Software testing and Quality Engineering

Lecture 2: *Foundations*

CSE1110

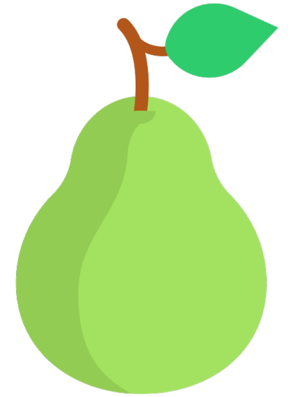
Maurício Aniche, Arie van Deursen
Delft University of Technology

Kazimir Malevich, Eidght Red Rectangles, Wikipedia

Labwork Announcements

- What should be in the report?
  - Answers to all the questions!
- Today's (Friday April 26) labwork session?
  - *You are all welcome to join today, even if it is not your group*

Labwork Announcements: Peer Review



- Peer review
  - Grade your self, given rubrics
  - Grade a randomly assigned (anonymous) students
- Make sure your code (comment) does not reveal your identity
  - Be careful in your comments.
  - Double check automatically generated javadocs...
- Upload zipped (src + pdf) file into the “peer” system.
- Conducting peer review is *mandatory*
- Timing is tight: *All deadlines are strict*
  - Please don’t fail your BSA by missing a deadline

Exemptions *only* via study counselor

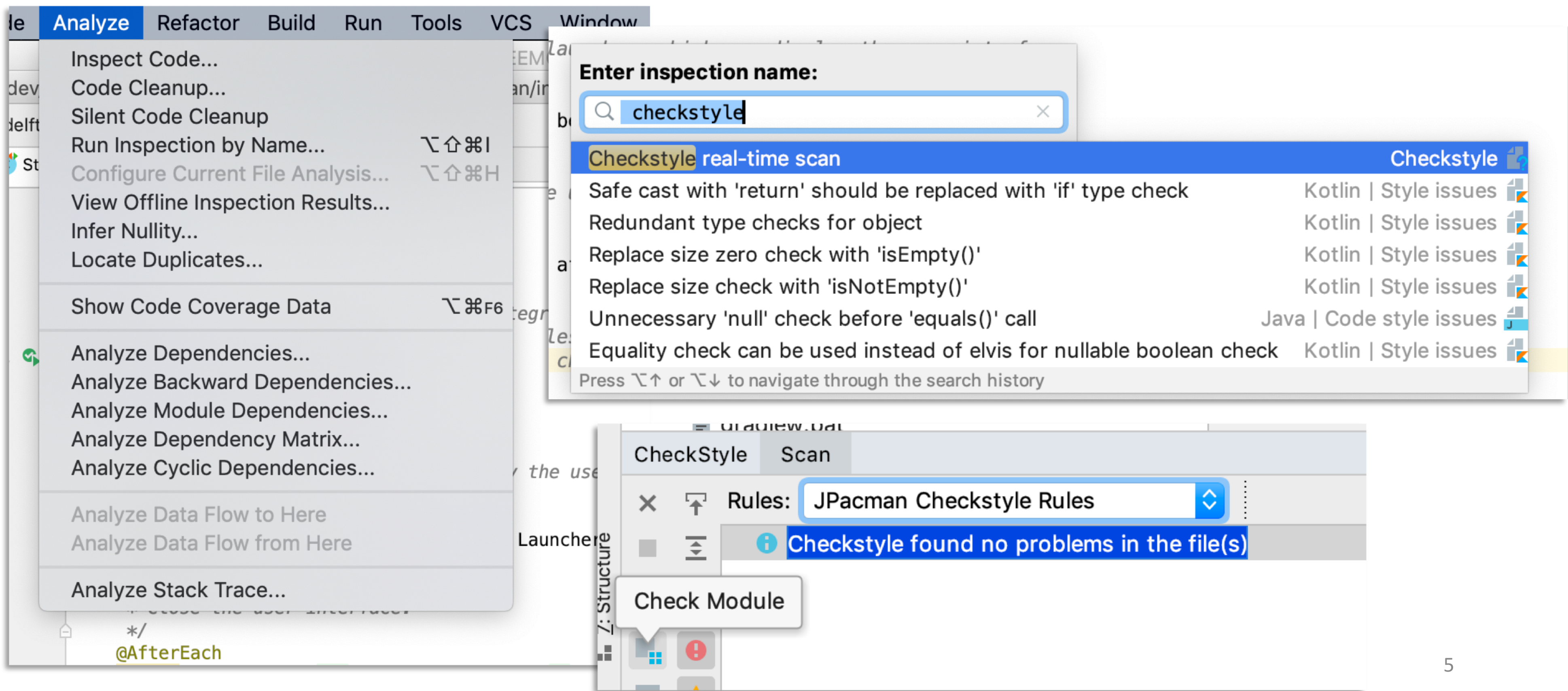
Labwork: Group Creation Deadline

- Today, 1pm – FIRM DEADLINE

- Be in a single group
- With exactly two members

- You can already start from github/serg-delft/jpacman

Running Checkstyle in IntelliJ?



Might Fail when Run with "Check Project"

CheckStyle | Scan

Rules: JPacman Checkstyle

The scan failed due to an error - please see the event log for more information

Check Project

Event Log

```
public class AmazingPointCalculator implements nl.tudelft.jp
    private int pelettesEaten;
    private static java.util.List<nl.tudelft.jpacman.board.Direct
    private static java.util.List<nl.tudelft.jpacman.board.Direct

    public AmazingPointCalculator() { /* compiled code */ }

    public void collidedWithAGhost(nl.tudelft.jpacman.level.Pl

    public void consumedAPellet(nl.tudelft.jpacman.level.Playe
```

Reformat Code
Optimize Imports
Delete...
Build Module 'jpacman_main'
Rebuild 'scoreplugins'
Reveal in Finder
Open in Terminal
Pitest all tests in module
Pitest classes in 'scoreplugins'
Local History
Git
Synchronize 'main/.../scoreplugins'
Compare With...
Mark Directory as → Excluded / Load Path Root
Remove BOM
ASM Bytecode Viewer

Run "check module instead"
Exclude resources folder from checkstyle.

SERG-Delft / jpacman

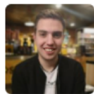
Unwatch

Code | Issues 1 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

Suppress spotbugs false positive in Java 11 #27

Merged **casperboone** merged 2 commits into `master` from `java-11-false-positive` a day ago

Conversation 1 | Commits 2 | Checks 2 | Files changed 3



casperboone commented a day ago

In Java 11 SpotBugs reports an issue about a redundant nullcheck for try with resources.

Unfortunately, this is a common issue with no fix available:

- spotbugs/spotbugs#259
- spotbugs/spotbugs#756

This PR suppresses the warning.

I've also added JDK 11 to the Travis config (we would have been able to see the error if we/I had done this earlier 😞) .

Multiple Choice:
People *love* git because git

A. supports understanding a change in its historical context

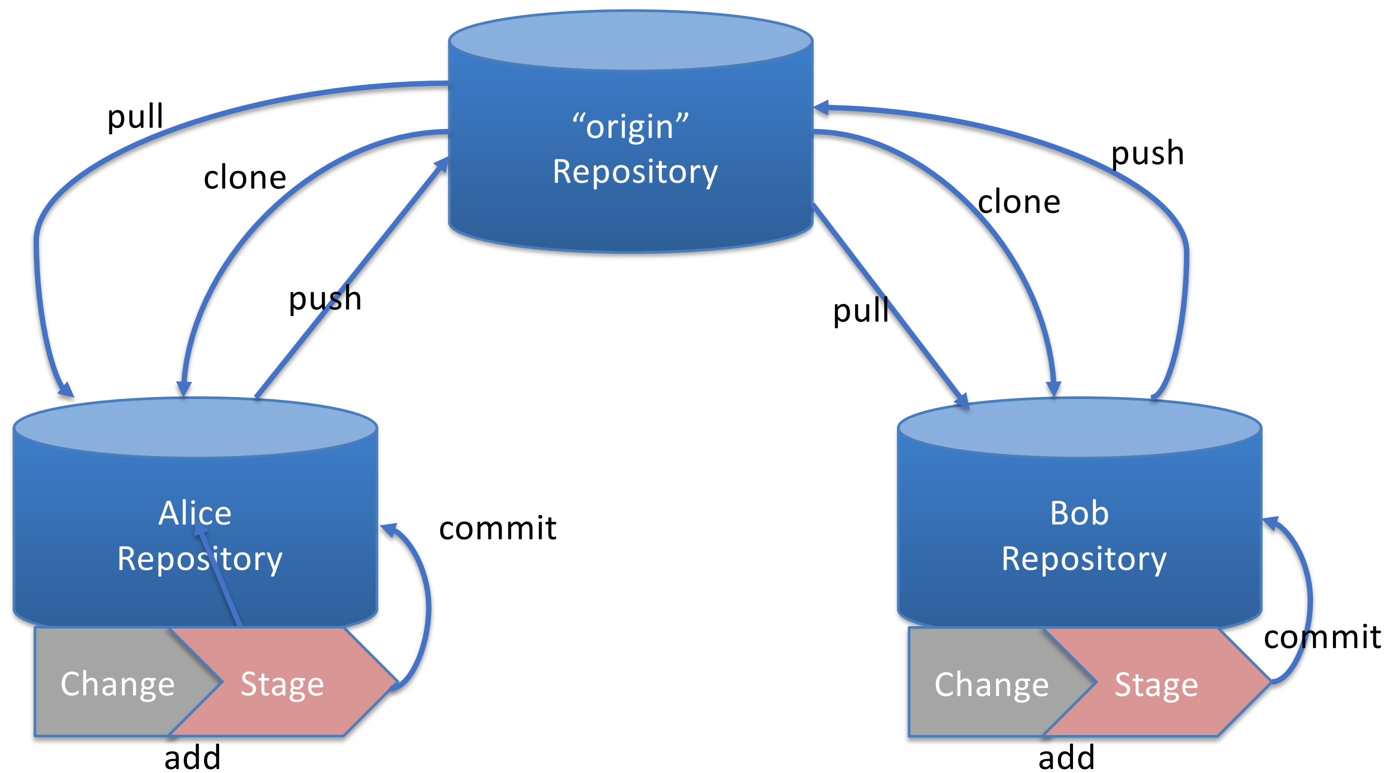
B. supports isolating changes and moving them around

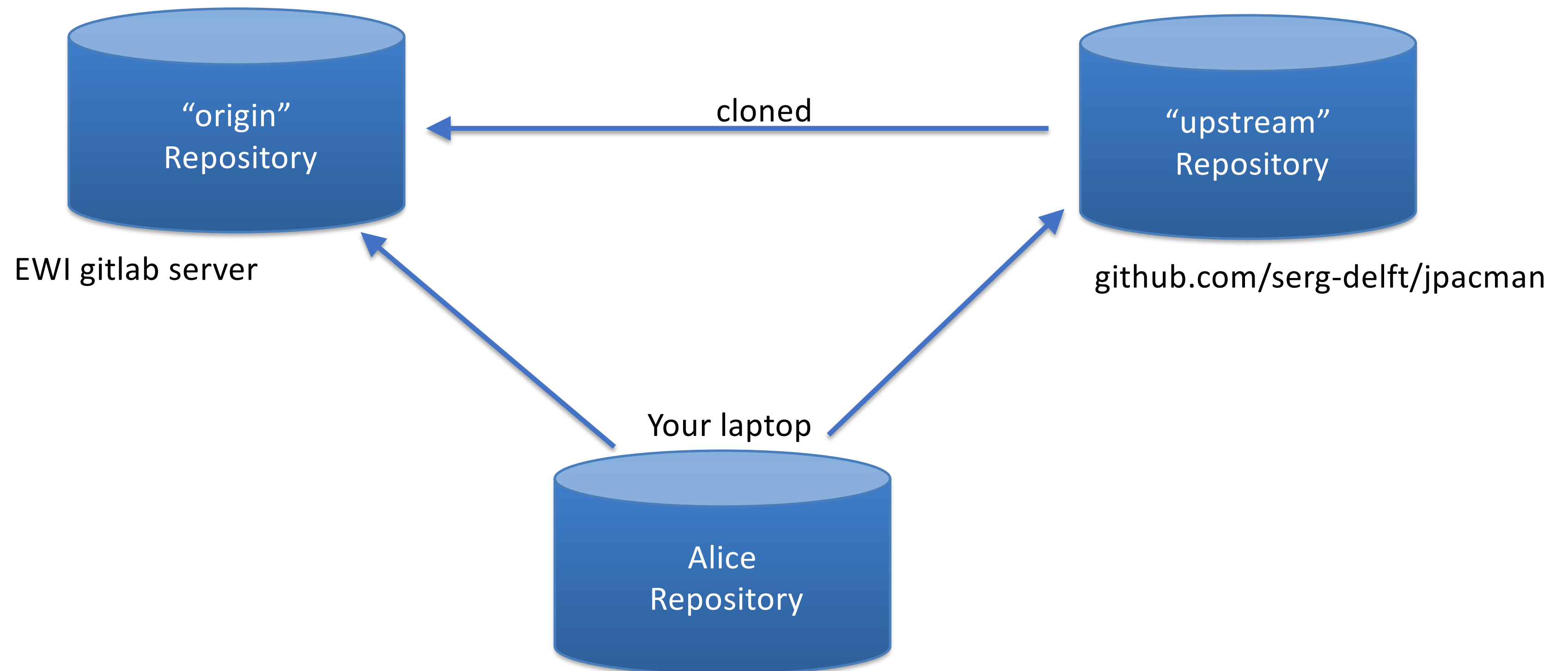
C. supports identifying and discussing changes

D. scales to 1000s of distributed developers

E. All of the above

Sharing Changes With the "Origin" Remote






```
git remote add upstream https://github.com/SERG-Delft/jpacman.git

git fetch upstream

git merge upstream/master -m "Merge remote-tracking branch 'upstream/master'"
```

Somewhat safer

git status

if there is any change commit first

if the current branch isn't master checkout master

gradle test

always make sure you run your test before any change

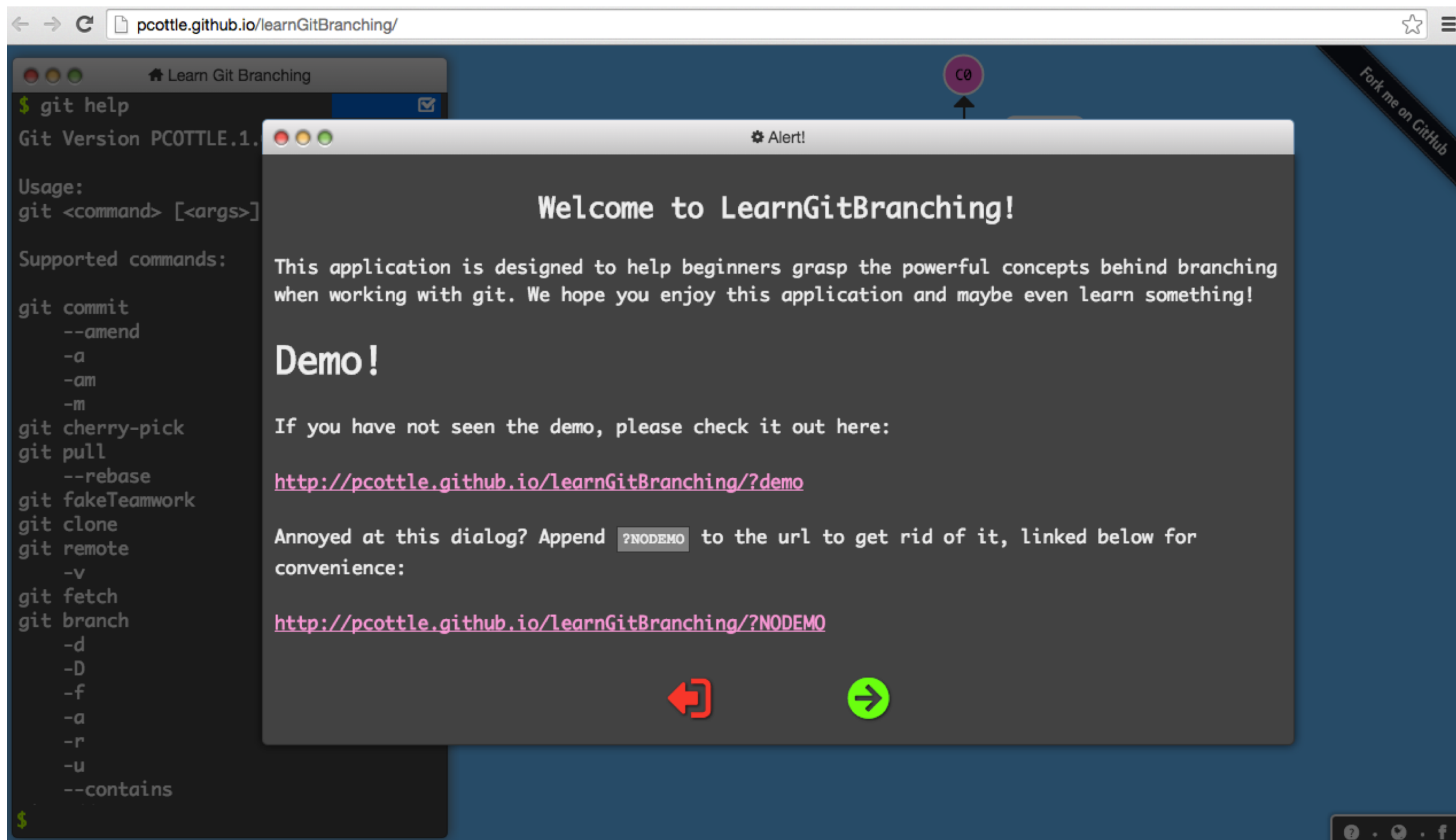
now your all set to merge in more changes

git remote add ...

git fetch ...

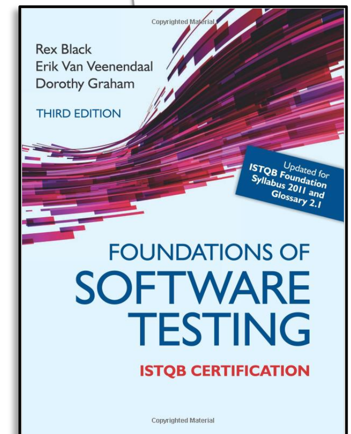
git merge ...

https://learngitbranching.js.org/



Reliable Knowledge in Software Testing?

- Software testing is all about making ***trade-offs***
  - Becomes easier with experience!
- Strategies, patterns, and processes are ***codified experience***
  - You will need to know them!
- Our body of knowledge grows as reflective engineers / researchers:
  - Codify their knowledge and pass it on
  - Analyze successes and failures and report on those
  - Propose, implement, and evaluate novel testing strategies



1.1.2

Terminology: **Failure**

"Deviation of the component or system from its expected delivery, service or result"

"Manifested inability of a system to perform required function"

Windows

A fatal exception 0E has occurred at 0028:C0011E36 in VXD VMM(01) +
00010E36. The current application will be terminated.

* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will
  lose any unsaved information in all applications.

Press any key to continue

Windows

A fatal exception 0E has occurred at 0028:C0011E36 in VXD VMM(01) +
00010E36. The current application will be terminated.

* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will
  lose any unsaved information in all applications.

Press any key to continue

Windows *failure* leads to “blue screen of death”.

1.1.2

Terminology: **Defect / Fault**

*“Flaw in component or system
that can cause the component or system
to fail to perform its required function”*

*“A defect, if encountered during execution,
may cause a failure of the component or system”*

Synonym: **Fault**

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer
                                 uint8_t *signature, UInt16 signatureLen
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

http://avandeursen.com/2014/02/22/gotofail-security/

```
1  static OSStatus
2  SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer
3                                   uint8_t *signature, UInt16 signatureLen
4  {
5      OSStatus        err;
6      ...
7
8      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
9          goto fail;
10     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
11         goto fail;
12         goto fail;
13     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
14         goto fail;
15     ...
16
17 fail:
18     SSLFreeBuffer(&signedHashes);
19     SSLFreeBuffer(&hashCtx);
20     return err;
21 }
```

Fault in Apple's Secure Socket Layer code

1.1.2

Terminology: **Error**

"A human action
that produces an incorrect result"

Synonym: **Mistake**

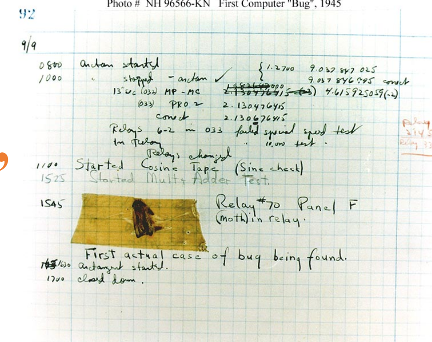
1.1.2

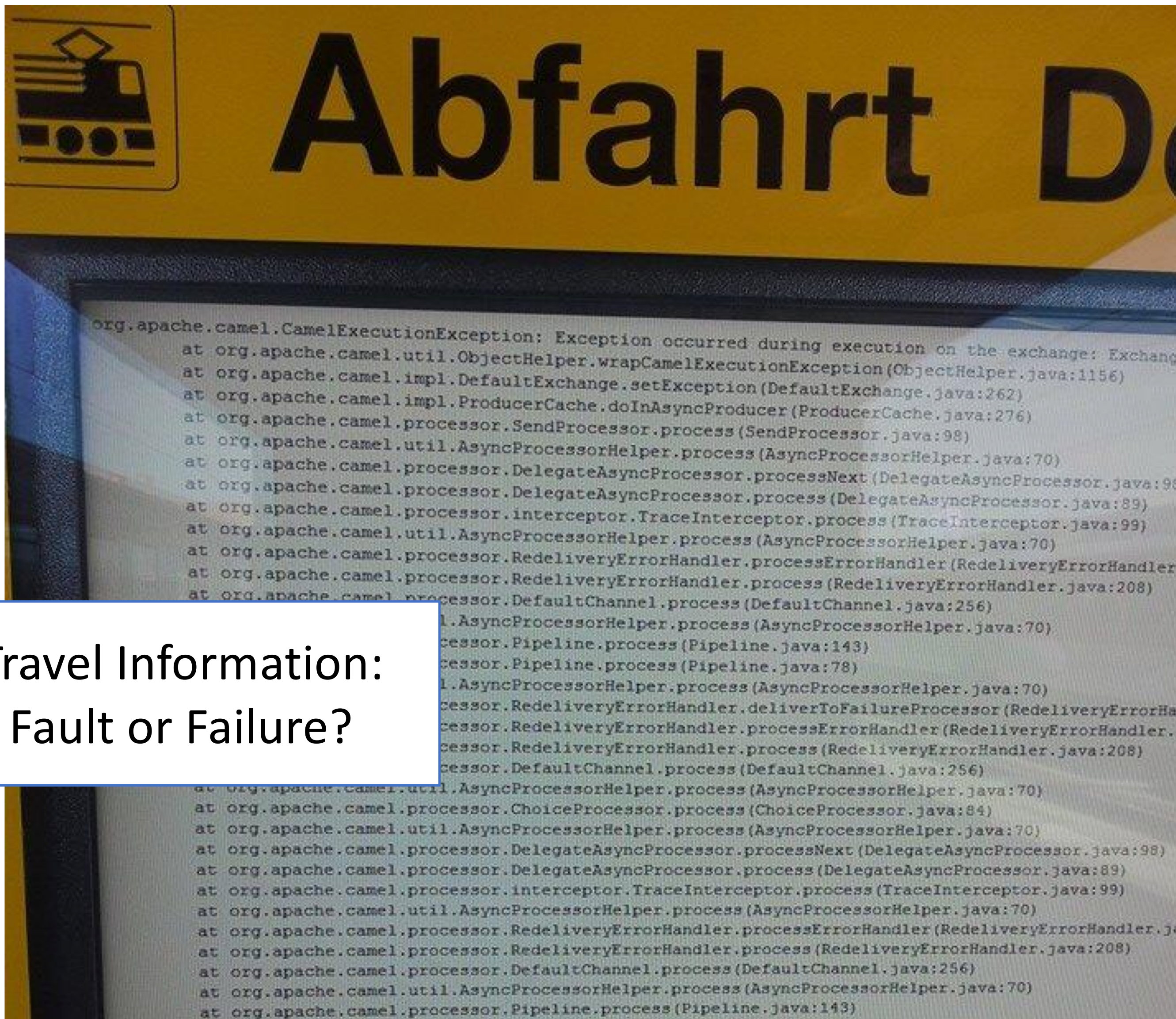
Faults, Failures, and Bugs

- **Failure:**
  - Manifested inability of a system to perform required function.
- **Defect (fault):**
  - missing / incorrect code
- **Error** (mistake)
  - human action producing fault
- **And thus**:
  - Testing: Attempt to trigger failures
  - Debugging: Attempt to find faults given a failure

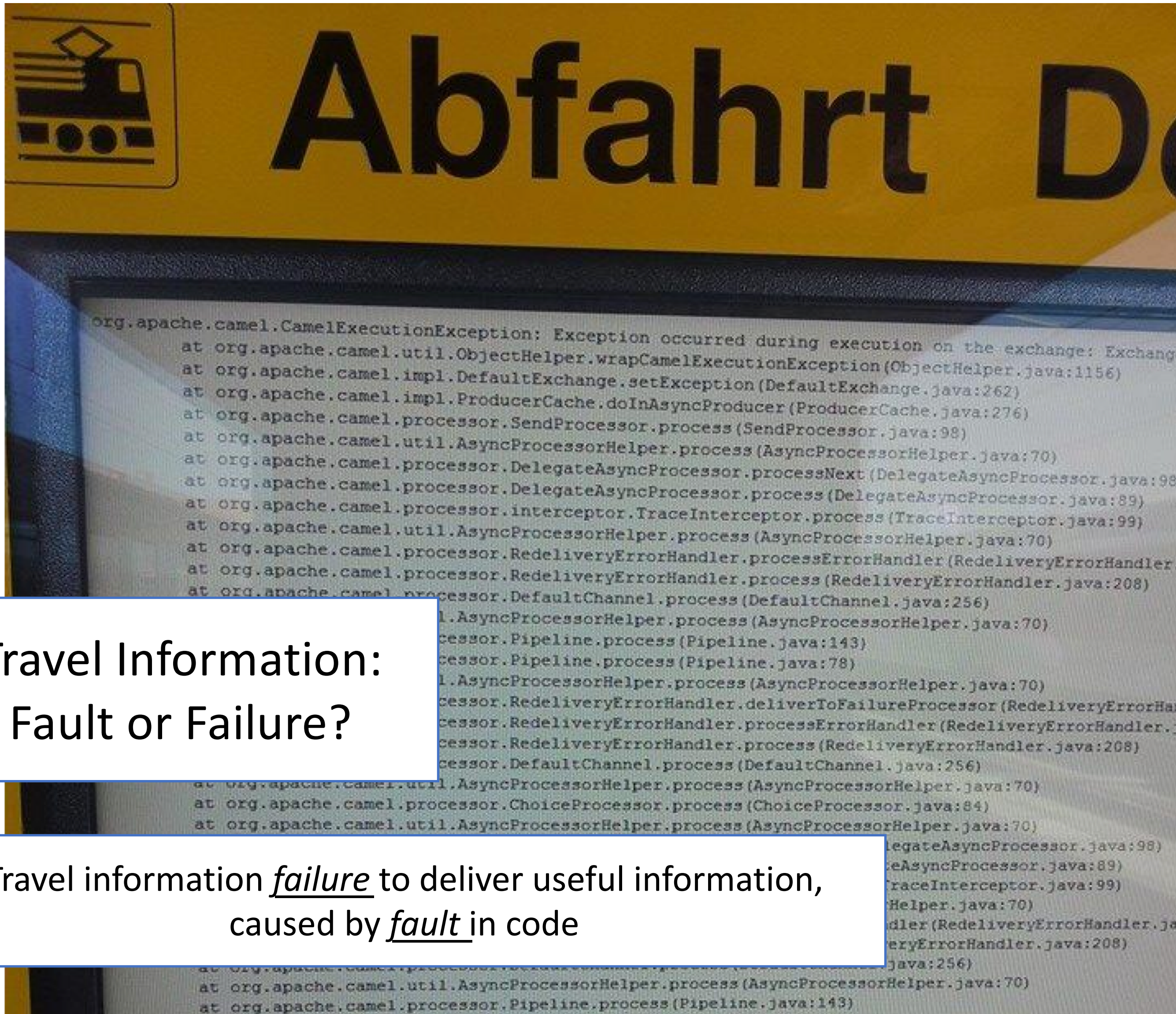
“bug”

Photo # NH 96566-KN First Computer "Bug", 1945





Travel Information:
Fault or Failure?



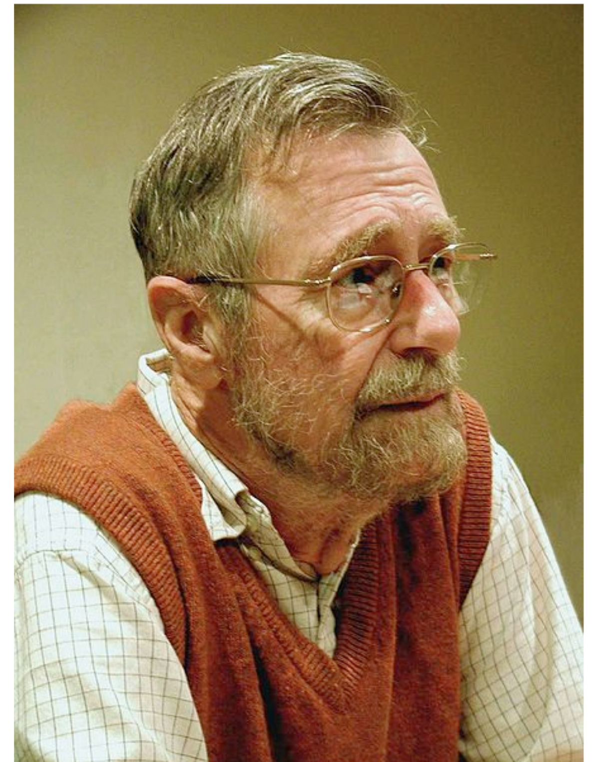
Travel Information:
Fault or Failure?

Travel information *failure* to deliver useful information, caused by *fault* in code

1.3.1

Principles of Testing #1

- Testing shows the *presence* of defects
- Testing does *not* show the *absence of defects!*
- *"no test team can achieve 100% defect detection effectiveness" (Black et al)*



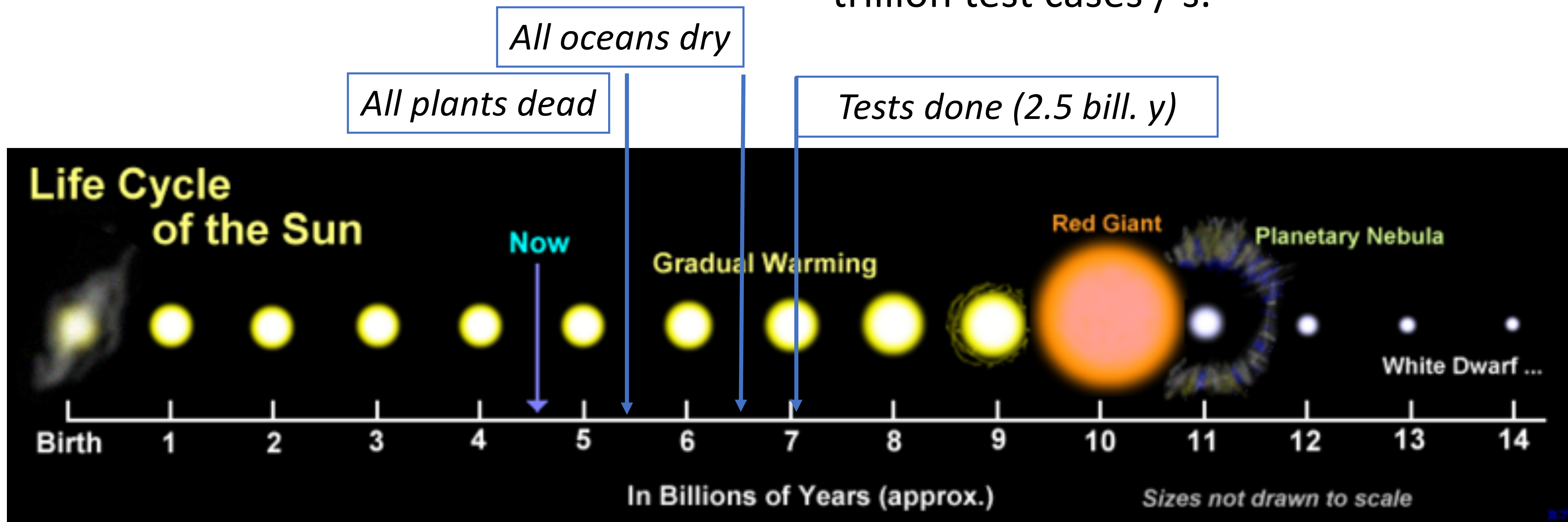
Principles of Testing #2: Exhaustive Testing is ………………

1.3.2

Principle #2: Exhaustive Testing is Impossible

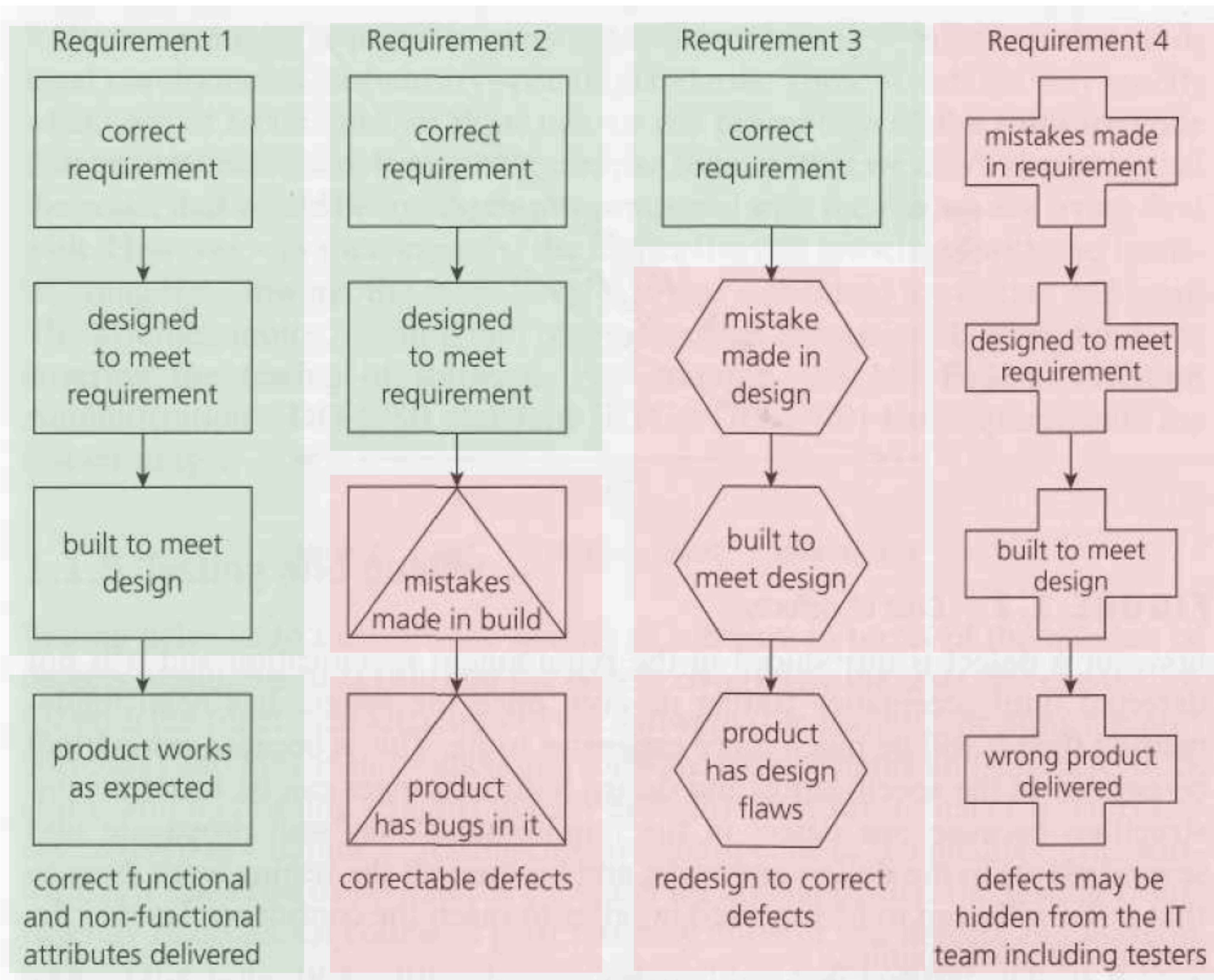
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- A simple program: 3 inputs, 1 output
- a,b,c: 32 bit integers
- trillion test cases / s.



Principles of Testing #3: *Test Early*

- Start testing as early as possible
  - To let tests guide design
  - To get feedback as early as possible
  - To find bugs when they are cheapest to fix
  - To find bugs when have caused least damage



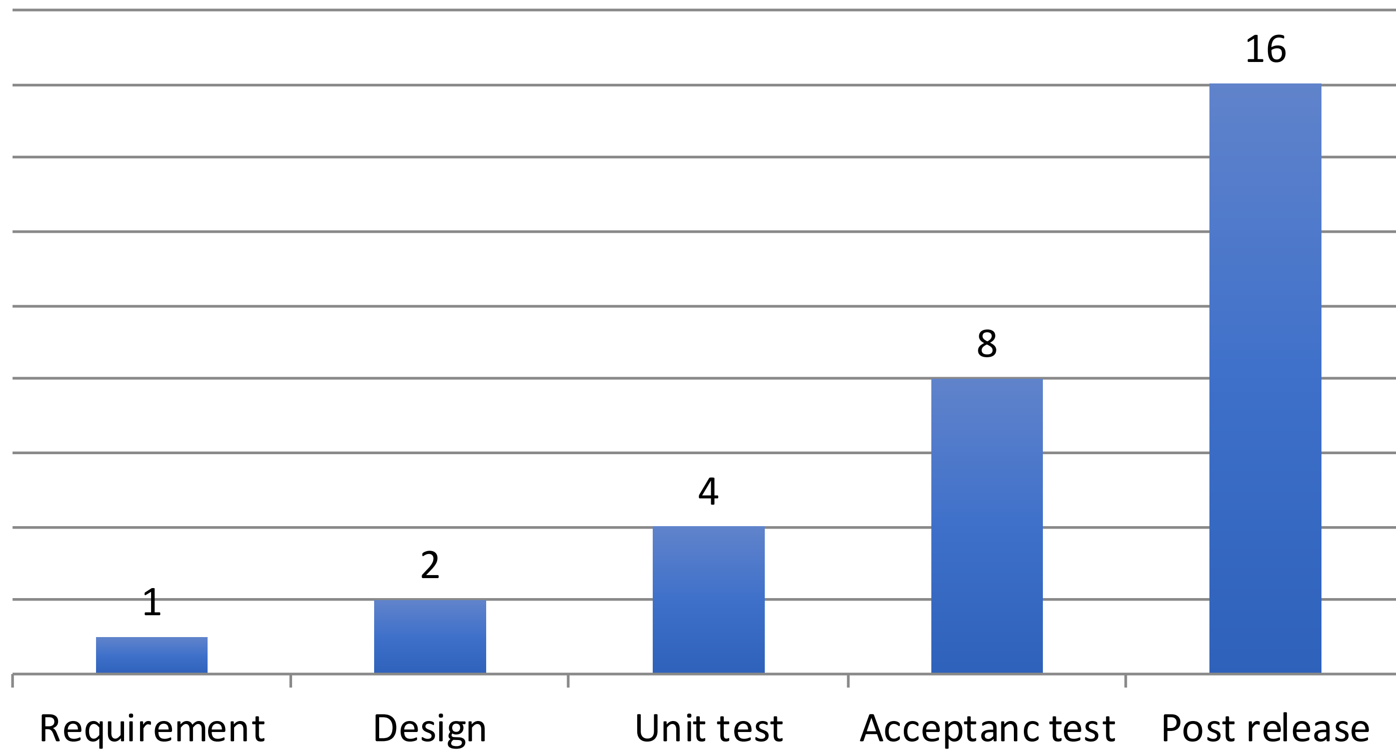
1.1.2
1.3.3

Faults can be introduced at any moment in the software development process

Finding faults in different development phases may require different types of testing

Cost to Repair: Early Discovery Pays Off

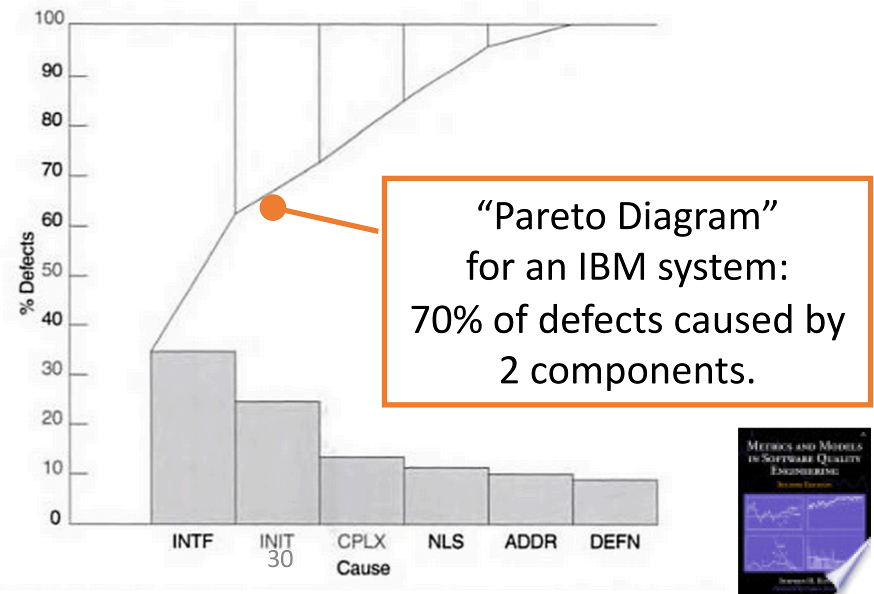
1.1.2



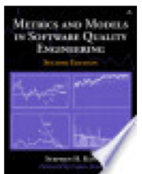
Principles of Testing #4:
Defects are likely to be Clustered

If you find a bug, keep on searching in its 'neighborhood'

- "Hot" components requiring frequent change, bad habits, poor developers, tricky logic, business uncertainty, innovative, size, ….
- Pareto Principle / Law of vital few:
  - *80% of effects come from 20% of causes*
- *Use to focus test effort*



"Pareto Diagram" for an IBM system: 70% of defects caused by 2 components.



Principles of Testing #5:
Is there one best test method for my project?

The <u>pesticide paradox:</u>

Every method you use to prevent or find bugs
leaves a residue of subtler bugs
against which those methods are ineffectual.

- Re-running the same test suite again and again on a changing program gives a false sense of security
- We need *variation in testing*

Principles of Testing #6:
Is there a single test method for any project?

- Testing is context-dependent

antonpp
@mordbergak
Follow

The infamous "null to null" trip with a stopover in Boston, thanks @AerLingus!

via Boston
Departs in 11 hours
Trip Details

null to null
Sun 29 Apr 2018 @ 16:20
1h01m Stopover in Boston

null to null
Fri 04 May 2018 @ 08:00
1h19m Stopover in Boston

CHECK IN IS NOT AVAILABLE

Principles of Testing #7
Absence-of-errors Fallacy

1.3.7

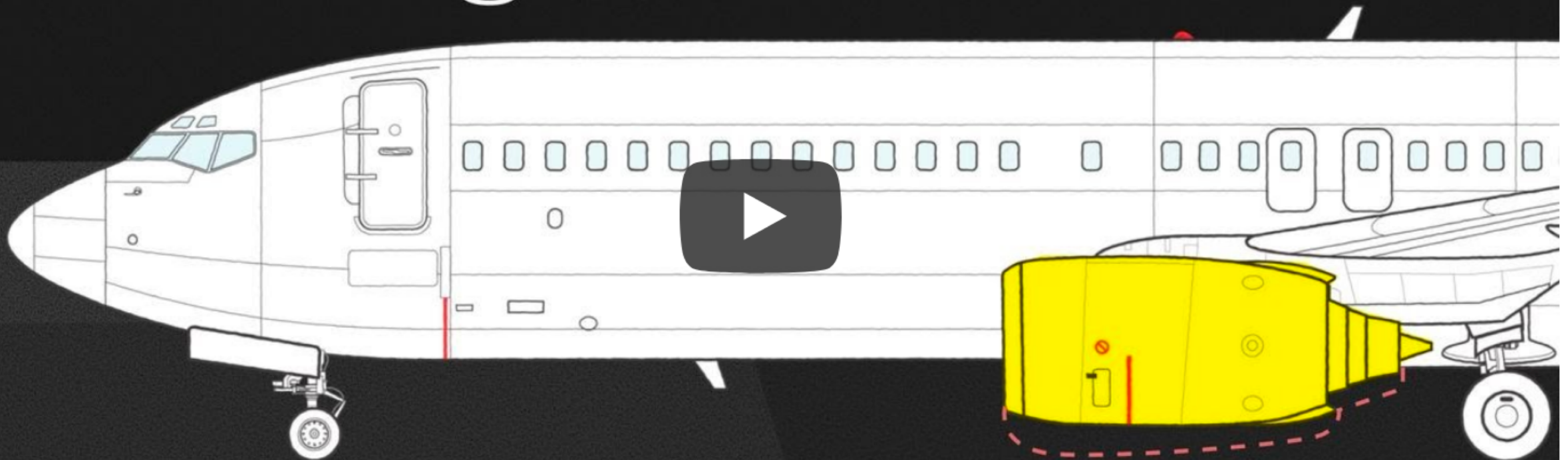
There is more to success than absence of errors

Thorough understanding of business value is necessary

“Building the software right versus building the right software.”

“Finding and removing defects is not a way to improve the overall quality or performance of a system” – Russ Ackoff

https://embeddedartistry.com/blog/2019/2/5/beyond-continual-improvement



https://www.youtube.com/watch?v=H2tuKiiznsY

http://www.boeing.com/commercial/737max/737-max-software-updates.page

Some testing quotes

- “completed a 120 737 test flights”
- “totaling more than 203 hours of air time”
- “with the updated system”
- “with our leaders on board the airplane”
- “operating as designed across a range of flight conditions”
- ”have experienced the new software through simulator sessions”
- “The team of Boeing pilots, engineers, technical experts and our partners were comprehensively testing the software

Revisiting the Principles?

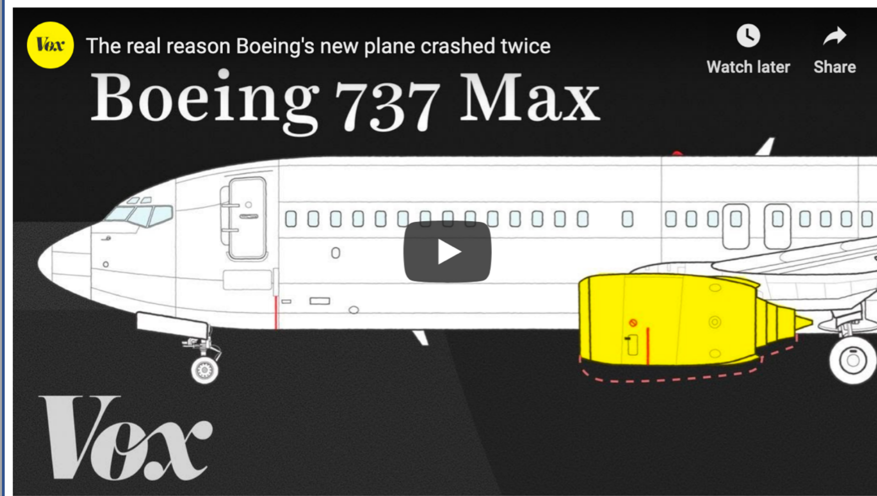
1. Testing cannot show absence of bugs
2. Exhaustive testing is impossible
3. Testing needs to start early
4. Defects tend to be clustered
5. Pesticide paradox yields test methods ineffective
6. Testing is context-dependent
7. There is more to quality than absence of defects

Boeing's effort to get the 737 Max approved to fly again, explained

A bigger problem than a software update.

By Matthew Yglesias | @mattyglesias | matt@vox.com | Apr 5, 2019, 10:30am EDT

SHARE



https://www.vox.com/2019/4/5/18296646/boeing-737-max-mcas-software-update

https://embeddedartistry.com/blog/2019/4/1/what-can-software-organizations-learn-from-the-boeing-737-...

April 1, 2019

What can software organizations learn from the Boeing 737 MAX saga?

Phillip Johnston · Product Development, Monday Morning Reading

One of the largest news stories over the past month was the grounding of Boeing 737 MAX-8 and MAX-9 aircraft after an Ethiopian Airlines crash resulted in the deaths of everyone on board. This is the second deadly crash of involving a Boeing 737 MAX. A Lion Air Boeing 737 MAX-8 crashed in October 2018, also killing everyone on board. As a result of these two crashes, Boeing 737 MAX airplanes have been temporarily grounded in over 41 countries, including China, the US, and Canada. Boeing also paused delivery of these planes, although they are continuing to produce them.

https://embeddedartistry.com/blog/2019/4/1/what-can-software-organizations-learn-from-the-boeing-737-max-saga

1.5

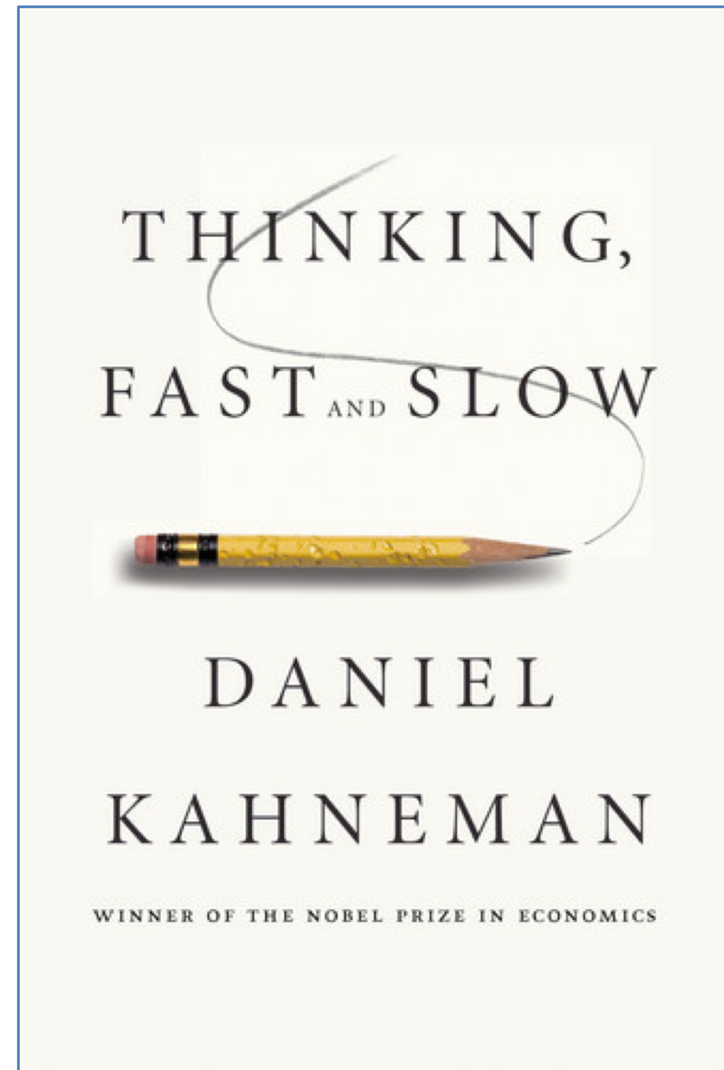
Psychology of Testing

- We are all biased
- *Independence of testing* required:
  - From self-testing to external approval
  - Testers with different backgrounds
- Good knowledge of *rigorous techniques and procedures* required

Cognitive Bias

- “System 1”: Fast, instinctive, emotional.
- "System 2”: slower, more deliberative, and more logical.

- System 2 requires *effort* and is happy to let System 1 do the work



Example of Cognitive Bias

- **“What You See is All There Is”** (WYSIATI)
  - Being satisfied with the evidence you see.
- Key problem in software engineering:
  - “It works on my machine”
  - “I’ve tried it, and it works”
  - “All 100 tests pass, we can ship”

FAA and the 737 MAX

- "The FAA delegates some certification and technical assessments to airplane manufacturers, citing lack of funding and resources to carry out all operations internally"
- FAA safety engineer:
  - "We were asked by management to re-evaluate what would be delegated. Management thought we had retained too much at the FAA."
- **"There wasn't a complete and proper review of the documents. Review was rushed to reach certain certification dates."**

https://embeddedartistry.com/blog/2019/4/1/what-can-software-organizations-learn-from-the-boeing-737-max-saga

Grady Booch
@Grady_Booch

Follow

Every line of code represents an ethical and moral decision.

1.6

ACM Software Engineering Code of Ethics

1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

2.4 Accept and provide appropriate professional review.

High quality professional work in computing depends on professional review at all stages. Whenever appropriate, computing professionals should seek and utilize peer and stakeholder review. Computing professionals should also provide constructive, critical reviews of others' work.

2.5 Give comprehensive and thorough evaluations of computer systems and their impacts, including analysis of possible risks.

Computing professionals are in a position of trust, and therefore have a special responsibility to provide objective, credible evaluations and testimony to employers, employees, clients, users, and the public. Computing professionals should strive to be perceptive, thorough, and objective when evaluating, recommending, and presenting system descriptions and alternatives. Extraordinary care should be taken to identify and mitigate potential risks in machine learning systems. A system for which future risks cannot be reliably predicted requires frequent reassessment of risk as the system evolves in use, or it should not be deployed. Any issues that might result in major risk must be reported to appropriate parties.

https://www.acm.org/code-of-ethics

2.2 Maintain high standards of professional competence, conduct, and ethical practice.

High quality computing depends on individuals and teams who take personal and group responsibility for acquiring and maintaining professional competence. Professional competence starts with technical knowledge and with awareness of the social context in which their work may be deployed. Professional competence also requires skill in communication, in reflective analysis, and in recognizing and navigating ethical challenges. Upgrading skills should be an ongoing process and might include independent study, attending conferences or seminars, and other informal or formal education. Professional organizations and employers should encourage and facilitate these activities.

https://www.acm.org/code-of-ethics

First Student-Provided Question!

Which test level corresponds to testing a system to check if individual component are fulfilling functionalities?

A. Acceptance Testing
B. Integration Testing
C. System Testing
D. Unit Testing

First Student-Provided Question!

Which test level corresponds to testing a system to check if individual component are fulfilling functionalities?

A. Acceptance Testing
B. Integration Testing
C. System Testing

D. Unit Testing

Which of the Following is Correct?

A. Finding a mistake requires executing a program
B. Finding a failure requires executing a program
C. Finding a defect requires executing a program.
D. Mistakes, defects, and failures can all be found without executing the program.

Which of the Following is Correct?

A. Finding a mistake requires executing a program

B. Finding a failure requires executing a program

C. Finding a defect requires executing a program.

D. Mistakes, defects, and failures can all be found without executing the program.

Software Life Cycle [Chapter 2]

Ch.2
Gloss.

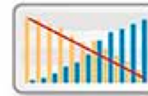
- Period of time that
  - begins when a software system is conceived
  - ends when the system is no longer available for use.

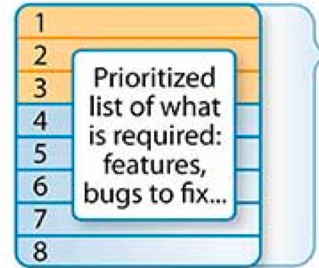
- Phases: concept development, requirements, design, implementation, test, installation, retirement

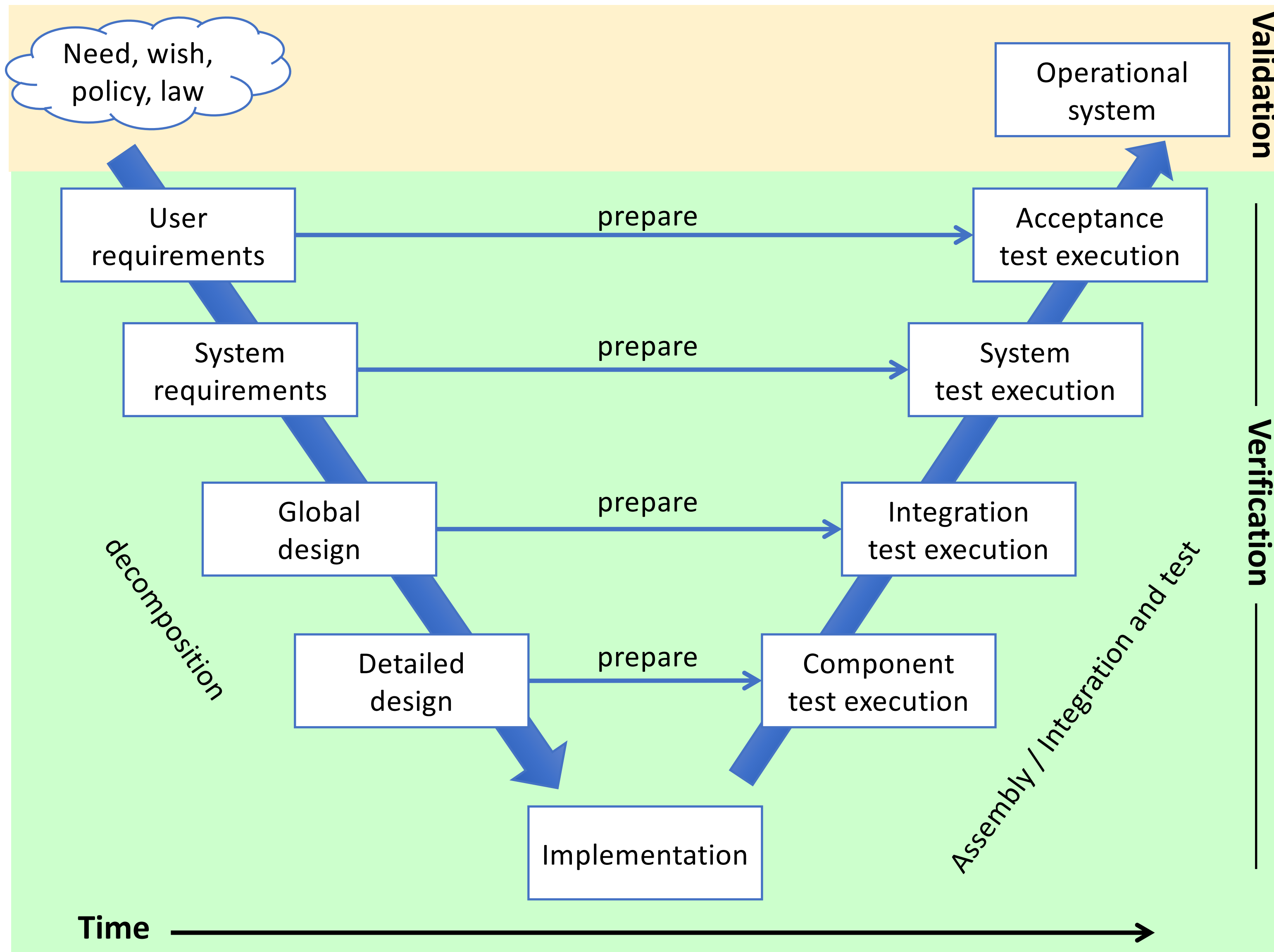
- Phases may overlap and be performed iteratively

2.1.2

The Agile Scrum Framework at a glance






Need, wish, policy, law
Operational system
Validation
User requirements
prepare
Acceptance test execution
System requirements
prepare
System test execution
Global design
prepare
Integration test execution
Verification
Detailed design
prepare
Component test execution
decomposition
Assembly / Integration and test
Implementation
Time

2.1.1

Verification versus Validation

When you get what you want …

Verification:

does the software system meet the requirements specifications?

Are we building the software right?

… but not what you need?

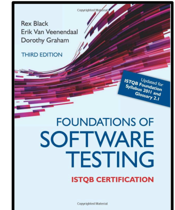
Validation:

does the software system meet the user's real needs?

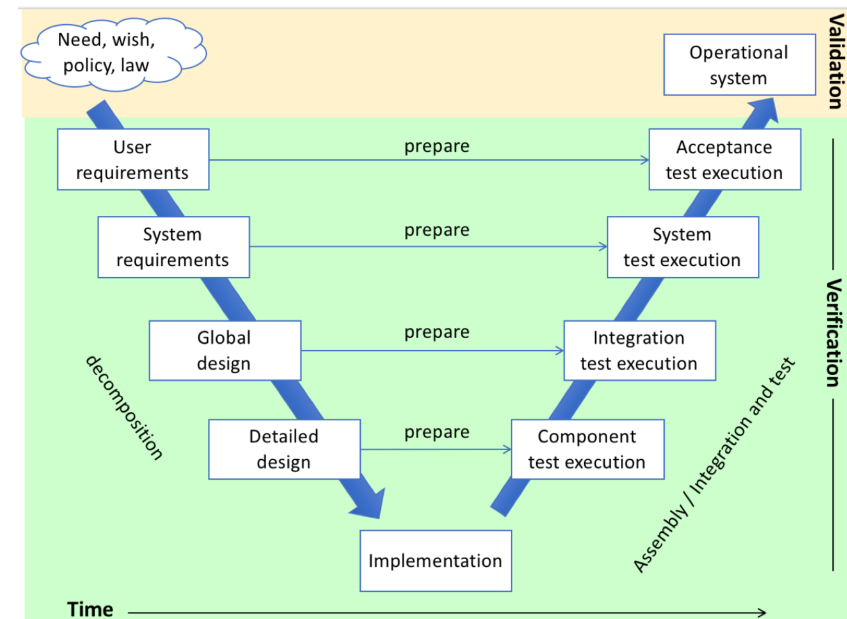
Are we building the right software?

Test Levels

2.2



- Component (unit) testing.
  Units in isolation
- Integration testing
  Interaction between units
- System testing
  System-level properties
- Acceptance testing
  Focus on user needs



Test levels in *right leg of V model*

Systems Thinking / Russ Ackoff

- The defining properties of a system are properties of the whole which none of its parts have
- A system is not the sum of the behavior of its parts, it is a product of their interactions
- The performance of a system depends on how the parts fit, not how they act taken separately

https://embeddedartistry.com/blog/2019/2/5/beyond-continual-improvement

2.3

Test Types

Group of test activities

Aimed at testing a component or system

Focused on a specific test objective

2.3

Test Types

- Testing of Function
  - Functional testing, black box testing
- Testing of software product characteristics
  - Non-functional testing
- Testing of software structure / architecture
  - Structural testing, white box testing
- Testing related to changes
  - Confirmation vs Regression testing

2.3.4

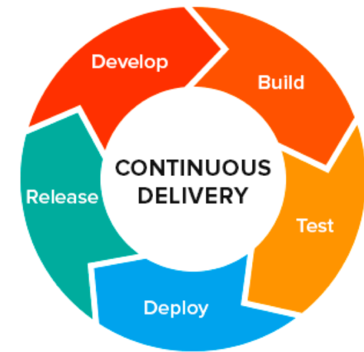
Regression Testing

- Testing of a previously tested program
  - Following modification
  - To ensure that defects have not been introduced
  - In unchanged areas of the software
  - As a result of the changes made.

- Performed when the software or its environment is changed.

- Continuous delivery? *Automate regression testing*

Testing As Software Evolves



- The norm in software development!
  - Many updates *per day* in modern web apps
  - Current cars (wireless updates)
  - Formula 1 (every two weeks)
  - Aircraft (737 max compatibility)
- Dependency and compatibility management are key
- Direct test efforts to changed software
  - Confirm change works as expected, without regressions.
  - Automated regression testing where possible

2.4

Maintenance Testing (ISTQB)

- Testing after a system is stable and deployed
  - Test changes to an operational system
  - Test impact of *changed environment* to an operational system (e.g. security updates of libraries used)

- Impact analysis:
  - Determine which parts are affected by change
  - Conduct *regression testing* for those.

Chapter 2 Key Points

- Software development follows life cycle activities
- The V-Model helps reason about verification vs validation, decomposition vs composition, and construction vs testing.
- We test at different levels
- Different test types target specific test objectives
- Iterative projects do continuous regression testing and thus maximize test automation

Static versus Dynamic Testing

Static Testing:
Testing of a component or system
at specification or implementation level
without execution of software
(e.g., reviews or static analysis)

Dynamic Testing:
Testing that involves the execution
of the software of a component or system.

3.2.1
3.2.2

Formal Reviews: Phases & Roles

Review Phases
- Planning
- Kick-off
- Preparation
- Review meeting
- Rework
- Follow-up

Reviewing Roles
- Moderator
- Author
- Scribe
- Reviewers
- Managers

"If you did not document it, you did not do it!"

- Inspections by Government & Notified Bodies:
  - If you do not follow regulation & your internal procedures (QMS), you cannot guarantee safety & effectiveness.
- Consequences:
  - Delivery stop for sites outside USA and/or close down for sites in the USA.
  - In case of safety (patient) issues & not sticking to the law: Jail for upper mgt.
- At Philips:
  - Inspection Back-office to answer questions fast & accurately

PHILIPS

Philips Quality Management System

iXR
IVVR Process Pyramid

LEGEND

Process

Procedures

Work instructions

Forms & Templates

Guidelines & Tutorials

Revision V1.6 22-Feb-2011

3.2.3

Types of Review

- **Walkthrough**
  - Author in the lead
- **Technical Review:**
  - Technical meeting to achieve consensus
- **Inspection:**
  - Peer review of documents
  - Relies on ‘visual inspection’ (= reading)
  - To detect defects / violations

Expectations, Outcomes, and Challenges of Modern Code Review

Alberto Bacchelli
REVEAL @ Faculty of Informatics
University of Lugano, Switzerland
alberto.bacchelli@usi.ch

Christian Bird
Microsoft Research
Redmond, Washington, USA
cbird@microsoft.com

***Abstract*—Code review is a common software engineering practice employed both in open source and industrial contexts. Review today is less formal and more "lightweight" than the code inspections performed and studied in the 70s and 80s. We empirically explore the motivations, challenges, and outcomes of tool-based code reviews. We observed, interviewed, and surveyed developers and managers and manually classified hundreds of review comments across diverse teams at Microsoft. Our study reveals that while finding defects remains the main motivation for review, reviews are less about defects than expected and instead provide additional benefits such as knowledge transfer, increased team awareness, and creation of alternative solutions to problems. Moreover, we find that code and change understanding is the key aspect of code reviewing and that developers employ a wide range of mechanisms to meet their understanding needs, most of which are not met by current tools. We provide recommendations for practitioners and researchers.**

I. Introduction

Peer code review, a manual inspection of source code by developers other than the author, is recognized as a valuable tool for reducing software defects and improving the quality of software projects [2], [1]. In 1976, Fagan formalized a highly structured process for code reviewing [13], based on line-by-line group reviews, done in extended meetings—*code inspections*. Over the years, researchers provided evidence on code inspection's benefits, especially in terms of defect finding, but the cumbersome, time-consuming, and synchronous nature of this approach hinders its universal adoption in practice [32].

Nowadays many organizations are adopting more lightweight code review practices to limit the inefficiencies of inspections. In particular, there is a clear trend toward the usage of tools developed to support code review [28]. In the context of this paper, we define *Modern Code Review*, as review that is (1) informal (in contrast to Fagan-style), (2) tool-based, and that (3) occurs regularly in practice nowadays, for example at companies such as Microsoft, Google [19], Facebook [36], and in other organizations and open source software (OSS) projects [40].

This trend raises questions, such as: What are the expectations for code review nowadays? What are the actual outcomes of code review? What challenges do people face in code review?

Answers to these questions can provide insight for both practitioners and researchers. Developers and other software project stakeholders can use empirical evidence about expectations and outcomes to make informed decisions about when to use code review and how it should fit into their development process. Researchers can focus their attention on the challenges faced by practitioners to make code review more effective.

We present an in-depth study of practices in teams that use modern code review, revealing what practitioners think, do, and achieve when it comes to modern code review.

Since Microsoft is made up of many different teams working on very diverse products, it gives the opportunity to study teams performing code review *in situ* and understand their expectations, the benefits they derive from code review, the needs they have, and the problems they face.

We set up our study as an exploratory investigation. We started without *a priori* hypotheses regarding how and why code review should be performed, with the aim of discovering what developers and managers expect from code review, how reviews are conducted in practice, and what the actual outcomes and challenges are. To that end, we (1) observed 17 industrial developers performing code review with various degrees of experience and seniority across 16 separate product teams with distinct reviewing cultures and policies; (2) interviewed these developers using a semi-structured interviews; (3) manually inspected and classified the content of 570 comments in discussions contained within code reviews; and (4) surveyed 165 managers and 873 programmers.

Our results show that, although the top motivation driving code reviews is finding defects, the practice and the actual outcomes are less about finding errors than expected: Defect related comments comprise a small proportion and mainly cover small logical low-level issues. On the other hand, code review additionally provides a wide spectrum of benefits to software teams, such as knowledge transfer, team awareness, and improved solutions to problems. Moreover, we found that context and change understanding is the key of any review. According to the outcomes they want to achieve, developers employ many mechanisms to fulfill their understanding needs, most of which are not currently met by any code review tool.

This paper makes the following contributions:

- Characterizing the motivations of developers and managers for code review and compare with actual outcomes.
- Relating the outcomes to understanding needs and discuss how developers achieve such needs.

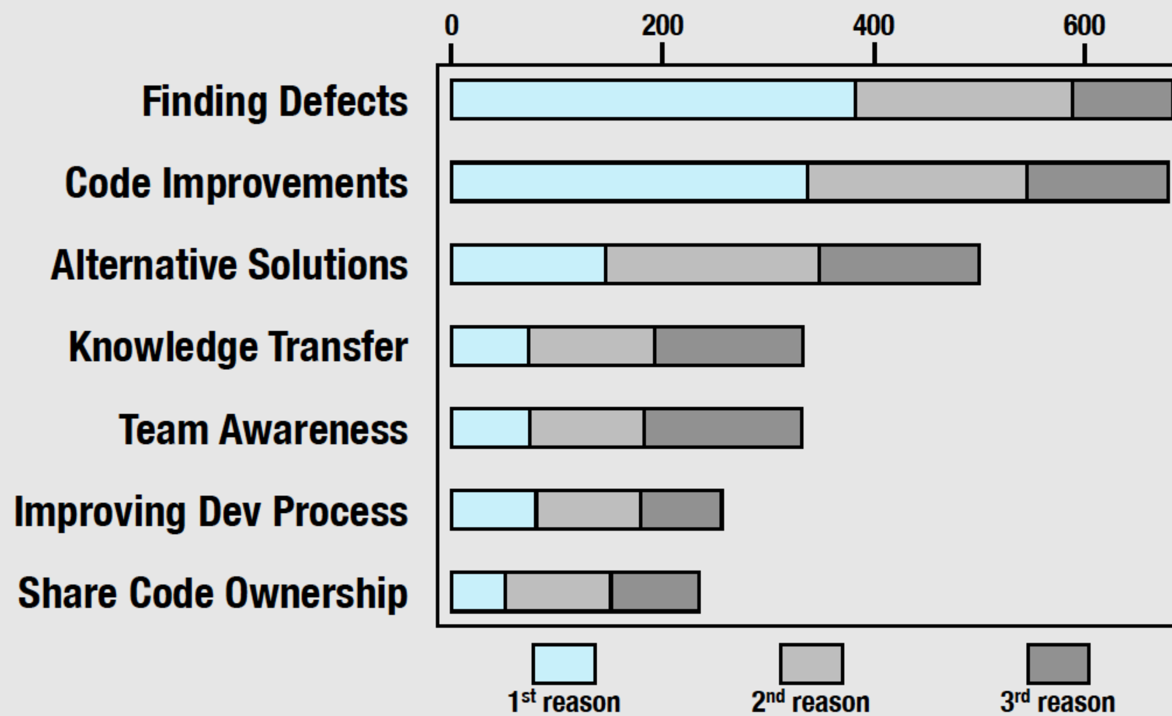
Based on our findings, we provide recommendations for practitioners and implications for researchers as well as outline future avenues for research.

978-1-4673-3076-3/13 © 2013 IEEE ICSE 2013, San Francisco, CA, USA

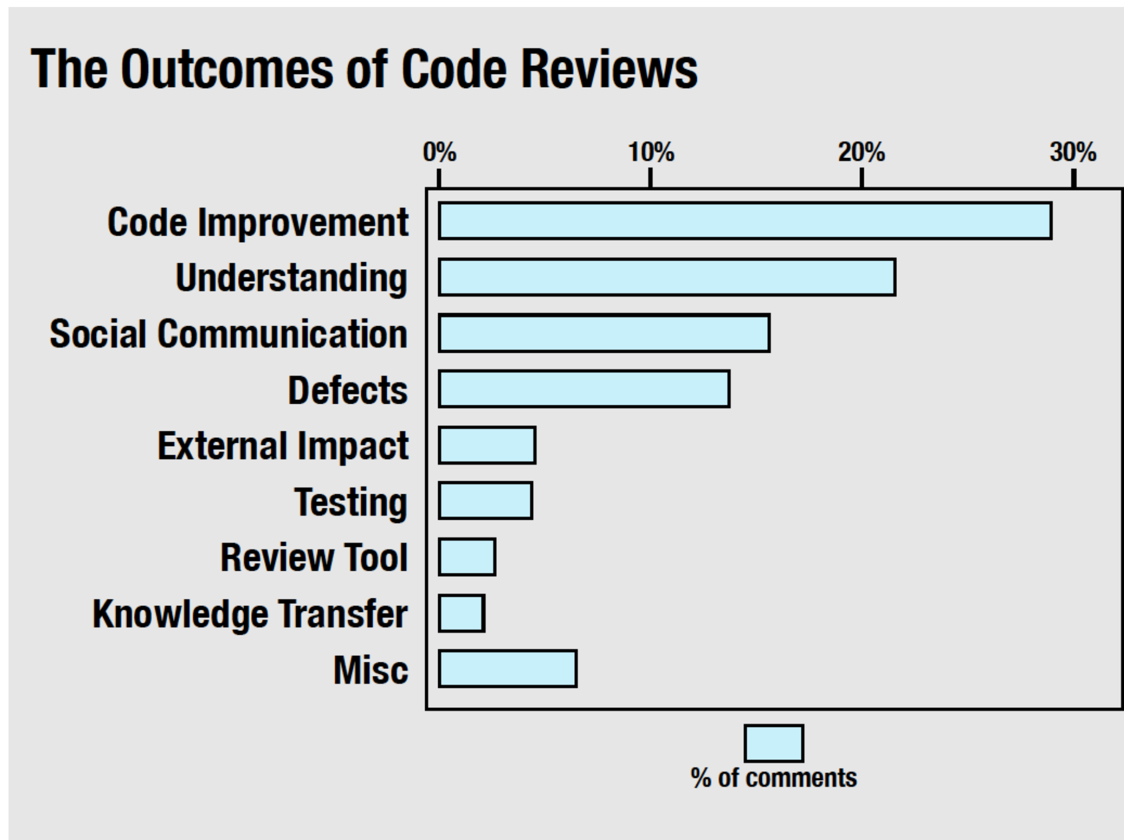
Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Why Do Programmers Do Code Reviews?

What Programmers Say:



Versus What They Actually Do:



http://alex.nederlof.com/blog/2013/05/24/the-truth-about-code-reviews/

MICHAELA GREILER

CODE REVIEW AT MICROSOFT

How does code review work at one of the largest software companies?

https://www.michaelagreiler.com/code-reviews-at-microsoft-how-to-code-review-at-a-large-software-company/

3.3

Static Analysis Tools

- Coding standards
- List of rules
- (Dis/en)abling rules
- Reviewing warnings
- False positives

- *Use IDE plugins to see warnings*
- *Agree in team which rule sets to be used.*

- Checkstyle:
  - Basic formatting

- PMD
  - Design faults

- ~~Find~~ SpotBugs
  - Design faults

- All: *libraries of configurable rules.*
- *.pmd, checkstyle.xml*

Not Secure | checkstyle.sourceforge.net/checks.html

About

- Checkstyle
- Release Notes
- Consulting
- Sponsoring

Documentation

- Configuration
  - Property Types
  - Filters
  - File Filters
- Running
  - Ant Task
  - Command Line
- **Checks**
  - Annotations
  - Block Checks
  - Class Design
  - Coding
  - Headers
  - Imports
  - Javadoc Comments
  - Metrics
  - Miscellaneous
  - Modifiers
  - Naming Conventions
  - Regexp
  - Size Violations
  - Whitespace
- Style Configurations
  - Google's Style
  - Sun's Style

Developers

- Extending Checkstyle
  - Writing Checks

Standard Checks

The Standard Checkstyle Checks are applicable to general Java coding style and require no external libraries. The standard checks are included in the base distribution.

The site navigation menu lets you browse the individual checks by functionality.

Checkstyle provides many checks that you can apply to your source code. Below is an alphabetical reference, the site navigation menu provides a reference organized by functionality.

AbbreviationAsWordInName	The Check validate abbreviations(consecutive capital letters) length in identifier name, it also allow in enforce camel case naming.
AbstractClassName	Ensures that the names of abstract classes conforming to some regular expression and check that `abstract` modifier exists.
AnnotationLocation	Check location of annotation on language elements.
AnnotationOnSameLine	The check does verifying that annotations are located on the same line with their targets.
AnnotationUseStyle	This check controls the style with the usage of annotations.
AnonInnerLength	Checks for long anonymous inner classes.
ArrayTrailingComma	Checks if array initialization contains optional trailing comma.
ArrayTypeStyle	Checks the style of array type definitions.

https://pmd.github.io/latest/pmd_rules_java_design.html#c...

PMD Source Code Analyzer Project

Nav | Download | Fork us on github | search...

CouplingBetweenObjects

Since: PMD 1.04

Priority: Medium (3)

This rule counts the number of unique attributes, local variables, and return types within an object. A number higher than the specified threshold can indicate a high degree of coupling.

This rule is defined by the following Java class:
net.sourceforge.pmd.lang.java.rule.design.CouplingBetweenObjectsRule

Example(s):

```
import com.Blah;
import org.Bar;
import org.Bardo;

public class Foo {
    private Blah var1;
    private Bar var2;

    //followed by many imports of unique objects
    void ObjectC doWork() {
        Bardo var55;
        ObjectA var44;
        ObjectZ var93;
        return something;
    }
}
```

This rule has the following properties:

Name	Default Value	Description	Multivalued
threshold	20	Unique type reporting threshold	no

Many rules are *metric-based.*

Issue warning if metric is above threshold.

https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#il-an-apparent-infinite-loop-il-infinite-loop

⊟ Correctness (CORRECTNESS)

NP: Method with Optional return type returns explicit null (NP_OPTIONAL_RETURN_NULL)

NP: Non-null field is not initialized (NP_NONNULL_FIELD_NOT_INITIALIZED_

VR: Class makes reference to unresolvable class or method (VR_UNRESOLVABLE_REFERENCE)

IL: An apparent infinite loop (IL_INFINITE_LOOP)

IO: Doomed attempt to append to an object output stream (IO_APPENDING_TO_OBJECT_OUTPUT_S

IL: An apparent infinite recursive loop (IL_INFINITE_RECURSIVE_LOOP)

IL: A collection is added to itself (IL_CONTAINER_ADDED_TO_ITSELF)

RpC: Repeated conditional tests (RpC_REPEATED_CONDITIONAL_TEST)

FL: Method performs math using floating point precision (FL_MATH_USING_FLOAT_PRECISION)

CAA: Possibly incompatible

Correctness (CORRECTNESS)

Probable bug - an apparent coding mistake resulting in code that was probably not what the developer intended. We strive for a low false positive rate.

NP: Method with Optional return type returns explicit null (NP_OPTIONAL_RETURN_NULL)

The usage of Optional return type (java.util.Optional or com.google.common.base.Optional) always means that explicit null returns were not desired by design. Returning a null value in such case is a contract violation and will most likely break client code.

NP: Non-null field is not initialized (NP_NONNULL_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR)

The field is marked as non-null, but isn't written to by the constructor. The field might be initialized elsewhere during constructor, or might always be initialized before use.

VR: Class makes reference to unresolvable class or method (VR_UNRESOLVABLE_REFERENCE)

This class makes a reference to a class or method that can not be resolved using against the libraries it is being analyzed with.

False Positive / Negative

- Many static analysis tools based on *heuristics*
  - Correct positive: Warning, and a problem (let's fix it! ☺)
  - Correct negative: No warning, no problem. (no need to act ☺)

  - False positive: Warning, but not a problem (annoying ☹)
  - False negative: Problem, but no warning (possibly dangerous ☹)

Comply or Explain

- Rule checking is enabled because rules are right in **99%** of the cases.
- For the 1% unjustified code warnings:
  - Add appropriate @SuppressWarnings of specific rule.

- For your gitlab regrets:
  - Add comment in report explaining how you now would do things differently

Merged **Suppress spotbugs false positive in Java 11** #27

Changes from 1 commit ▾ File filter... ▾ ✕ Clear filters Jump to... ▾

Diff settings ▾ Review changes ▾

6 src/main/java/nl/tudelft/jpacman/sprite/SpriteStore.java

```
 2    2
      3  + import edu.umd.cs.findbugs.annotations.SuppressFBWarnings;
      4  +
 3    5    import java.awt.image.BufferedImage;
 4    6    import java.io.IOException;
 5    7    import java.io.InputStream;
@@ -57,6 +59,10 @@ public Sprite loadSprite(String resource) throws IOException {
57   59         * @throws IOException
58   60         *             When the resource could not be loaded.
59   61         */
     62  +     @SuppressFBWarnings(
     63  +         value = "RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE",
     64  +         justification = "false positive in java 11"
     65  +     )
60   66        private Sprite loadSpriteFromResource(String resource) throws IOException {
61   67            try (InputStream input = SpriteStore.class.getResourceAsStream(resource)) {
62   68                if (input == null) {
```

Types of Static Analysis Tools

- Lexical: Words, strings, and regexps.
- Syntactic: Tree of program structure
- Control flow graph
- Data flow graph
- Symbolic execution

(Code) Metrics

- Goal: “Assess software complexity”
- Question: “How complex is the conditional logic in the code?”
- Metrics:
  - “Nr of if-statements”,
  - “Nr of conditions in if-statements”
  - “Nr of switch statements”
  - …

Example Class-Level Metrics

- Volume: Nr of lines
- Complexity: Nr of if-statements per method
- Coupling: Nr of classes a class depends on
- (Lack of) Cohesion: correlation between variables and methods

McCabe's Cyclomatic Complexity

3.3.2

- C = |E| - |N| + 2
- C = # decision points + 1
- C = # of if-statements + 1
  (+ #switch cases)

C > 10: method too complex [McCabe, 1976]

[C correlated with #lines of code]

Will be discussed when as part of "code coverage" lecture

File | /Users/arie/dev/jpacman/build/reports/jacoco/test/html/index.html

jpacman

Cyclomatic Complexity

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	L
nl.tudelft.jpacman.level		70%		59%	70	151	95	
nl.tudelft.jpacman.ui		77%		47%	54	86	21	
nl.tudelft.jpacman.npc.ghost		85%		62%	43	89	15	
default		0%		0%	12	12	21	
nl.tudelft.jpacman.board		86%		58%	44	93	2	
nl.tudelft.jpacman.sprite		86%		59%	30	70	11	
nl.tudelft.jpacman		69%		25%	12	30	18	
nl.tudelft.jpacman.points		60%		75%	1	11	5	
nl.tudelft.jpacman.game		87%		60%	10	24	4	
nl.tudelft.jpacman.npc		97%		83%	1	8	1	
Total	1,013 of 4,494	77%	269 of 613	56%	277	574	193	1,

Software Metrics Best Practices

- Raw values need to context to become meaningful
  - Use a benchmark
  - Focus on trends
  - Focus on worst (highest risk) 10%
- Measure with a clear goal
- Metric may signal problem
  - Treat the problem, not the metric
- Adopt palette of metrics measuring different characteristics

E. Bouwers, J. Visser, and A. van Deursen. Getting what you Measure. CACM, May 2012

prac

DOI:10.1145/2209249.2209266

Article development led by acmqueue
queue.acm.org

Four common pitfalls in using software metrics for project management.

BY ERIC BOUWERS, JOOST VISSER, AND ARIE VAN DEURSEN

Getting What You Measure

ARE SOFTWARE METRICS helpful tools or a waste of time?
For every developer who treasures these mathematical abstractions of software systems there is a developer who thinks software metrics are invented just to keep project managers busy. Software metrics can be very powerful tools that help achieve your goals but it is important to use them correctly, as they also have the power to demotivate project teams and steer development in the wrong direction.
For the past 11 years, the Software Improvement

- Metric in a bubble;
- Treating the metric;
- One-track metric; and
- Metrics galore.

Knowing about these pitfalls will help you recognize them and, hopefully, avoid them, which ultimately leads to making your project successful. As a software engineer, your knowledge of these pitfalls helps you understand why project managers want to use software metrics and helps you assist the managers when they are applying metrics in an inefficient manner. As an outside consultant, you need to take the pitfalls into account when presenting advice and proposing actions. Finally, if you are doing research in the area of software metrics, knowing these pitfalls will help place your new metric in the right context when presenting it to practitioners. Before diving into the pitfalls, let's look at why software metrics can be considered a useful tool.

Software Metrics Steer People
"You get what you measure." This phrase definitely applies to software project teams. No matter what you define as a metric, as soon as it is used to evaluate a team, the value of the metric moves toward the desired value. Thus, to reach a particular goal, you can continuously measure properties of the desired goal and plot these measurements in a place visible to the team. Ideally, the desired goal is plotted alongside the current measurement to

FIGURE 1 Lines of Code of a Software System

(thousands)
350
300
250
200
150
100
50
0
lines of code
Jan 2010
Mar 2010
May 2010
July 2010
Sept 2010
Nov 2010
Jan 2011
Mar 2011
May 2011
July 2011

[Intermezzo: *Undecidable Problems*]

- Most interesting properties you'd want to test are *undecidable.*
- Thus, even with the cleverest AI possible, there is no systematic procedure to demonstrate such properties.
- Such properties can be reduced to the "Halting Problem":
  - A general algorithm to the problem of determining whether a program and a given input will halt or not *cannot exist* for all program-input pairs.

Chapter 3 Key Points

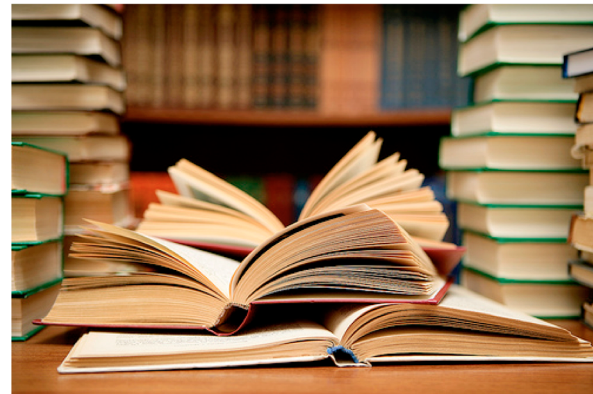
- Static analysis is based on code analysis, not code execution
- Many properties of interest are undecidable
- Static analysis is often based on heuristics
  - Can lead to false positives or negatives
- Static analysis are based on rule sets
  - Agree on one, then comply or explain
- Code review is also about knowledge transfer
- Interpreting metrics requires context and root cause analysis

How can I know that my program won't stop?

- Can I just try it?
- You may have to wait infinitely long

How can I know that my program won't stop?

- Can I carefully read it?
- That may help.
  But real programs are very complex.



http://www.flickr.com/photos/abee5/8314929977/

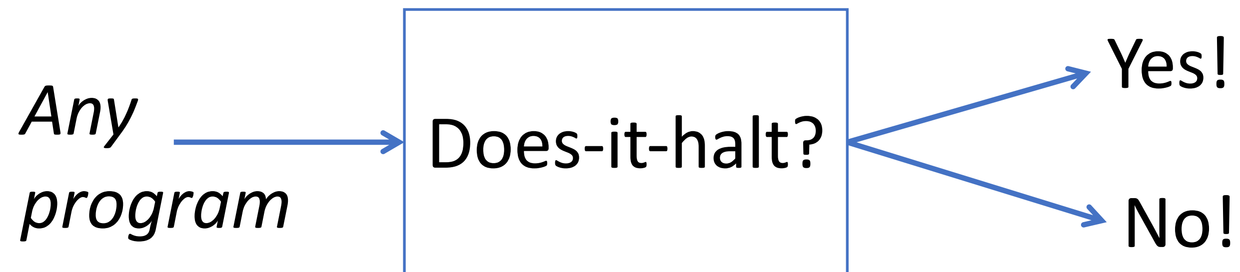
How can I know that my program won’t stop?

- Isn’t there some other program that can tell me if my program will ever stop?

- *Can’t TU Delft come up with a clever program that can check for any other program whether it will loop for ever?*

TU Delft

Wanted!



But provably impossible to make!!