Software Testing, Verification and Validation

September 21, 2022
Week #2 — Lecture #1

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Name: José Campos
E-mail: jcmc@fe.up.pt
Webpage: https://jose.github.io

Speciality: software testing

TVVS 2022/2023

Official website:
https://sigarra.up.pt/feup/en/ucurr_geral.ficha_uc_view?pv_ocorrencia_id=501971
Alternative: https://paginas.fe.up.pt/~jcmc/tvvs/2022-2023/index.html

- Verification vs Validation
- Static and Dynamic Testing
- Black-box Testing
    - Equivalence Class Partitioning
    - Boundary Value Analysis
    - Model-based Testing
- White-box Testing
    - Structural Testing (Line and Decision coverage, Path coverage)
    - Logical Coverage (Condition coverage, Modified Condition/Decision Coverage (MC/DC))
    - Dataflow Testing
    - Mutation Testing
- Regression Testing
- Integration Testing, System Testing, Acceptance Testing
- Advanced Testing techniques, e.g., Search-based Software Testing
- Tests Planning and Documentation, Defect Management

TVVS 2022/2023

- 1 x Lecture class (MESW and M.EIC)
  - Wednesday 11:30am — 1:00pm
  - Room: B006

- 1 x Recitation class (MESW)
  - Friday 9:30am — 11:00am
  - Room: B109

- 1 x Recitation class (M.EIC)
  - Friday 11:30am — 1:00pm
  - Room: B107

- 1 x Office hour (MESW and M.EIC)
  - Wednesday 2:30pm — 3:30pm
  - Room: D107

TVVS 2022/2023

- Grading system/formula: 50% project + 50% final exam

To successfully complete this course

  - project >= 47.5% and exam >= 47.5%

- 1 project composed by ~9 mini assignments (~1 per week)

At the end of this course, you would be able to

- **Plan a Verification and Validation strategy** that includes a selection of different techniques and tools.

- Design and **develop tests at different levels** (i.e., unit, integration, system) using standard and well-adopted tools that could effectively test complex software systems.

- **Derive tests that deal with exceptional and corner cases** by performing several different techniques (e.g., boundary analysis) as well as able to reflect on their limitations, when and when not to apply them in a given context.

- Measure the efficiency of the developed tests by means of **different** test **adequacy metrics** (e.g., line, decision, condition, MC/DC coverage).

- Write **maintainable test** code by avoiding well-known test's issues (e.g., flakiness, unreadable, slow, dependent, fat tests, etc.)

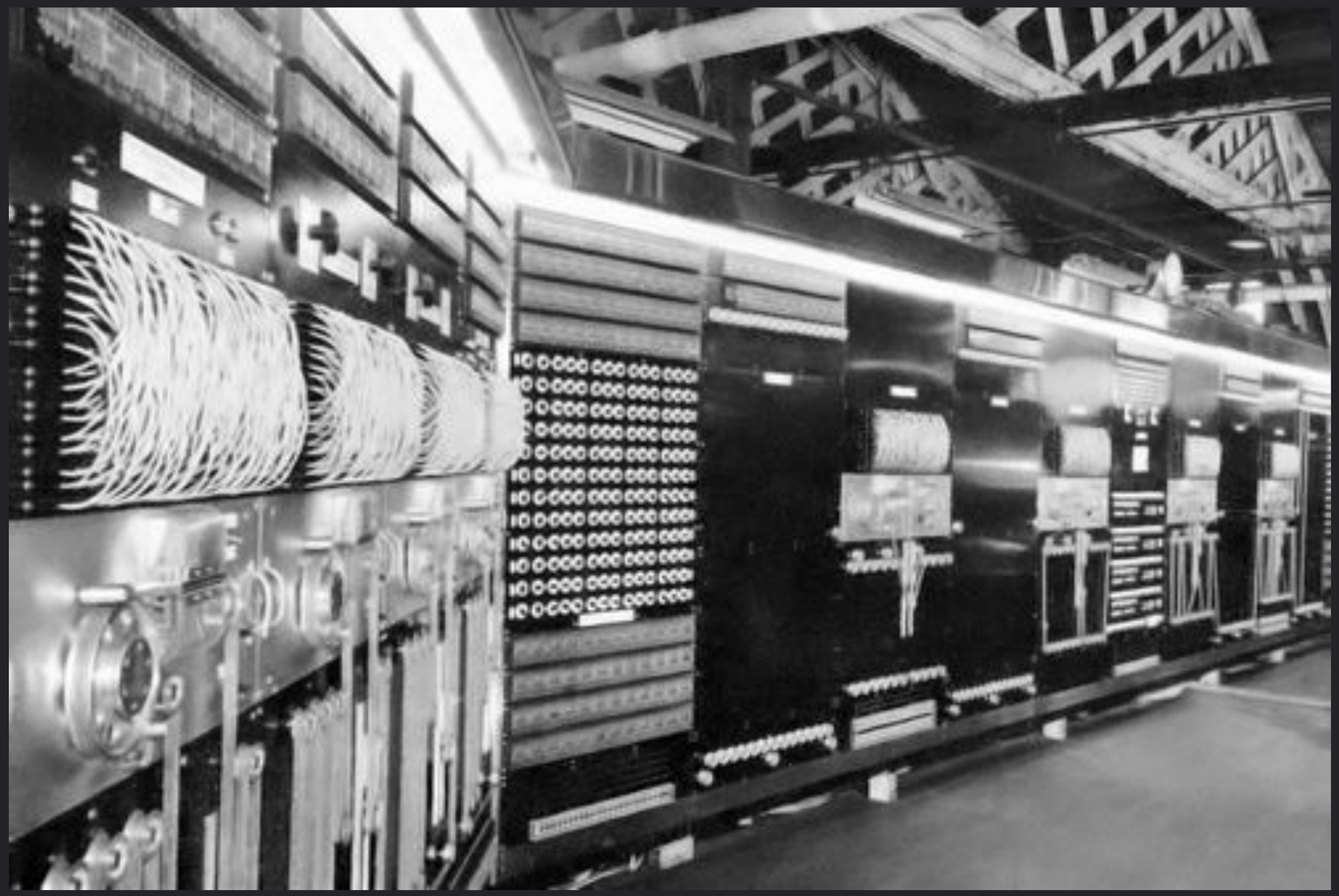
TVVS 2022/2023

(disclaimer)

- Slides based on the following sources:
  - CSE1110, Software Quality and Testing @ TUDelft by Arie van Deursen, Maurício Aniche, Frank Mulder
  - Software Verification and Validation @ FCUL < 2020/2021 by Eduardo Marques, Vasco Vasconcelos, Francisco Martins, and João Neto
  - Software Testing, Verification and Validation @ FEUP by Ana Paiva

Why do we need to verify and validate our software?

Because software *bugs* are everywhere!
Literally!



History [edit]

Main article: Bug (engineering)

The Middle English word *bugge* is the basis for the terms "bugbear" and "bugaboo" as terms used for a monster.[6]

The term "bug" to describe defects has been a part of engineering jargon since the 1870s[7] and predates electronics and computers; it may have originally been used in hardware engineering to describe mechanical malfunctions. For instance, Thomas Edison wrote in a letter to an associate in 1878:[8]

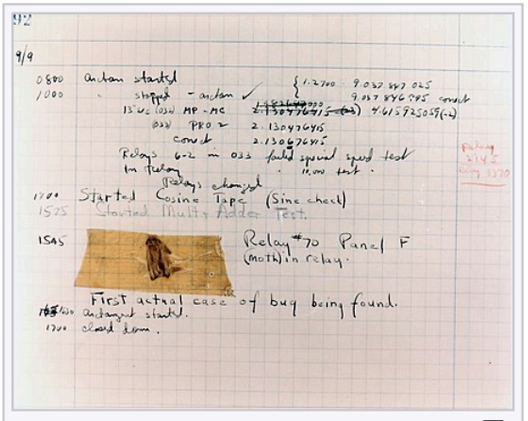
> ... difficulties arise—this thing gives out and [it is] then that "Bugs"—as such little faults and difficulties are called—show themselves[9]

Baffle Ball, the first mechanical pinball game, was advertised as being "free of bugs" in 1931.[10] Problems with military gear during World War II were referred to as bugs (or glitches).[11] In a book published in 1942, Louise Dickinson Rich, speaking of a powered ice cutting machine, said, "Ice sawing was suspended until the creator could be brought in to take the bugs out of his darling."[12]

Isaac Asimov used the term "bug" to relate to issues with a robot in his short story "Catch That Rabbit", published in 1944.

The term "bug" was used in an account by computer pioneer Grace Hopper, who publicized the cause of a malfunction in an early electromechanical computer.[13] A typical version of the story is:

> In 1946, when Hopper was released from active duty, she joined the Harvard Faculty at the Computation Laboratory where she continued her work on the Mark II and Mark III. Operators traced an error in the Mark II to a moth trapped in a relay, coining the term *bug*. This bug was carefully removed and taped to the log book. Stemming from the first bug, today we call errors or glitches in a program a *bug*.[14]



A page from the Harvard Mark II electromechanical computer's log, featuring a dead moth that was removed from the device.

Hopper was not present when the bug was found, but it became one of her favorite stories.[15] The date in the log book was September 9, 1947.[16][17][18] The operators who found it, including William "Bill" Burke, later of the Naval Weapons Laboratory, Dahlgren, Virginia,[19] were familiar with the engineering term and amusedly kept the insect with the notation "First actual case of bug being found." This log book, complete with attached moth, is part of the collection of the Smithsonian National Museum of American History.[17]

The related term "debug" also appears to predate its usage in computing: the *Oxford English Dictionary*'s etymology of the word contains an attestation from 1945, in the context of aircraft engines.[20]

The concept that software might contain errors dates back to Ada Lovelace's 1843 notes on the analytical engine, in which she speaks of the possibility of program "cards" for Charles Babbage's analytical engine being erroneous:

> ... an analysing process must equally have been performed in order to furnish the Analytical Engine with the necessary *operative* data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the *cards* may give it wrong orders.

Software *bugs* cost millions

ROCKET LAUNCH ERRORS

In 1996, a European Ariane 5 rocket was set to deliver a payload of satellites into Earth orbit, but problems with the software caused the launch rocket to veer off its path a mere 37 seconds after launch. As it started disintegrating, it self-destructed (a security measure). The problem was the result of code reuse from the launch system's predecessor, Ariane 4, which had very different flight conditions from Ariane 5. More than $370 million were lost due to this error.

Software *bugs* “kill” people

Boeing finds another software problem on the 737 Max

killed 346 people

It’s at least the third flaw found since the plane was grounded last year

By Sean O'Kane | @sokane1 | Feb 6, 2020, 12:42pm EST

https://www.theverge.com/2020/2/6/21126364/boeing-737-max-software-glitch-flaw-problem
https://www.youtube.com/watch?v=H2tuKiiznsY

Software *bugs* expose people's data

LILY HAY NEWMAN SECURITY 12.10.2018 02:19 PM

A New Google+ Blunder Exposed Data From 52.5 Million Users

A month after Google had already decided to shut down Google+, a new bug made its problems much, much worse.

https://www.wired.com/story/google-plus-bug-52-million-users-data-exposed/

iOS 14.4
Apple Inc.
Downloaded

iOS 14.4 includes improvements and bug fixes for your iPhone.

For information on the security content of Apple software updates, please visit this website:
https://support.apple.com/kb/HT201222

Twitter
3 Days Ago

UPDATE

We made improvements and squashed bugs so Twitter is even better for you.

Version 8.52.1 • 115.8 MB

Facebook
2 Days Ago

UPDATE

We've updated the app to fix some crashes and make features load faster.

Version 304.1 • 228.2 MB

Instagram
Feb 1, 2021

UPDATE

The latest version contains bug fixes and performance improvements.

Version 173.0 • 141.6 MB

WhatsApp Messenger
Jan 28, 2021

UPDATE

Bug fixes.

Version 2.21.20 • 78.7 MB

Messenger
Feb 1, 2021

UPDATE

We update the app regularly so that we can make it better for you. Get the latest version for all of the available Messenger features. This version includes several bug fixes and performance improvements. Thanks for using Messenger!

Version 298.0 • 66.7 MB

Twitch: Live Game Streaming
Feb 1, 2021

OPEN

Bug fixes, stability fixes, and app optimizations

Version 10.0 • 147.3 MB

Skype for iPhone
Jan 29, 2021

OPEN

We're listening to your feedback and working hard to improve Skype. Here's what's new:
- Now you can have big conversations, because Skype calls now support up to 100 participants for audio group calls
- Bug fixes and stability improvements

Visit https://go.skype.com/whatsnew for more details.

Version 8.68 • 106.7 MB

Spotify: Music and podcasts
Jan 29, 2021

OPEN

We're always making changes and improvements to Spotify. To make sure you don't miss a thing, just keep your Updates turned on.

Bug fixes and improvements in this version include:
- Fixed stability and performance issues

Version 8.5.96 • 128.7 MB

Signal - Private Messenger
Feb 2, 2021

UPDATE

• We fixed a few bugs that were reported to us by the beta community and people who are really fast at downloading new things.

Interested in helping improve Signal? We're hiring. Visit https://signal.org/workworkwork to learn more.

Version 5.3.2 • 98.6 MB

Shazam: Music Discovery
Jan 27, 2021

OPEN

Bug fixes and performance improvements.

Love the app? Rate us! Your feedback is music to our ears, and it helps us make Shazam even better. Got a question? Visit support.apple.com

Version 14.3 • 20.4 MB

The term “*bug*” is ambiguous … are we referring to the source code or to outcome of a failed execution?

Failure, Fault, and Error

(falha, falta, erro)

It is common to hear different terms to indicate that a software system is not behaving as expected. Just to name a few: *error*, *mistake*, *defect*, *bug*, *fault*, and *failure*. To describe the events that led to a software crash more precisely, we need to agree on a certain vocabulary. For now, this comes down to three terms: **failure**, **fault**, and **error**.

Failure, Fault, and Error

(falha, falta, erro**)**

A **failure** is a component of the (software) system that is not behaving as expected. Failures are often visible to the end user. An example of a failure is a mobile app that stops working, or a news website that starts to show yesterday's news on its front page. The software system did something it was not supposed to do.

Failure, **Fault**, and Error

(falha, **falta**, erro**)**

Failures are generally caused by *faults*. **Faults** are also called *defects* or *bugs*. A fault is the flaw in the component of the system that caused the system to behave incorrectly. A fault is technical and, in our world, usually refers to source code, such as a comparison in an `if` statement that uses a "less than" operator instead of a "greater than" operator.

Failure, **Fault**, and Error

(falha, **falta**, erro**)**

Failures are generally caused by *faults*. **Faults** are also called *defects* or *bugs*. A fault is the flaw in the component of the system that caused the system to behave incorrectly. A fault is technical and, in our world, usually refers to source code, such as a comparison in an `if` statement that uses a "less than" operator instead of a "greater than" operator.

Note that the existence of a fault in the source code does not necessarily lead to a failure. If the source code containing the fault is never executed, it will never cause a failure. Failures only occur when the system is being used and someone notices it not behaving as expected.

Failure, Fault, and **Error**

(falha, falta, erro)

Finally, we have an **error**, also called *mistake*. An error is the human action that caused the system to run not as expected. For example, a developer didn't cover an obscure condition because they misunderstood the requirement. Plugging a cable into the wrong socket is an example of a hardware mistake.

In other words: an *error* by a developer can lead to a *fault* in the source code that will eventually result in a *failure*.

In the code example above: the *failure* was the program returning a large number, the *fault* was a bad if/else if condition, and the *error* was not dealing properly with that case.

Verification & Validation

Verification & Validation

Verification and **validation** are both about assessing the quality of a system. However, they do have a subtle difference, which can be quickly described by a single question:

Verification: "Have we built the software right?"
"Desenvolvemos o software corretamente?"

Validation: "Have we built the right software?"
"Desenvolvemos o software correto?"

Note how both *verification* and *validation* are fundamental for ensuring the delivery of high-quality software systems.

Verification & Validation

Verification: Have we built the software right? (i.e., does it implement the requirements)

Verification, is about the system behaving as it is supposed to, according to the specification.

In simple words, this mostly means that the system behaves without any *bugs*. Note that this does not guarantee that the system is useful: a system might work beautifully, *bug*-free, but not deliver the features that customers really need.

Verification & **Validation**

Validation: Have we built the right software? (i.e., do the deliverables satisfy the customer)

Validation concerns the features that our software system offers, and the customer (i.e., for whom the system is made):

- is the system under development what the users really want and/or need?
- is the system actually useful?

Validation techniques, thus, focus on understanding whether the software system is delivering the business value it should deliver.

Verification & Validation

Verification is often assessed through static analysis, aka static testing, (e.g., code review, editors' check source code, compilers check syntax, etc.)

Validation typically involves dynamic analysis, aka dynamic testing, i.e., when the software itself is executed.

What are the benefits of VV?

- **Cost-Effective:** It is one of the important advantages of software testing. Testing any IT project on time helps you to save your money for the long term. In case if the bugs caught in the earlier stage of software testing, it costs less to fix.
- **Security:** It is the most vulnerable and sensitive benefit of software testing. People are looking for trusted products. It helps in removing risks and problems earlier.
- **Product quality:** It is an essential requirement of any software product. Testing ensures a quality product is delivered to customers.
- **Customer Satisfaction:** The main aim of any product is to give satisfaction to their customers. UI/UX Testing ensures the best user experience.

So, why do not we just test more and more?

So, why do not we just test more and more?

TL;DR: developing tests is a very time-consuming task which is subject to incompleteness and further errors.

Why is software testing so hard?

- **Bugs are not uniformly distributed**, some components in some software systems present more bugs than other components.
- **How many tests are enough?** Creating too many tests, without proper consideration, might lead to ineffective tests (besides costing too much time and money). Creating too many tests, without proper consideration, might lead to ineffective tests (besides costing too much time and money).
- **When should I stop testing?** Exhaustive testing is impossible. Imagine a software system that has “just” 300 different flags (or configuration settings). Those flags can be set to either true or false and they can be set independently from the others. The software system behaves differently according to the configured combination of flags. This implies that we need to test all the possible combinations. A simple calculation shows us that 2 possible values for each of the 300 different flags gives 2^300 combinations (2037035976334486086268445688409378161051468393665936250636140449354381299763336706183397376) that need to be tested. As a matter of comparison, this number is higher than the estimated number of atoms in the universe. In other words, this software system has more possible combinations to be tested than the universe has atoms!

Why is software testing so hard?

- As Dijkstra used to say, **program testing can be used to show the presence of bugs, but never to show their absence**. In other words, while we might find more bugs by simply testing more, they will never ensure that the software system is 100% *bug*-free. They will only ensure that the cases we test for behave as expected.
- To test a piece of software, we need a lot of **variation in our tests**. For example, we want variety in the inputs when testing a method, like we saw in the example above. To test the software well, however, we also need variation in the testing strategies that we apply.
- The **context** also plays an important role in how one devises tests. For example, devising tests for a mobile app is very different from devising tests for a web application, or video game. In other words: testing is context-dependent.

Static Verification

Static Verification

Static Testing is a software testing technique which is used to check *faults* in a software application without executing its source code. It is concerned with the analysis of the static system representation (source code, documents, models, prototypes, etc.) to discover *faults*.

- Early detection of faults prior to test execution.
- Early warning about suspicious aspects of the code or design.
- Detecting dependencies and inconsistencies in software models, such as links.
- Improved maintainability of code and design.
- Prevention of faults, if lessons are learned in development.

Static Verification

The two main types of static testing techniques are:

- **Manual examinations**: Manual examinations include analysis of code done manually, also known as **Reviews.**
- **Automated analysis using tools:** Automated analysis are basically static analysis which is done using tools.

Reviews

A **review** in static testing is a process or meeting conducted to find causes of failures, i.e., faults in the program, rather than the failures themselves. By reviewing the program all team members get to know about the progress of the project and sometimes the diversity of thoughts may result in excellent suggestions.

Cost of Reviews

- 5% to 15% of development effort. Note: testing and debugging represents 50% of development effort.

But it is worth it:

- Reduce faults by a factor of 10. [Yourdon, Structured Walkthroughs]
- 10 times reduction in fault reaching test, testing cost reduced by 50% to 80%. [Freedman & Weinberg, Handbook of Walkthroughs, Inspections & Technical Reviews]

What can be reviewed?

Anything written down can be inspected:

- Policy, strategy, business plans, marketing or advertising materials, contracts
- System requirements, feasibility studies, acceptance test plans
- Test plans, test designs, test cases, test results
- System designs, logical & physical
- Source code
- User manuals, procedures, training material

Types of Reviews

A single artifact may be the subject to more than one review. Reviews can be classified into four parts:

1. Walkthroughs (author in the lead)
2. Technical review (technical meeting to achieve consensus)
3. Inspections (peer review of documents, relies on ‘visual inspection’, aka reading)

1. Walkthroughs

Author guide the participants through the document according to his or her thought process to achieve a common understanding and to gather feedback.

- It is not a formal process.
- It is led by the authors.
- Useful for the people if they are not from the software discipline, who are not used to or cannot easily understand software development process.
- Is especially useful for higher level documents like requirement specification, etc.

Main goals:

- To present the documents both within and outside the software discipline in order to gather the information regarding the topic under documentation.
- To explain or do the knowledge transfer and evaluate the contents of the document.
- To achieve a common understanding and to gather feedback.
- To examine and discuss the validity of the proposed solutions.

2. Technical Review

Includes peer and technical experts, no management participation. Can be rather subjective.

- It is less formal review.
- It is led by the trained moderator but can also be led by a technical expert.
- It is often performed as a peer review without management participation.
- Faults are found by the experts (such as architects, designers, key users) who focus on the content of the document.
- In practice, technical reviews vary from quite informal to very formal.

Main goals:

- To ensure that an early stage the technical concepts are used correctly.
- To access the value of technical concepts and alternatives in the product.
- To have consistency in the use and representation of technical concepts.
- To inform participants about the technical content of the document.

3. Inspection

- It is the most formal review type.
- It is led by the trained moderators.
- The artifacts to be inspected and given out in advance.
- It involves peers to examine the product.
- The faults found are documented in a logging list or issue log.
- A formal follow-up is carried out by the moderator.

Main goals:

- It helps the author to improve the quality of the document under inspection.
- It removes faults efficiently and as early as possible.
- It improve product quality.
- It create common understanding by exchanging information.
- It learn from faults found and prevent the occurrence of similar faults.

Code Review at Google

> At Google, each code change is reviewed. Period.

> At Google, code review are on average completed within 4 hours.

by Michaela Greiler https://www.michaelagreiler.com/code-reviews-at-google/

Code Review at Google

1. It all starts after Mark has made some changes to the code and wants those code changes to be merged with the shared codebase.
2. Before sending the code out for review Mark needs to run the code through a static analysis tool, i.e., Tricorder. Mark reviews the results of the static analysis tool and when he is happy with his changes, he sends the changes to at least one code reviewer.
3. The code reviewer carefully looks through the code and leaves comments if she sees a problem or needs some clarification. Mark then addresses each comment either by changing the code or replying to the comment. If Mark made some changes to the code under review, he uploads the new version for reviewers to check again. If a reviewer is satisfied, she can approve the change by marking it as “LGTM” (looks good to me). To be able to commit the code to the shared codebase, at least one reviewer must approve the code.

Code Review at Google

- 75% of the code reviews have just one reviewer.
- 90% of the code reviews have fewer than 10 files changed. Most of the changes also have only around 24 lines of code changed.

5 REASONS GOOGLERS REVIEW CODE

and everyone?

Education

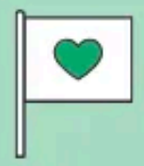
Mentoring, learning, knowledge disemination.

Accident prevention

Find bugs and defects, ensure high quality code

Gatekeeping

Prevent arbitrary code to be committed, security



Tracing & tracking

Understanding evolution and why and how code changed

Readable Code

Maintaining norms, consistent style and design, and having adequate tests

Summary

- Static testing is to find faults as early as possible.
- Static testing not a substitute for dynamic testing (will talk about that in the coming weeks), both find a different types of faults.
- Reviews are an effective technique for static testing.
- Reviews not only help to find faults but also understand missing requirements, design faults, non-maintainable code.

References

- Paul Ammann, Jeff Offutt; Introduction to Software Testing, 2nd Edition, 2016. ISBN: 978-1-107-17201-2
- Paul C. Jorgensen; Software Testing A Craftsman's Approach, 4th Edition, 2013. ISBN: 978-1-466-56069-7
- Dorothy Graham, Rex Black, Erik van Veenendaal; Foundations of Software Testing: ISTQB Certification, 4th Edition, 2020. ISBN: 978-1-473-76479-8
- Ilene Burnstein; Practical Software Testing, 2003. ISBN: 978-0-387-95131-7
- Gordon Fraser and José Miguel Rojas; Software Testing, 2019. ISBN 978-3-030-00262-6
- Mark Utting; Practical Model-Based Testing, 2007. ISBN: 978-0-12-372501-1
- Tomek Kaczanowski; Bad Tests, Good Tests, 2013. ISBN: 978-8-393-84713-6
- Chak Shun Yu, Christoph Treude, Maurício Aniche; Comprehending Test Code: An Empirical Study, 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). ISBN:978-1-7281-3095-8
- Michal Young and Mauro Pezzè; Software testing and analysis: process, principles, and techniques, 2008. ISBN: 978-0-471-45593-6

References

- D. Graham, R. Black, and E. van Veenendaal. “Foundations of Software Testing: ISTQB Certification”, 4th Edition, 2020.
- E. Bouwers, J. Visser, and A. van Deursen. “Getting what you Measure”. CACM, May 2012
- A. Bacchelli and C. Bird. “Expectations, Outcomes, and Challenges of Modern Code Review”. ICSE 2013.
- Alex Nederlof. “The truth about code reviews”. 2013.
- Michaela Greiler’s blog on Code Review, https://www.michaelagreiler.com/code-review-blog-post-series