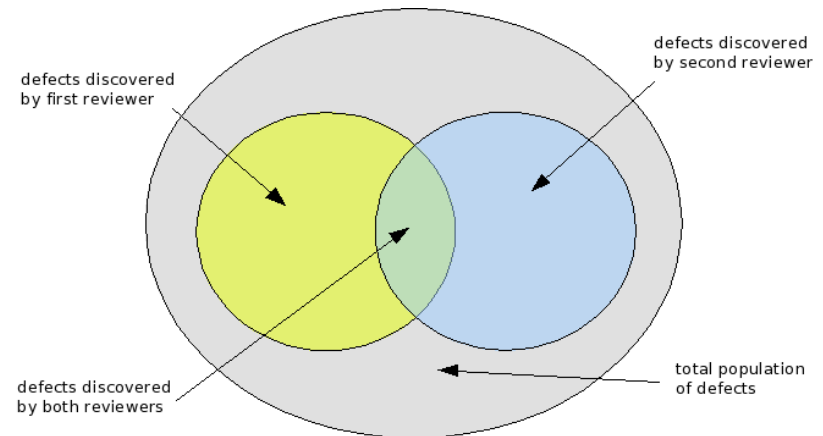
- Uses data collected by two or more independent collectors
- Collected via reviews or tests

CAPTURE-RECAPTURE

- Each collector finds some defects out of the total number of defects
- Some of these defects found will overlap
- Method
 1. Count the number of defects found by each collector (n_1, n_2)
 2. Count the number of intersecting defects found by each collector (m)
 3. Calculate defects found = $(n_1 + n_2) - m$
 4. Estimate total defects = $\frac{(n_1 * n_2)}{m}$
 5. Estimate remaining defects
$$remainder = \frac{(n_1 * n_2)}{m} - (n_1 + n_2) - m$$
- If multiple collectors, assign A to the highest collected number and set B to the rest of the collected defects. When multiple engineers find the same defect, count it just once.



When most findings overlap



When there is little overlap