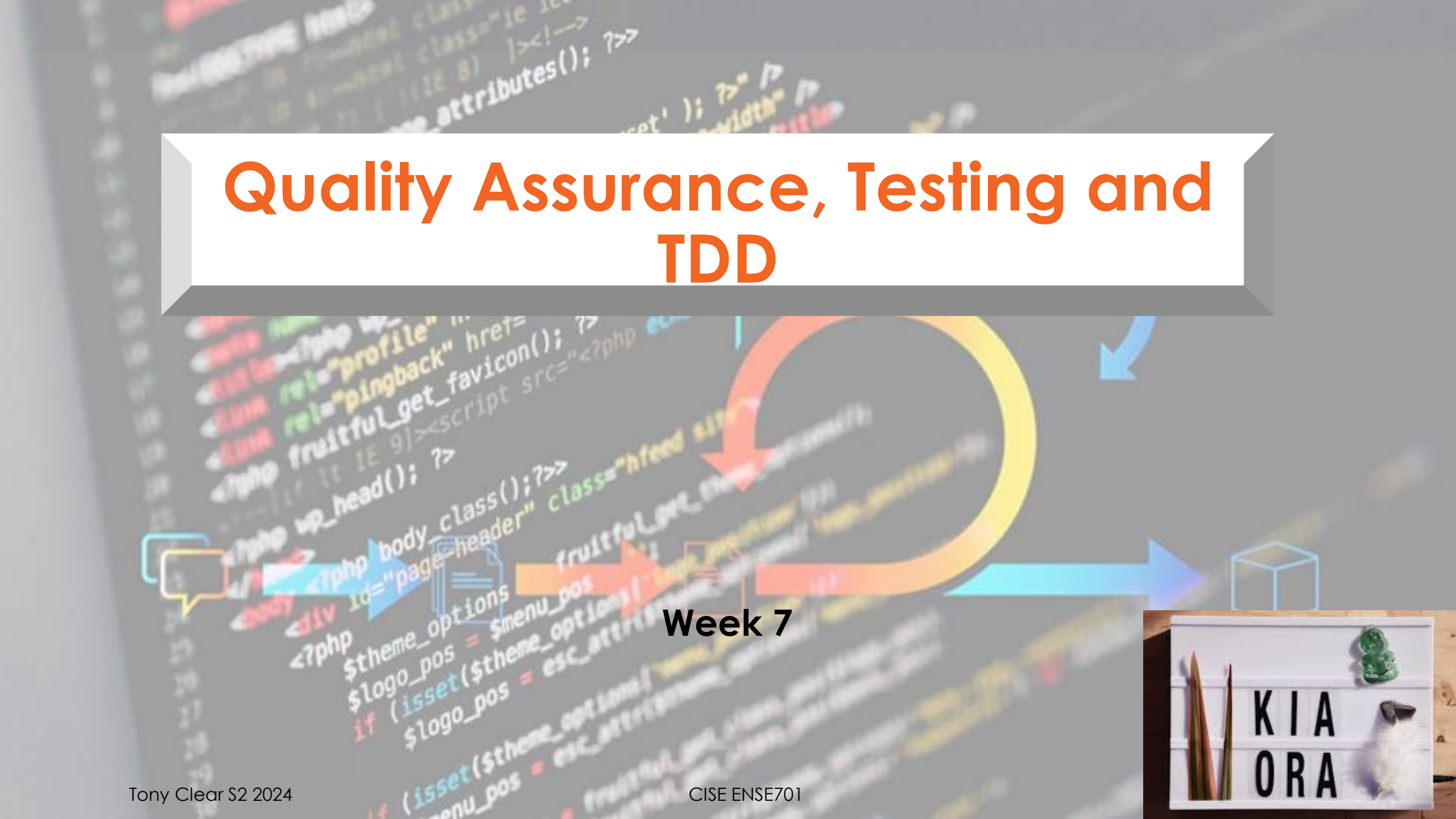
Quality Assurance, Testing and TDD



Week 7

Taking Stock

The schedule for the course

Where are we now?

The Assessment Schedule

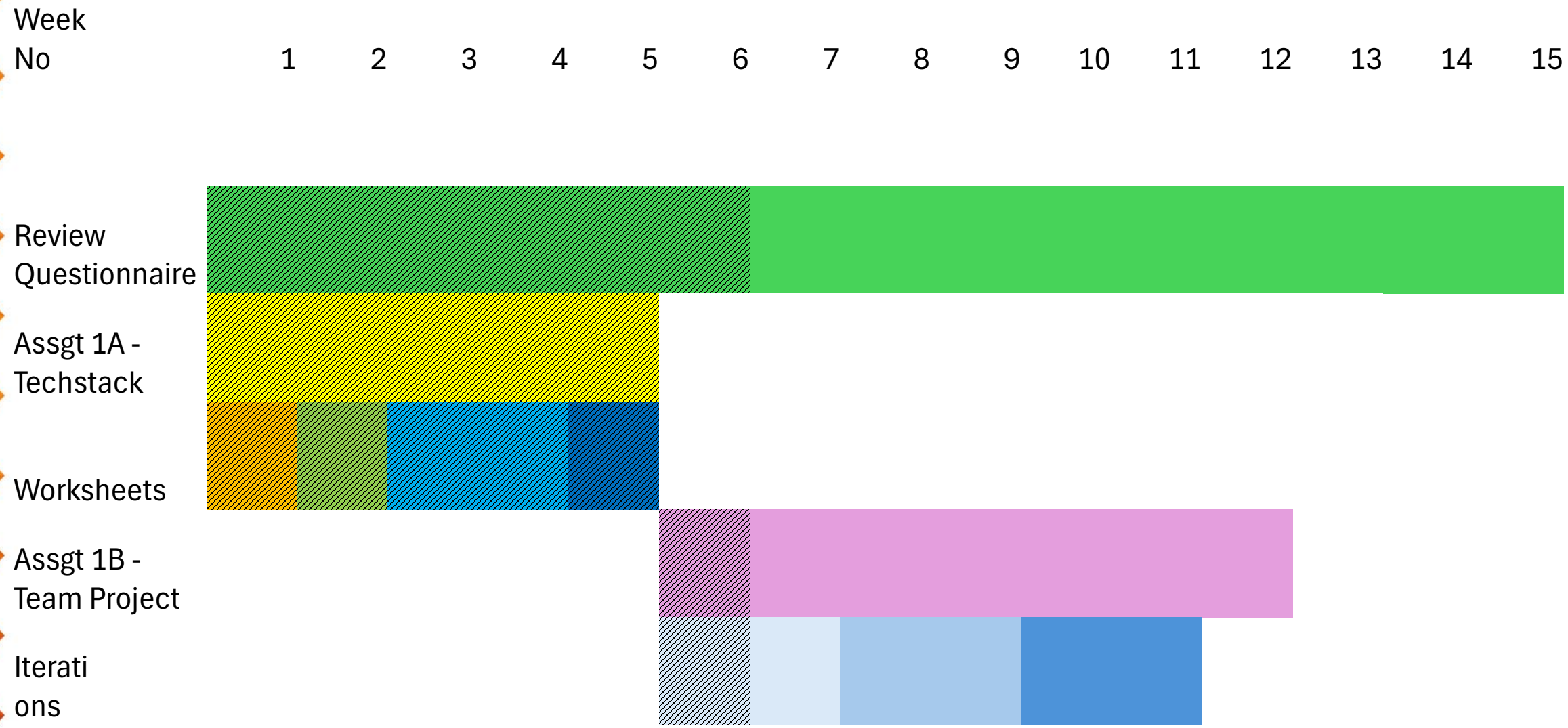
Progress – feedback, any issues?

Overview - what's coming up?

The Lecture Schedule

How does it relate to the assessment?

Taking Stock



Assignments Drive your Learning

Ass 1A preparing for Software Development (20%)
(*Individual*)

- Set up the tools an individual needs to support coding, good code craft, version control and unit testing
- Set up the tools needed to collaborate with a team to achieve product goals together

Sharing code – integrate code, review code,
Setup the tools needed to work with the selected Tech Stack (front-end/backend)
Set up tools to assure quality of product
Set up tools to deploy the product to the cloud
Set up tools to monitor and alert issues post deploy
Learn how to use the tools
Learn how to use the Tech Stack
Understand the product goals -> Product Backlog
Sprint 1 Goals -> Sprint Backlog

Submission in Tutorials weeks 1-5 (sign off by TA)
Evidence portfolio and demo

Ass1B Full SDLC full stack product Dev (50%)
(*small team - 4 Including QA*)
Capability building by Developing a Product in a small team
Apply a new tech stack and tool set
Practice DevOps and Scrum WoW
Collaborate with a Product Owner and team
Three sprints to learn fast – fast feedback
Submit – reviews weeks 8,10,12 (tutorials)
Capability and learning Portfolio with Evidence
Product increments

Sprint 1 weeks 6 and 7
Sprint 2 weeks 8 and 9
Sprint 3 weeks 10 and 11

Ass 2 Knowledge Check (30%)
(*Individual, online questions*)
A set of questions about scenarios to confirm you have understood main language and principles
Sometime in Revision weeks (Faculty schedules)

Team Contract and Commitments

Additions:

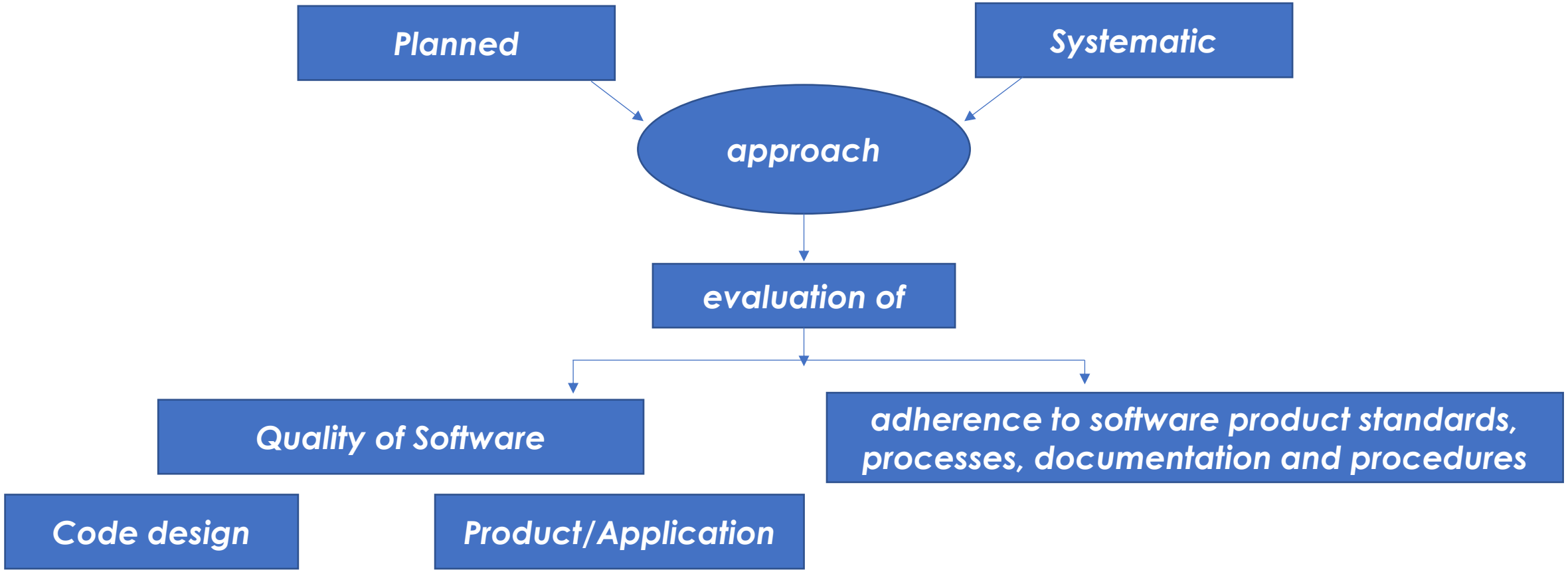
Revision History

Date	Version	Description	Author
<dd/mm/yyyy>	*<x.x>*	*<details>*	*<name>*

Team Issue Report *(date and revision version)*:

- Explanation of Adverse situation
- Strategy for making up for deficiencies to the team such as: skills lack or lack of team contribution
- Strategy for demonstrating team commitment
- Author and signature

What is Quality Assurance?



Key Aspects of Quality?

…software quality has been termed “the elusive target” [9], which can be viewed from five different perspectives:

- The *transcendental view* sees quality as something that can be recognized but not defined.
- The *user view* sees quality as fitness for purpose.
- The *manufacturing view* sees quality as conformance to specification.
- The *product view* sees quality as tied to inherent characteristics of the product.
- The *value-based view* sees quality as dependent on the amount a customer is willing to pay for it.

Clear, T. (2011, June). THINKING ISSUES: A 'potted guide' to quality assurance for computing capstone projects. *ACM Inroads, 2*(2), 14-15. https://doi.org/DOI: 10.1145/1963533.1963536

Key Aspects of Quality & Quality Assurance - Process ?

Quality in Learning

there are differing models for educational quality. The one I prefer is that …whereby learning … as a *'transformative process'* for the student – like the 'transcendental view' of quality.

Differing ways of producing quality outcomes depending upon the goals

the need for a ***quality process*** to ensure predictable and high quality outcomes. This is where selection of a methodology to fit the needs of the project is necessary.

Clear, T. (2011, June). THINKING ISSUES: A 'potted guide' to quality assurance for computing capstone projects. *ACM Inroads, 2*(2), 14-15. https://doi.org/DOI: 10.1145/1963533.1963536

Key Aspects of Quality – Roles & Measurement ?

A further important element of a *quality process* - the identification and ***allocation of roles and responsibilities*** to appropriately skilled team members

an alternative to process, **measurement** is a classic approach to quality, whether that addresses process or product dimensions.

For instance, the ISO9126 ***standard*** specifies a set of software quality attributes, including: functionality; reliability; efficiency; usability; maintainability and portability

Clear, T. (2011, June). THINKING ISSUES: A 'potted guide' to quality assurance for computing capstone projects. *ACM Inroads, 2*(2), 14-15. https://doi.org/DOI: 10.1145/1963533.1963536

Key Aspects of Quality –Metrics and Testing?

A set of ***metrics*** accompany standards,

- a yardstick by which conformance to the standard can be demonstrated or a lack of conformance can be highlighted.
- For students, **standards for coding and document formatting**, **or for recording meeting minutes** may be relevant examples, where **compliance** with the quality standard **can be objectively demonstrated.**

testing, while part of QA - more properly classified as a ***quality control*** activity (QC) rather than QA…. it is inherently **part of the production function, but as a control check added on at the end**.

- BUT a **well framed and multi layered testing strategy** (including unit tests, integration tests, usability tests, performance and stress tests, acceptance tests etc.) is **a key element supporting a QA framework** for systems related projects.

Clear, T. (2011, June). THINKING ISSUES: A 'potted guide' to quality assurance for computing capstone projects. *ACM Inroads, 2*(2), 14-15. https://doi.org/DOI: 10.1145/1963533.1963536

Key Aspects of Quality - Reviews?

more in-line activities of ***quality review*** have much to offer.

For instance Robert Glass when asked for the three best software engineering practices came up with *"inspections, inspections, inspections"* arguing that they "do a better job of error-removal than any competing technology"

Reviews in turn can be *periodic* and *formalised* through mechanisms such as "a walkthrough and a formal technical review" [3], "design and code inspections" [7] or other forms of audit.

Clear, T. (2011, June). THINKING ISSUES: A 'potted guide' to quality assurance for computing capstone projects. *ACM Inroads, 2*(2), 14-15. https://doi.org/DOI: 10.1145/1963533.1963536

Key Aspects of Quality - Continuous Processes?

more ***continuous processes*** such as **pair programming**, or the shared workshop models e.g. (JAD) [4].

Test Driven Development (TDD) [13], in which design of tests leads development work, can also be thought of as a ***continuous review process***, whereby **quality is "built in" from the outset**.

specific *practices* may be applied e.g. ongoing practices of **continuous integration**, **regular (e.g. daily) code builds**, **refactoring** etc.

or more control oriented practices such as **change control**, **configuration and version management [2].**

Clear, T. (2011, June). THINKING ISSUES: A 'potted guide' to quality assurance for computing capstone projects. *ACM Inroads, 2*(2), 14-15. https://doi.org/DOI: 10.1145/1963533.1963536

Key Aspects of Quality - Continuous Improvement?

Finally **at a meta-level** - the notion of ***continual process improvement***, or **software process and practice improvement (SPPI)**,

- "aims to build an infrastructure and culture that support effective methods, practices, and procedures and integrate into the ongoing way of doing business"

meta-level thinking … as students reflect upon the effectiveness of the processes and practices they have applied in their projects. Ideally, they would **adapt and refine them** as they proceed.

- At a minimum, to **reflect upon the processes and practices they have applied** during their projects and **demonstrate awareness of how they could have done things differently** and what those improvements might look like in future.

Clear, T. (2011, June). THINKING ISSUES: A 'potted guide' to quality assurance for computing capstone projects. *ACM Inroads, 2*(2), 14-15. https://doi.org/DOI: 10.1145/1963533.1963536

Key Aspects of Quality – the QA Accountability?

A document (pdf) of a practical **Team agreement** including

- commitments to one another,
- how the team will communicate,
- team roles/accountabilities (including QA for each iteration) and an explanation of why the roles/accountabilities are needed.

1. A description of the Quality Assurance accountability and how it has been exercised within the team
   a. An explanation of your team's emphasis on QA for that iteration
   b. a reflection on how successful you think it was and why?
   c. a reflection on what you would emphasise or do differently for the next iteration

Evolving Views of Software Quality

Over the years, there has been debate about what constitutes software quality and how it should be measured.

This controversy has caused uncertainty across the software engineering community, affecting levels of commitment to the many potential determinants of quality among developers.

An up-to-date catalogue of software quality views could provide developers with contemporary guidelines and templates

Ndukwe, I. G., Licorish, S. A., Tahir, A., & MacDonell, S. G. (2023). How have views on software quality differed over time? Research and practice viewpoints. *Journal of Systems and Software, 195*, 111524.

Software Quality Models

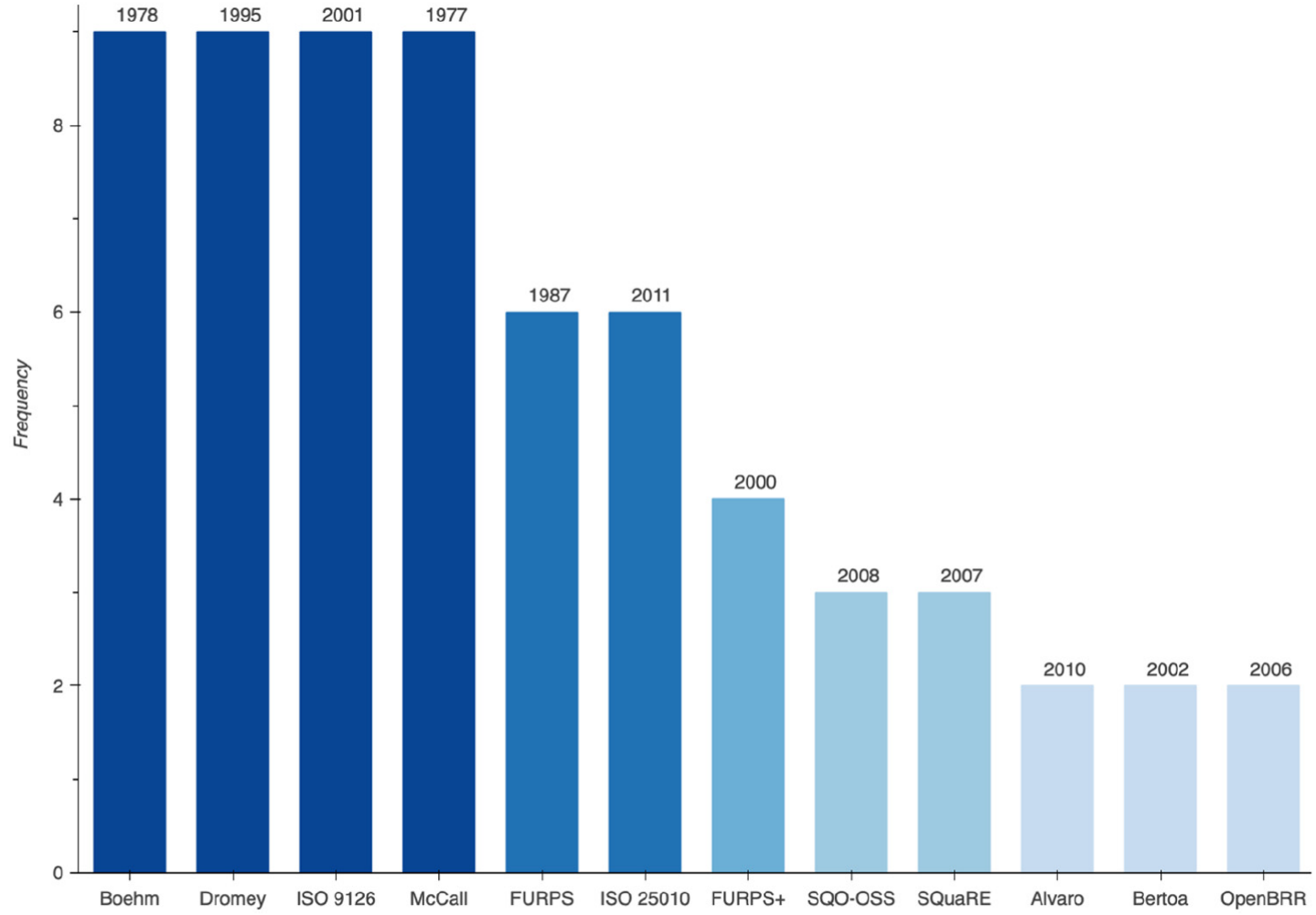


Fig. 1. Frequencies of software quality models with the year each model was proposed.

Ndukwe, I. G., Licorish, S. A., Tahir, A., & MacDonell, S. G. (2023). How have views on software quality differed over time? Research and practice viewpoints. *Journal of Systems and Software, 195*, 111524.

Top 20 Software Quality Characteristics

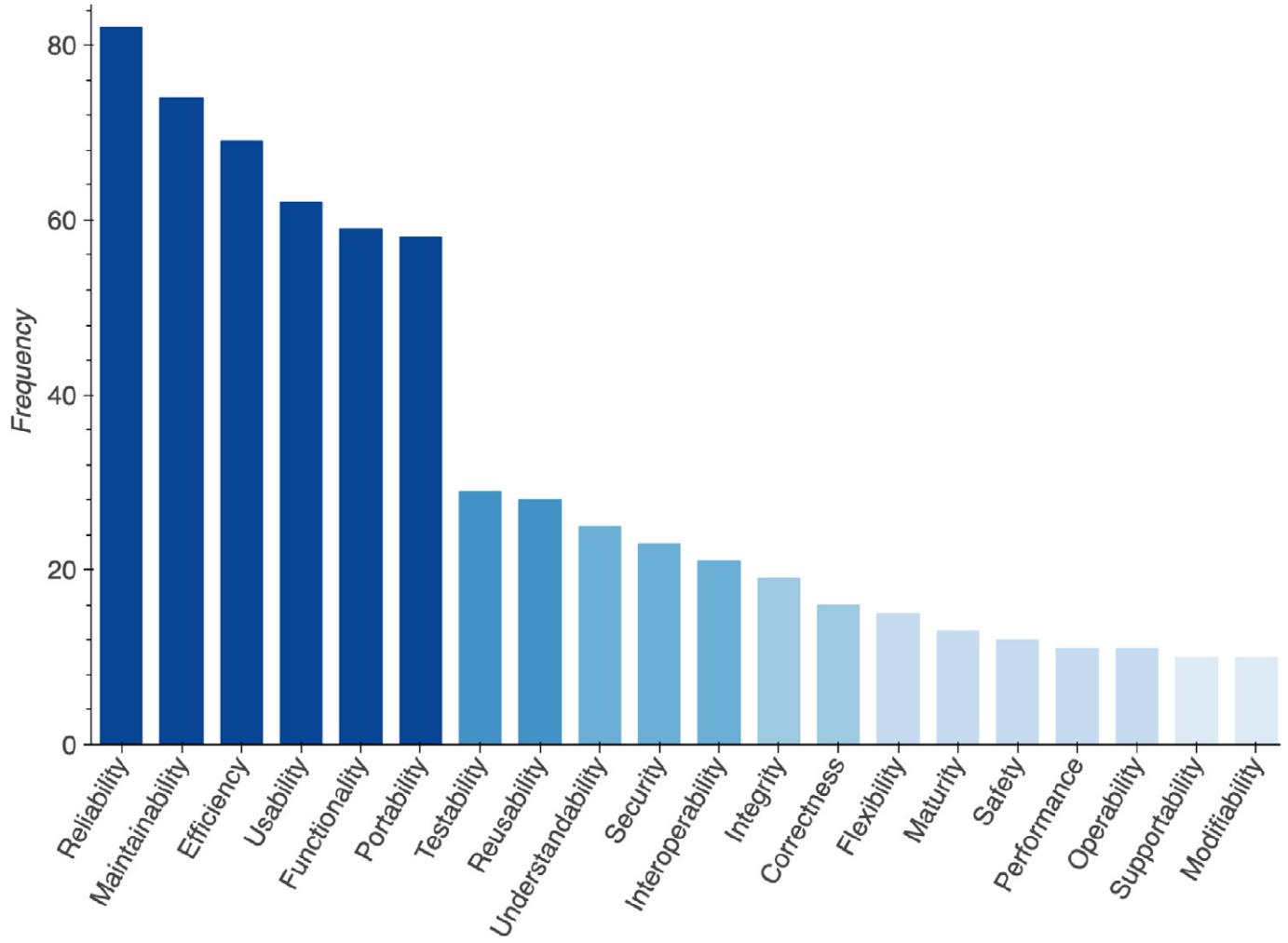


Fig. 2. Frequencies of top twenty software quality characteristics.

Ndukwe, I. G., Licorish, S. A., Tahir, A., & MacDonell, S. G. (2023). How have views on software quality differed over time? Research and practice viewpoints. *Journal of Systems and Software, 195*, 111524.

Questions about quality and testing?

What are the quality criteria to test the quality of code against

What is the quality testing practices to run the tests against these criteria?

How will we know if it passes or fails the quality test?

What should be done if the quality test is failed?

What practices will help to ensure quality test fails do not happen?

Catch low quality

Prevent low quality

How many tests will it take to prove something is correct ?

You can never be 100% sure there are no defects – you can only prove there is a defect!

Then… What is Testing?

Testing is verifying the behaviour of your application is what is specified/expected

Maybe writing code or to do the test or using the application

Code that passes all the tests we can think of is good quality (ready to be released)

Testing needs a set of criteria to test against and the ability to recognize pass or fail

Expected **output** of a function/class/component for a given (type of) **input**(s)

The expected behaviour of an application or system while being used

A given set of coding standards – naming conventions, design principles

A set of standards for UI components and good practice

Good test? "The product should be user friendly"

Why write tests? (4:00)

https://youtu.be/ZmVBCpefQe8

Why Write Tests?

Documentation
Tests are specifications of how our code should work

Consistency
Verify that developers are following good practice and the conventions of our team

Comfort and Confidence
A strong test suite is like a warm blanket – feel ok to release, experiment

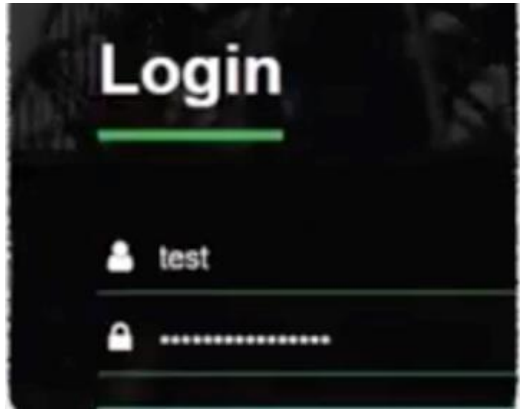
Productivity
We write tests because it allows us to ship code faster

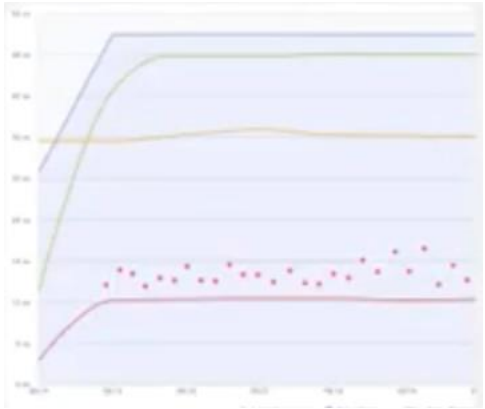
Types of software testing

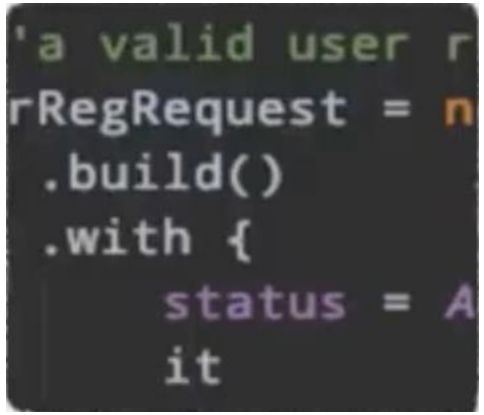
Black Box

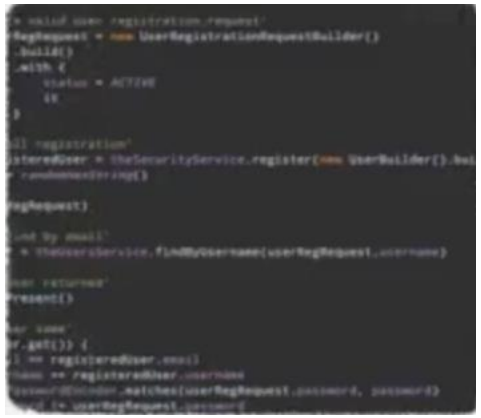
Developers do these

White Box





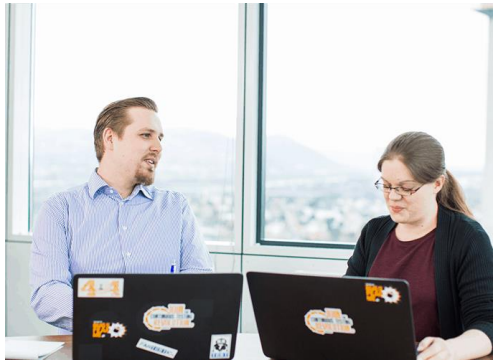




Functional

Regulatory
Compliance

Usability



Exploratory

System

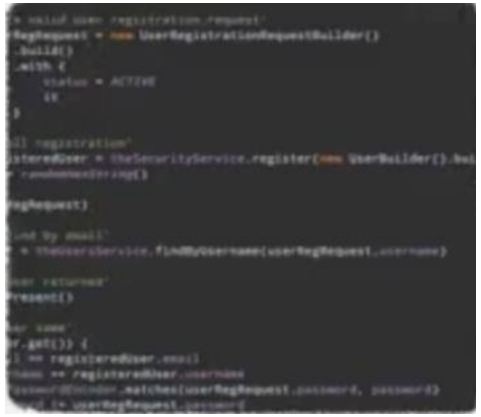
Performance
Stress
Load/stability
Security

Static tests

Linters
SonarQbe
Load/stability
Security

Unit Tests

Integration



Regression

Static tests

Code analysis while you are coding or a large code base

Visual Studio Code squiggly lines
Linters (ESLInt)
Other tools - Sonarqube

The risk of no integration testing

Integration of code developed bit by bit is high risk

DEPENDENCIES

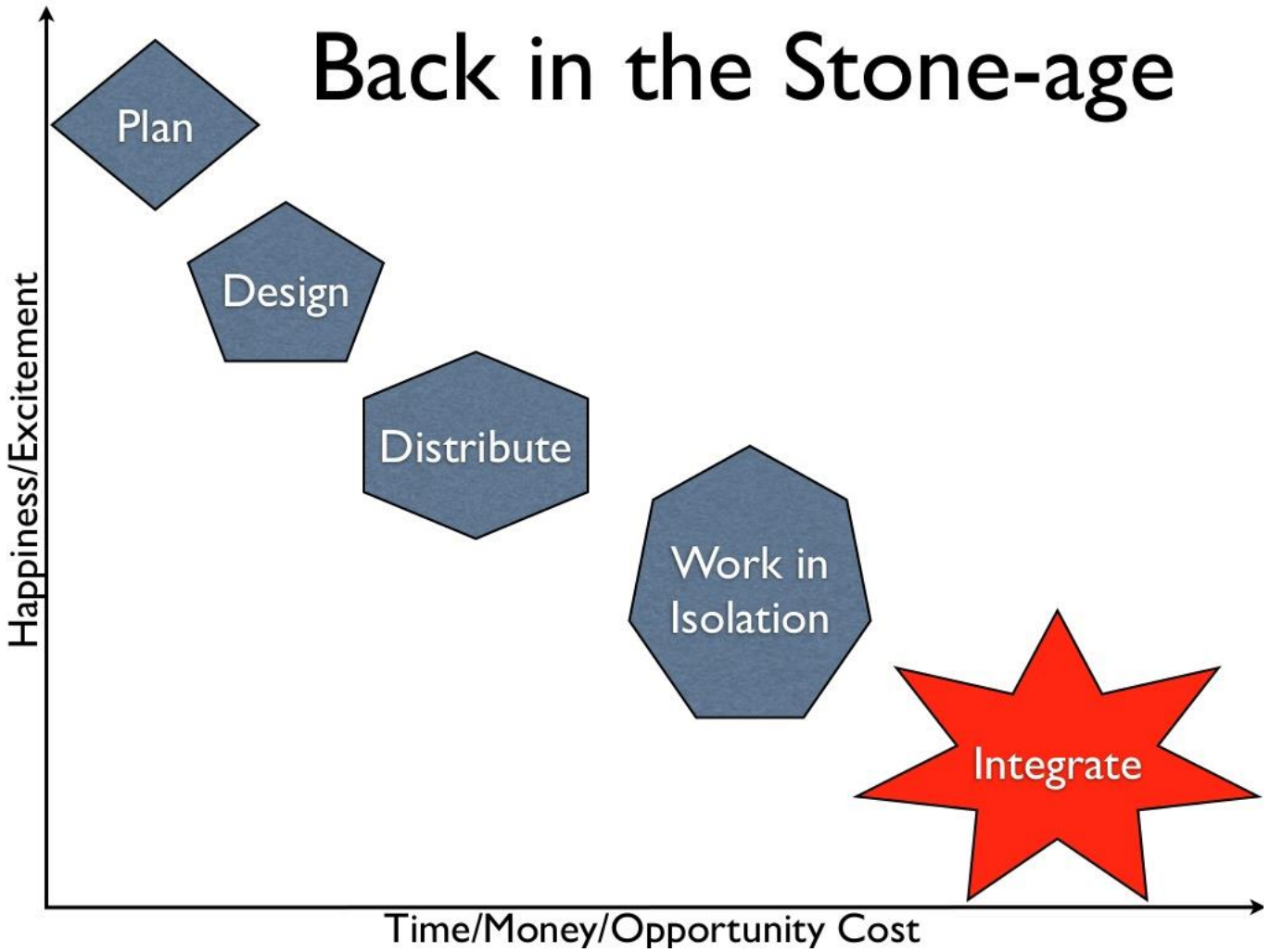
↓

COORDINATION

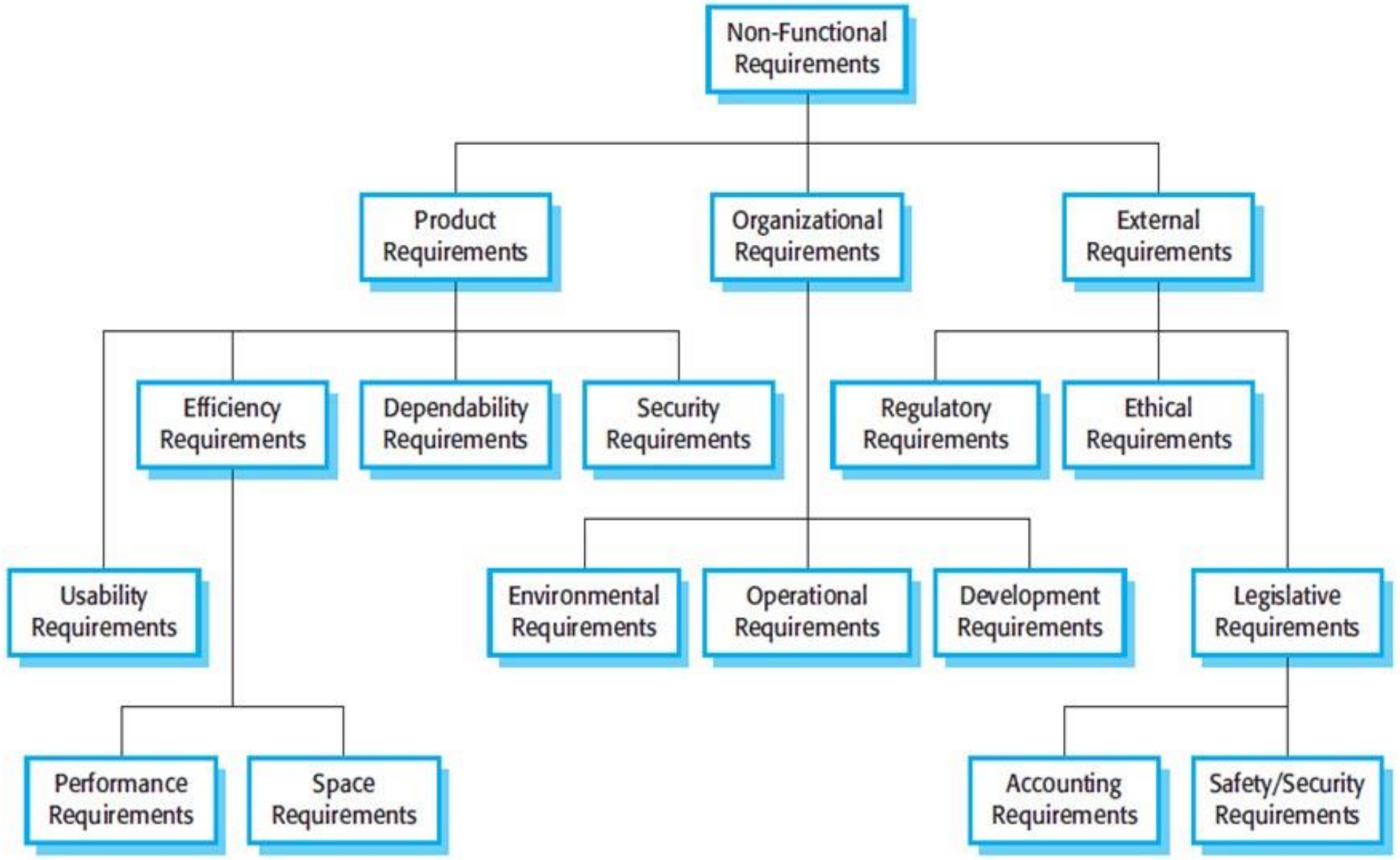
It works on my machine

I didn't realise you were doing it that way

I didn't understand what you were doing



Non-Functional Requirements or Quality Requirements



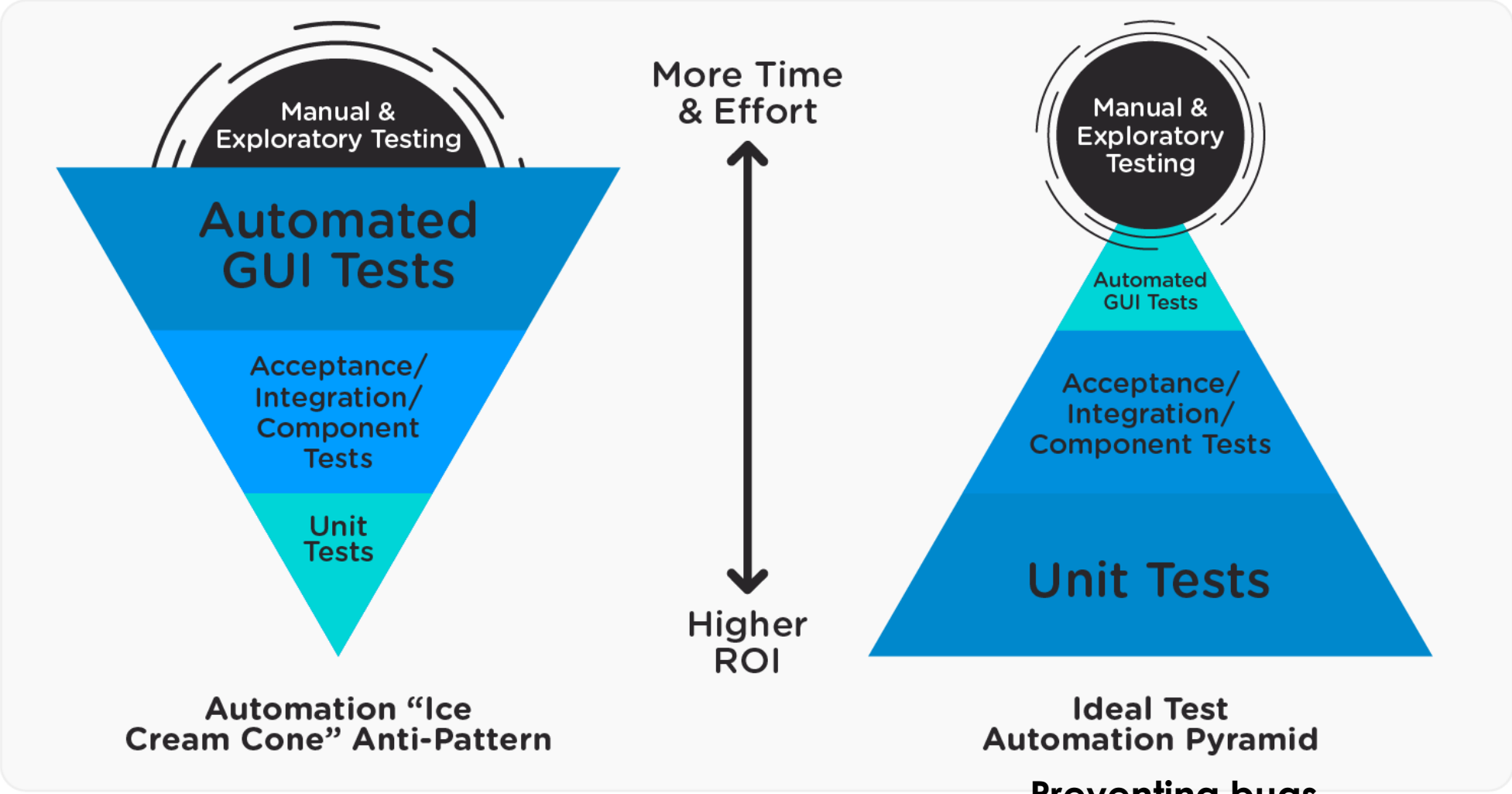
Robincse.blogspot.com

Non-Functional Requirements or Quality Requirements

Functional Requirements	Non Functional Requirements
• Product features	• Product property
• Describe the actions with which the user work is concerned	• Describe the experience of the user while doing the work
• A functions that can be captured in use cases	• Non-functional requirements are global constraints on a software system that results in development costs, operational costs
• A behaviors that can be analyzed by drawing sequence diagrams, state charts, etc	• Often known as software qualities
• Can be traced to individual set of a program	• Usually cannot be implemented in a single module of a program

http://image.slidesharecdn.com/non-functionalrequirements-150522055922-lva1-app6892/95/non-functional-requirements-do-we-really-care-9-638.jpg?cb=1432274604

Test Pyramid – which types of tests to spend the most effort on

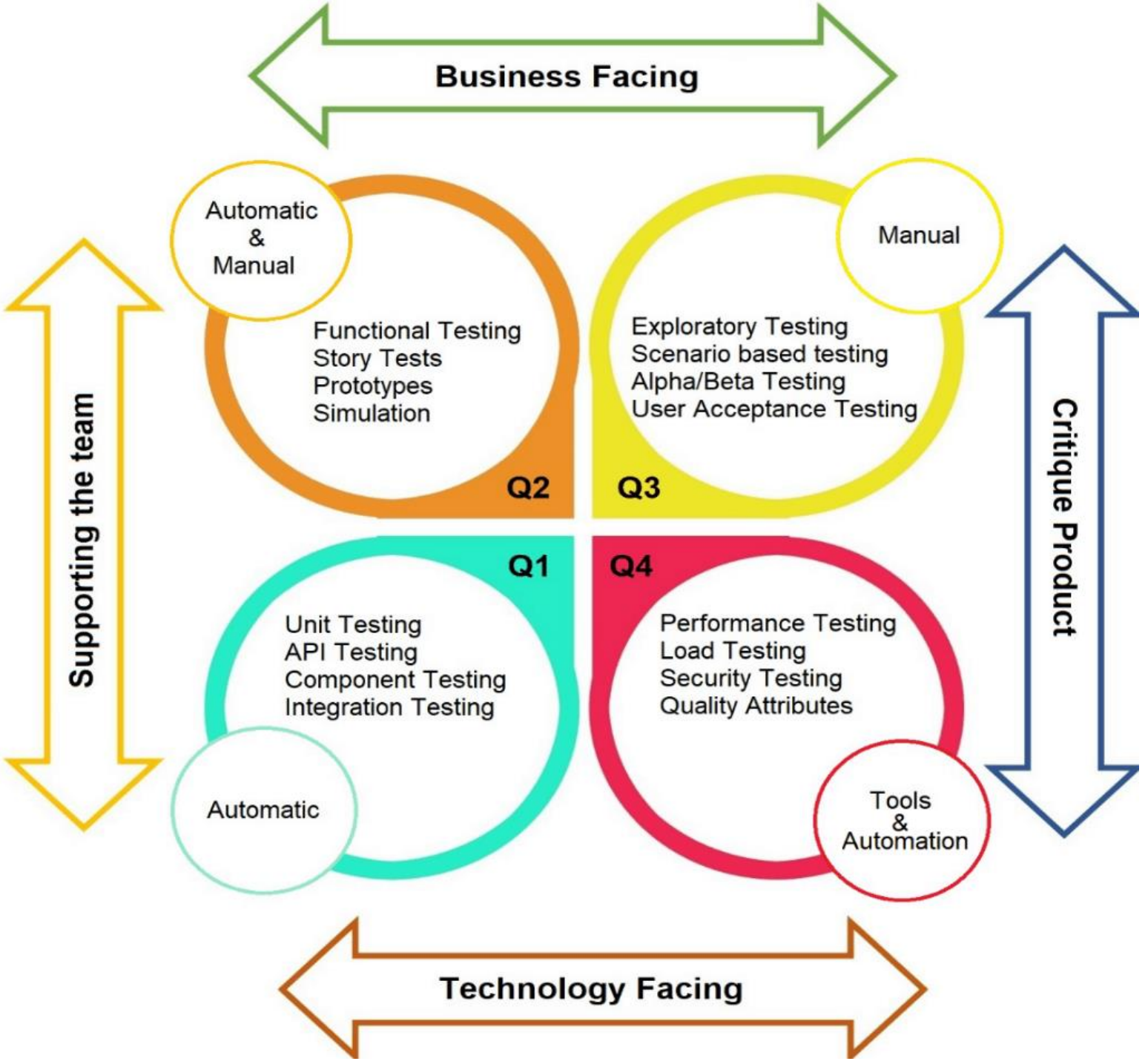


Finding bugs

Preventing bugs
(Mike Cohn's Agile testing pyramid)

https://abstracta.us/wp-content/uploads/2015/10/Automation_pyramid-EN.png

Crispin's Agile testing Quadrants



Webomates.com

Tony Clear S2 2024

Unit Tests with React (10:38)

https://youtu.be/ZmVBCpefQe8

A Basic Test

```
const expected = true;
const actual = false;

if (actual !== expected) {
  throw new Error(`${actual} is not ${expected}`);
}
```

Unhelpful Output

```
node tests/basic.test.js
/Users/chrisschmitz/code/presentation-react-testing/tests/basic.test.js:5
throw new Error(`${actual} is not ${expected}`);
^

Error: true is not false
  at Object.<anonymous> (/Users/chrisschmitz/code/presentation-react-testing/tests/basic.test.js:5:9)
  at Module._compile (internal/modules/cjs/loader.js:776:30)
  at Object.Module._extensions..js (internal/modules/cjs/loader.js:787:10)
  at Module.load (internal/modules/cjs/loader.js:653:32)
  at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
  at Function.Module._load (internal/modules/cjs/loader.js:585:3)
  at Function.Module.runMain (internal/modules/cjs/loader.js:829:12)
```

A Jest Test

```
const expected = true;
const actual = false;

test("it works", () => {
  expect(actual).toBe(expected);
});
```

```
➜ presentation-react-testing yarn jest tests/basic-jest.test.js
yarn run v1.19.0
$ /Users/chrisschmitz/code/presentation-react-testing/node_modu
FAIL tests/basic-jest.test.js
  ✕ it works (6ms)

  ● it works

    expect(received).toBe(expected) // Object.is equality

    Expected: false
    Received: true

      3 |
      4 | test("it works", () => {
    > 5 |   expect(actual).toBe(expected);
        |                  ^
      6 | });
      7 |

      at Object.<anonymous>.test (tests/basic-jest.test.js:5:18
```

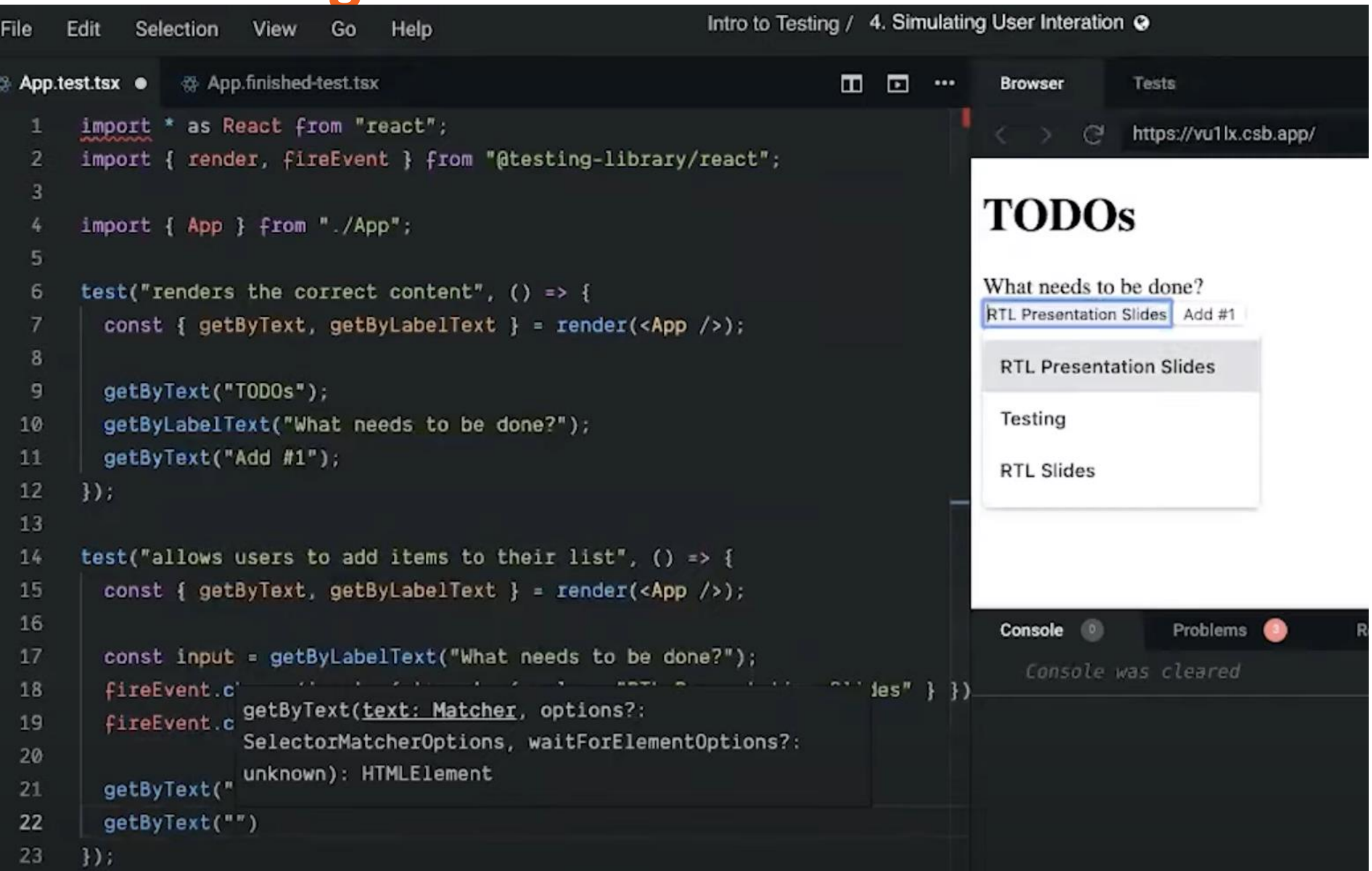
Test assertion Library
Test Runner
Mocking features

React testing – does it render correctly

App.test.tsx ● App.tsx

```
1  import * as React from "react";
2  import * as ReactDOM from "react-dom";
3  import { getQueriesForElement } from "@testing-library/dom";
4
5  import { App } from "./App";
6
7  test("renders the correct content", () => {
8    const root = document.createElement("div");
9    ReactDOM.render(<App />, root);
10
11   const { getByText, getByLabelText } = getQueriesForElement(root);
12
13   expect(getByText("TODOs")).not.toBeNull();
14   expect(getByLabelText("What needs to be done?").not.toBeNull();
15   // expect(root.querySelector("button").textContent).toBe("Add #1");
16 });
```

Simulating user interaction



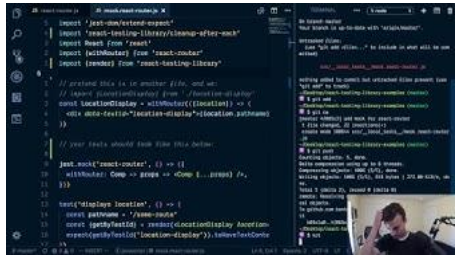
Unit testing Frameworks

https://www.youtube.com/watch?v=3e1GHCA3GP0
Unit Testing with Jest and React testing Library (24 mins 2020)

https://www.youtube.com/watch?v=ZmVBCpefQe8&t=13s
Getting started with React Testing (51 mins (2020)

https://www.youtube.com/watch?v=7r4xVDI2vho&t=30s
Jest Crash Course

https://www.youtube.com/watch?v=XDkSaCgR8g4&t=24s
Component Unit Testing and mocking with react testing library (14 mins 2018)



Testing Resources

https://www.youtube.com/watch?v=3e1GHCA3GP0&t=7s

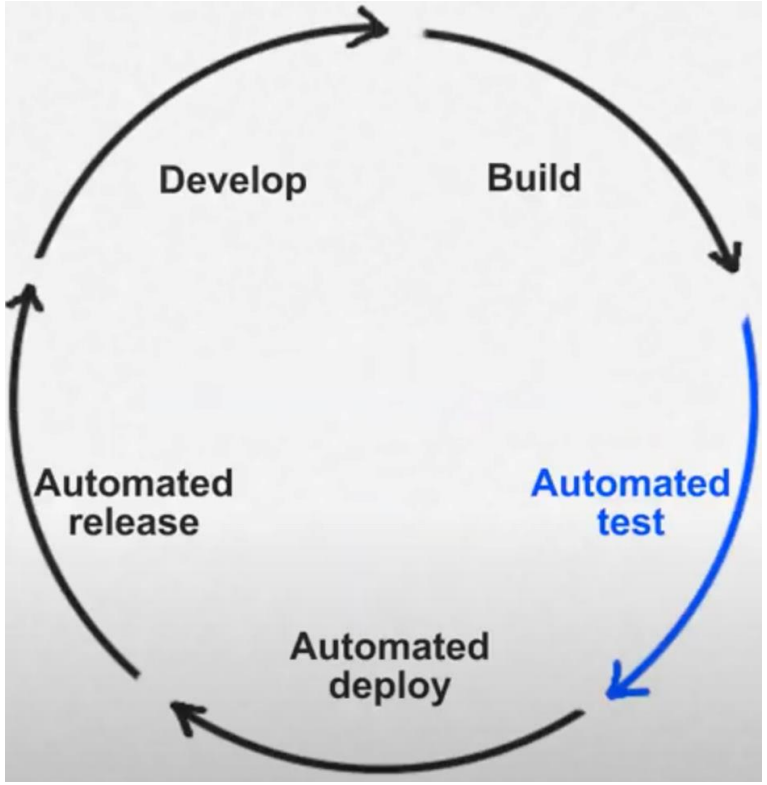
Complete Guide to Component testing with Jest for beginners. Crash course on Jest and mocking

https://www.youtube.com/watch?v=XDkSaCgR8g4&t=26s

Component Unit Testing (and mocking) with react-testing-library

Test Automation

Regression testing – run ALL tests in a code base (or selected, prioritised ones)



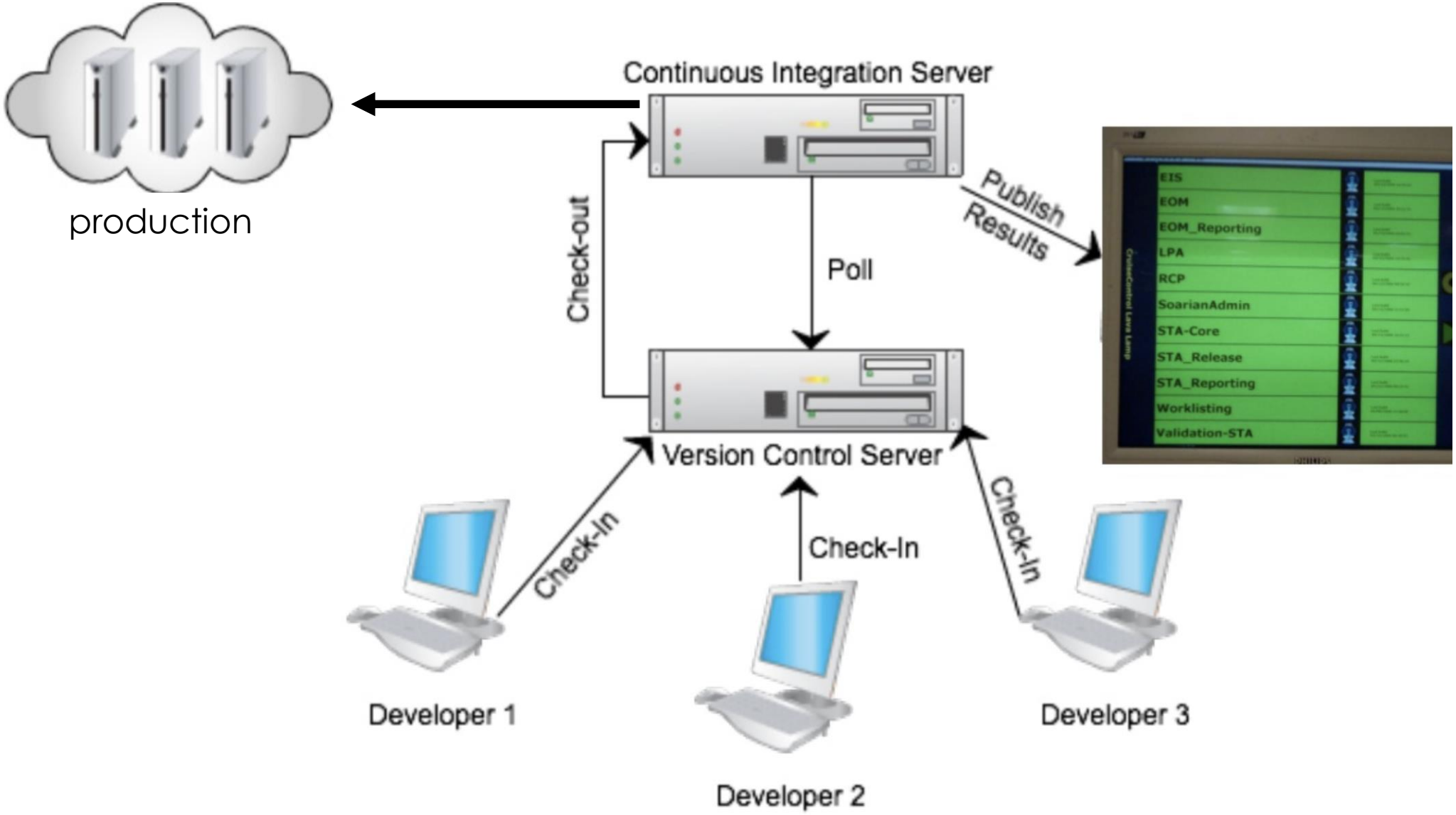
Continuous Delivery and DevOps ways of working rely on test automation to make rapid deployment possible

Test automation benefits

- guarantees tests are run at the right time and everything stops if they fail (still need to write and maintain the tests!)
- Frees up developers and testers for more analytical testing (exploratory, code reviews)

See the work of Michael Bolton and Linda Crispin!

Automating Integration Testing Continuously



What is Automated Testing? (7 mins, 2019)

https://youtu.be/Nd31XiSGJLw

Test cases for a unit test design

Test case ID	Test Case Description	Test setup Pre-conditions	Test Steps Instructions	Test data for each step (inputs)	Expected results Outputs

Write two test cases for this requirement for an online store:

If a user is a VIP customer and they order more than 10 items then they should not pay freight

Would this be a good user story?

As a VIP customer I want to be rewarded for my status so I can save money

Acceptance criteria (user stories)

Success criteria, Acceptance Tests

GIVEN [an initial context]

WHEN [some event happens]

THEN [an expected state linked to the functionality]

Given the person making an order is logged on and is a VIP customer and they order 11 items, **when** the order is confirmed **then** delivery of the order will be free

Backend API testing with Postman or Insomnia

Postman also covered in Worksheet 2

https://www.postman.com/product/what-is-postman/

https://learning.postman.com/docs/writing-scripts/script-references/test-examples/#getting-started-with-tests

Test Driven Development

Developers are responsible for writing unit tests

These are a check that their code is behaving as expected – choosing expected input and output values – particularly EDGE cases.

jUnit pHpUnit nUnit

https://medium.com/codeclan/testing-react-with-jest-and-enzyme-20505fec4675

https://medium.com/javascript-in-plain-english/i-tested-a-react-app-with-jest-testing-library-and-cypress-here-are-the-differences-3192eae03850

9 excuses why developers don't test their code

Can you think of any?…..

1. "My Code Works Fine — Why Should I Even Bother Testing It?"
2. "This Piece of Code Is Untestable"
3. "I Don't Know What to Test"
4. "Testing Increases the Development Time, and We're Running Out of Time"
5. "The Requirements Are No Good"
6. "This Piece of Code Doesn't Change"
7. "I Can Test This Way Faster If I do It Manually"
8. "The Client Only Wants to Pay for Deliverables"
9. "This Piece of Code Is So Small … It Won't Break Anything"

https://medium.com/better-programming/9-excuses-why-programmers-dont-test-their-code-8860a616b1b5

https://www.youtube.com/watch?v=llaUBH5oayw&list=FLAw_BzmvV1FBxEPeV4pDqug&index=7

What is TDD and the workflow? What are the benefits and why? What is the red green cycle What is refactoring?

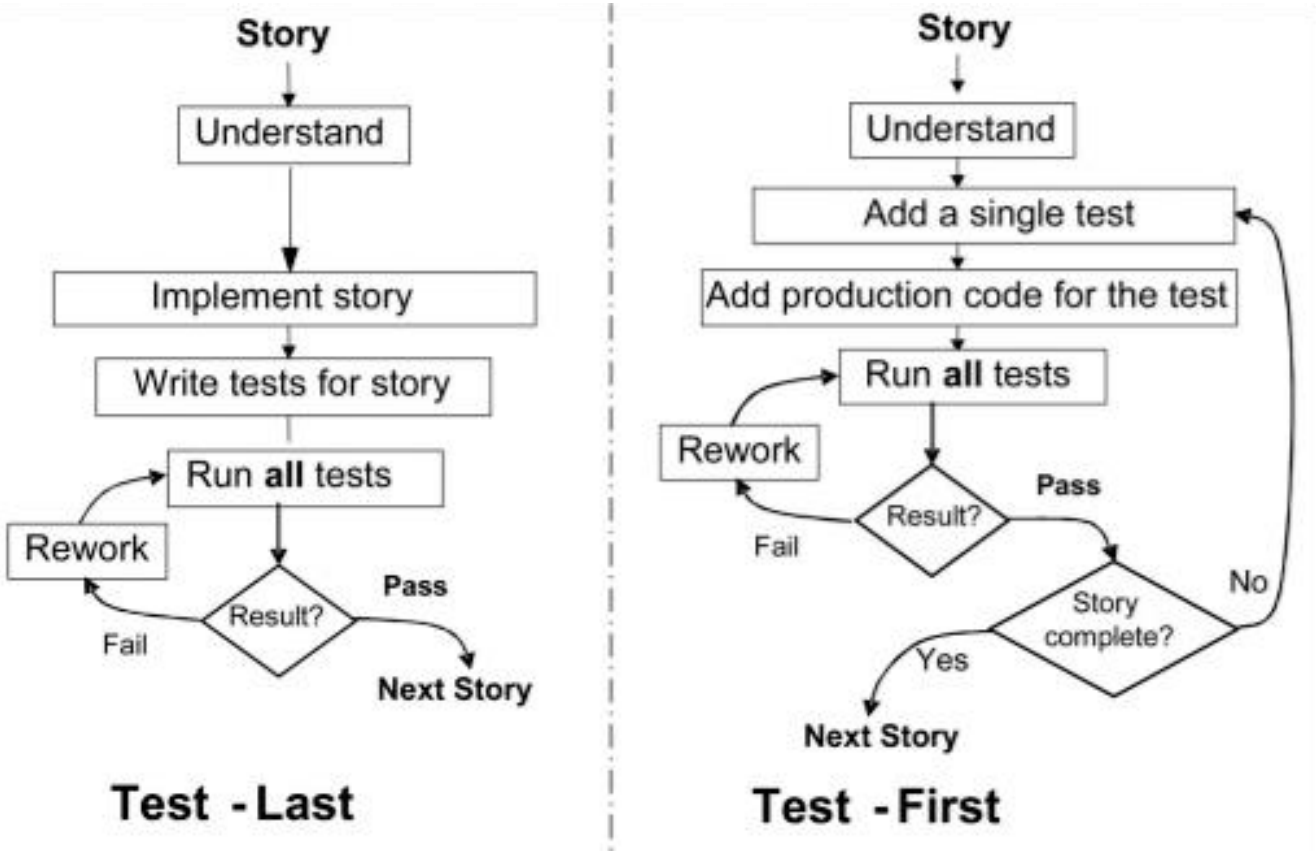
When Test Driven Development Goes Wrong

Test Driven Development is one of the best ways that we have to amplify our talent as software developers, maybe software engineers. This Software Engineering practice is one of the best ways to improve the quality of your code, but it is difficult to do it well, and it often goes wrong.

The interesting thing is that when it goes wrong, it may be at its most valuable. TDD is a cornerstone of Continuous Delivery, BDD and DevOps. Using it to give us valuable, efficient feedback on the quality of our designs is at the heart of its value but is often missed by people who are new to it. Dave explores these ideas with some real code examples to demonstrate the value of TDD. In this episode, Dave Farley explains 5 common ways that TDD goes wrong, how to fix them, and what we can learn from them.

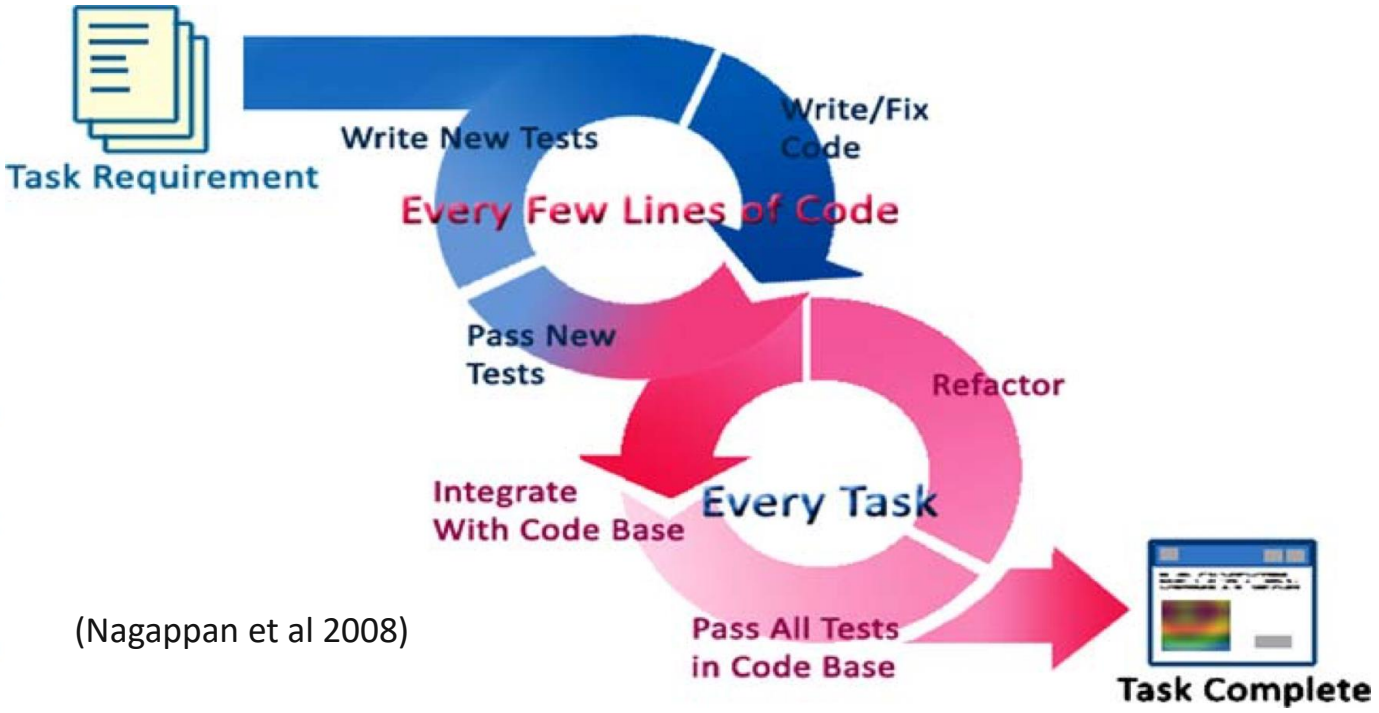
https://www.youtube.com/watch?v=UWtEVKVPBQ0&list=FLAw_BzmvV1FBxEPeV4pDqug&index=7

Test Driven Development



H.Erdogmus,*etal.*,"Ontheeffectivenessofthetest-firstapproachto programming," *Software Engineering, IEEE Transactions on,* vol. 31, pp. 226-237, 2005.

TDD = TFD + Refactoring



(Nagappan et al 2008)

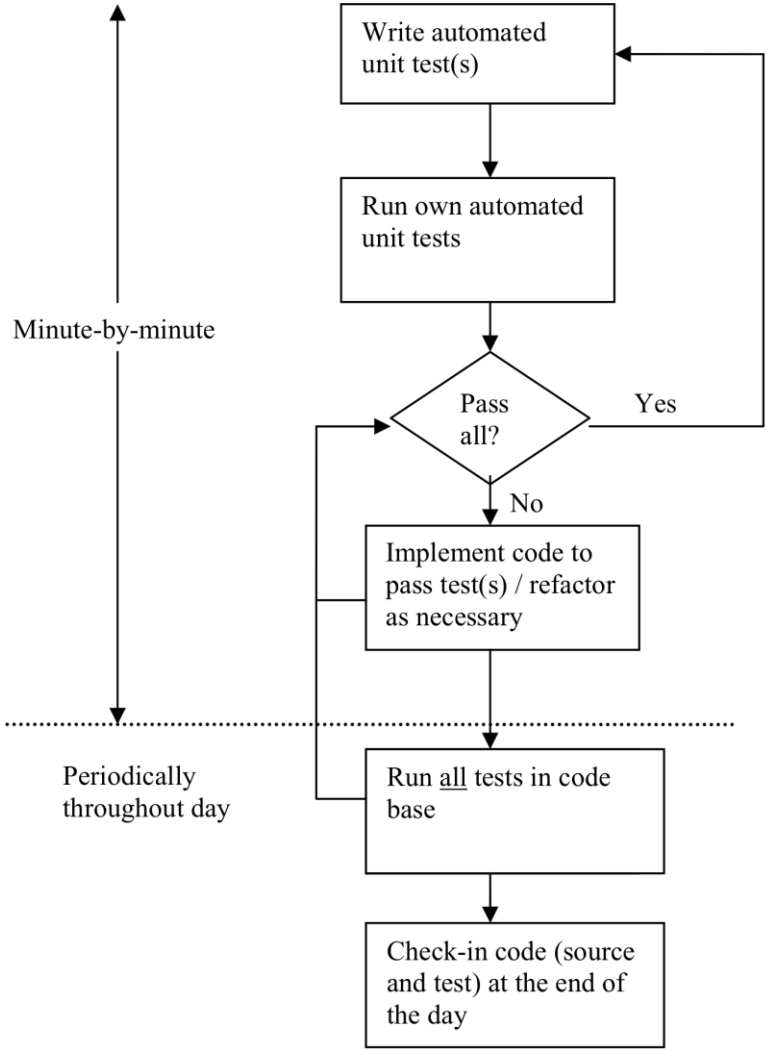


Figure 1: Test-Driven Development

(Bhat & Nagappan, 2006)

My client wants the dev to make a slight change to the number of books for a VIP discount to 10 or more.

Is the code change I make and example of Refactoring the code?

NO, a functional change is NOT refactoring
Refactoring just changes the code structure

Examples of TDD - for further study

Intro to TDD and where it came from and why it is needed. Good content – delivery SLOW
https://www.youtube.com/watch?v=dWayn0QsJr8

"Toy" Example of TDD code writing in Java and jUnit – look at others' comments
https://www.youtube.com/watch?v=O-ZT_dtlrR0

Another step-by step "toy" example in Java and jUnit– red and green refactoring

Lynda.com
Programming Foundations: Test-Driven Development. (EXCELLENT!)
Spring: Test-Driven Development with Junit
Exploring test-driven development with the assert keyword
From Advanced Java Programming course

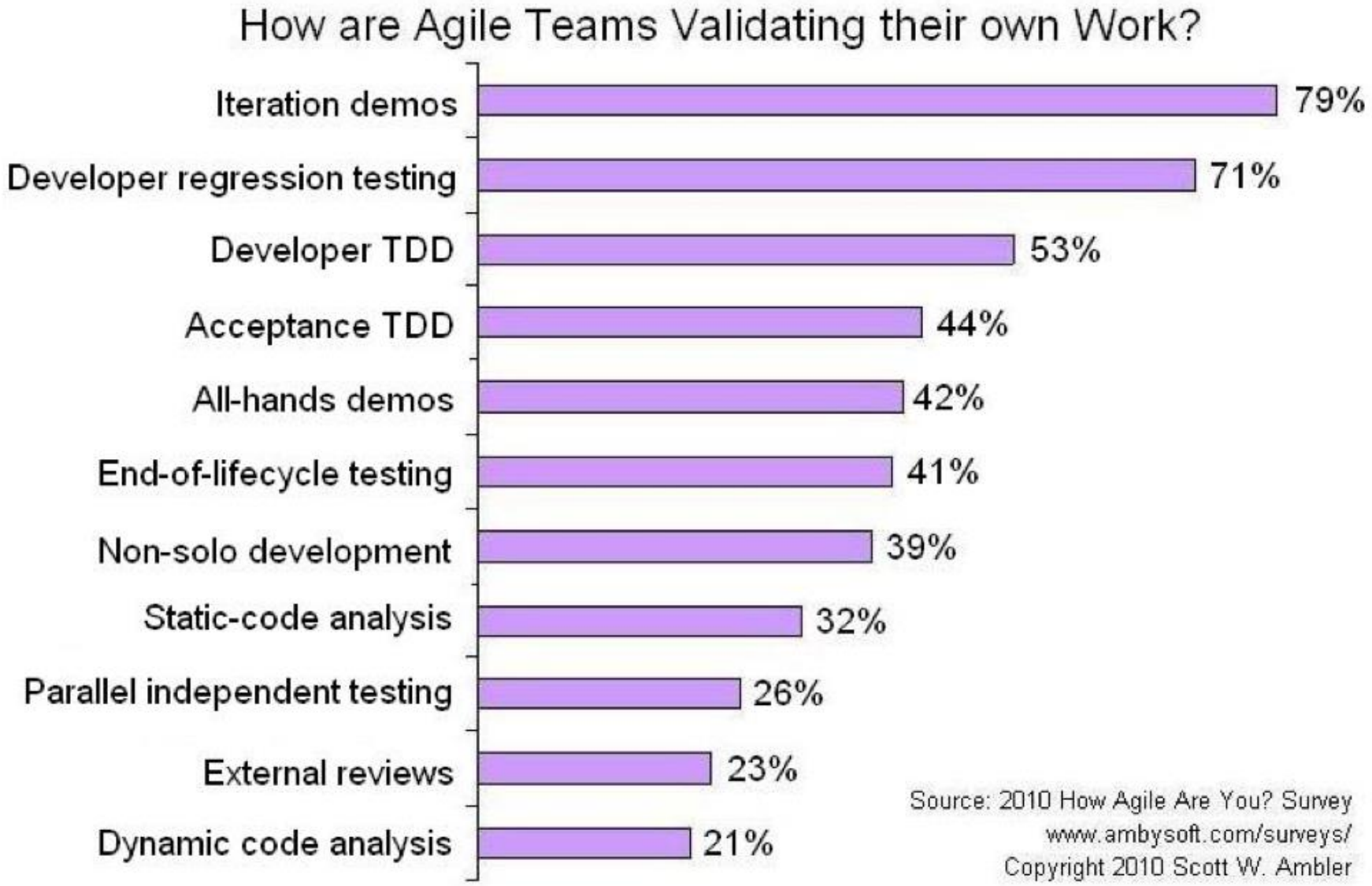
Testing setup

Sometimes we need a test database or **staging database** so we can do testing with data that isn't in the production database, and set up certain initial database conditions

Sometimes to test our object we rely on another class/object not yet coded – we can write a **mock object** for it that behaves (accepts/returns whatever) like the real object would with no actual processing. Our test can work with the mock object.

Don't forget about data – a test data base can be vital, a demo without a solid set of test data will be a failure ☹

Cf. test data for worksheet 4


How are Agile Teams Validating their own Work?
Iteration demos
79%
Developer regression testing
71%
Developer TDD
53%
Acceptance TDD
44%
All-hands demos
42%
End-of-lifecycle testing
41%
Non-solo development
39%
Static-code analysis
32%
Parallel independent testing
26%
External reviews
23%
Dynamic code analysis
21%
Source: 2010 How Agile Are You? Survey
www.ambysoft.com/surveys/
Copyright 2010 Scott W. Ambler

Test-Driven Development: Concepts, Taxonomy, and Future Direction

Test-driven development creates software in very short iterations with minimal upfront design. Poised for widespread adoption, TDD has become the focus of an increasing number of researchers and developers.

David Janzen
Simex LLC

Hossein Saiedian
University of Kansas

The *test-driven development* strategy requires writing automated tests prior to developing functional code in small, rapid iterations. Although developers have been applying TDD in various forms for several decades,[1] this software development strategy has continued to gain increased attention as one of the core extreme programming practices.

XP is an *agile method* that develops object-oriented software in very short iterations with little upfront design. Although not originally given this name, TDD was described as an integral XP practice necessary for analysis, design, and testing that also enables design through refactoring, collective ownership, continuous integration, and programmer courage.

Along with pair programming and refactoring, TDD has received considerable individual attention since XP's introduction. Developers have created tools specifically to support TDD across a range of languages, and they have written numerous books explaining how to apply TDD concepts. Researchers have begun to examine TDD's effects on defect reduction and quality improvements in academic and professional practitioner environments, and educators have started to examine how to integrate TDD into computer science and software engineering pedagogy. Some of these efforts have been implemented in the context of XP projects, while others are independent of them.

TEST-DRIVEN DEVELOPMENT DEFINED

Although its name implies that TDD is a testing method, a close examination of the term reveals a more complex picture.

The test aspect

In addition to testing, TDD involves writing automated tests of a program's individual units. A unit is the smallest possible testable software component. There is some debate about what exactly constitutes a unit in software. Even within the realm of object-oriented programming, both the class and method have been suggested as the appropriate unit. Generally, however, the method or procedure is the smallest possible testable software component.

Developers frequently implement test drivers and function stubs to support the execution of unit tests. Test execution can be either a manual or automated process and can be performed by developers or designated testers. Automated testing involves writing unit tests as code and placing this code in a test harness or framework such as JUnit. Automated unit testing frameworks minimize the effort of testing, reducing a large number of tests to a click of a button. In contrast, during manual test execution developers and testers must expend effort proportional to the number of tests executed.

Traditionally, unit testing occurred after developers coded the unit. This can take anywhere from a few minutes to a few months. The unit tests might be written by the same programmer or by a designated tester. With TDD, the programmer writes the unit tests prior to the code under test. As a result, the programmer can immediately execute the tests after they are written.

0018-9162/05/$20.00 © 2005 IEEE

Does Test-Driven Development Really Improve Software Design Quality?

David S. Janzen, *California Polytechnic State University, San Luis Obispo*
Hossein Saiedian, *University of Kansas*

TDD is first and foremost a design practice. The question is, how good are the resulting designs? Empirical studies help clarify the practice and answer this question.

Software developers are known for adopting new technologies and practices on the basis of their novelty or anecdotal evidence of their promise. Who can blame them? With constant pressure to produce more with less, we often can't wait for evidence before jumping in. We become convinced that competition won't let us wait.

Advocates for test-driven development claim that TDD produces code that's simpler, more cohesive, and less coupled than code developed in a more traditional test-last way. Support for TDD is growing in many development contexts beyond its common association with Extreme Programming. Examples such as Robert C. Martin's bowling game demonstrate the clean and sometimes surprising designs that can emerge with TDD,[1] and the buzz has proven sufficient for many software developers to try it. Positive personal experiences have led many to add TDD to their list of "best practices," but for others, the jury is still out. And although the literature includes many publications that teach us how to do TDD, it includes less empirical evaluation of the results.

In 2004, we began a study to collect evidence that would substantiate or question the claims regarding TDD's influence on software.

TDD misconceptions

We looked for professional development teams who were using TDD and willing to participate in the study. We interviewed representatives from four reputable Fortune 500 companies who claimed to be using TDD. However, when we dug a little deeper, we discovered some unfortunate misconceptions:

- Misconception #1: TDD equals automated testing. Some developers we met placed a heavy emphasis on automated testing. Because TDD has helped propel automated testing to the forefront, many seem to think that TDD is only about writing automated tests.
- Misconception #2: TDD means write all tests first. Some developers thought that TDD involved writing the tests (all the tests) first, rather than using the short, rapid test-code iterations of TDD.

Unfortunately, these perspectives miss TDD's primary purpose, which is design. Granted, the tests are important, and automated test suites that can run at the click of a button are great. However,

0740-7459/06/$25.00 © 2006 IEEE

Does Test-Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study

Maria Siniaalto[1] and Pekka Abrahamsson[2]

[1] F-Secure Oyj,
Elektroniikkatie 3, FIN-90570 Oulu, Finland
Maria.Siniaalto@f-secure.com
[2] VTT Technical Research Centre of Finland,
P.O. Box 1100, FIN-90571 Oulu, Finland
Pekka.Abrahamsson@vtt.fi

Abstract. It is suggested that test-driven development (TDD) is one of the most fundamental practices in agile software development, which produces loosely coupled and highly cohesive code. However, how the TDD impacts on the structure of the program code have not been widely studied. This paper presents the results from a comparative case study of five small scale software development projects where the effect of TDD on program design was studied using both traditional and package level metrics. The empirical results reveal that an unwanted side effect can be that some parts of the code may deteriorate. In addition, the differences in the program code, between TDD and the iterative test-last development, were not as clear as expected. This raises the question as to whether the possible benefits of TDD are greater than the possible downsides. Moreover, it additionally questions whether the same benefits could be achieved just by emphasizing unit-level testing activities.

Keywords: Test-Driven Development, Test-first Programming, Test-first Development, Agile Software Development, Software Quality.

1 Introduction

Test-driven development (TDD) is one of the core elements of Extreme Programming (XP) method [1]. The use of the TDD is said to yield several benefits. It is claimed to improve test coverage [2] and to produce loosely coupled and highly cohesive systems [3].

What Makes Testing Work: Nine Case Studies of Software Development Teams

Christopher D Thomson*
Business School
University of Hull
Hull, UK
c.thomson@hull.ac.uk

Mike Holcombe, Anthony J H Simons
Department of Computer Science
University of Sheffield
Sheffield, UK
{m.holcombe,a.simons}@dcs.shef.ac.uk

***Abstract*— Recently there has been a focus on test first and test driven development; several empirical studies have tried to assess the advantage that these methods give over testing after development. The results have been mixed. In this paper we investigate nine teams who tested during coding to examine the effect it had on the external quality of their code. Of the top three performing teams two used a documented testing strategy and the other an ad-hoc approach to testing. We conclude that their success appears to be related to a testing culture where the teams proactively test rather than carry out only what is required in a mechanical fashion.**

Testing; test first; test driven development; extreme programming;emprical; qualitative; testing culture.

I. Introduction

Extreme programming (XP) [1] presents what is, on the surface, a simple but effective testing practice known as *test first*, or *test driven development*. The idea is reassuringly simple, that unit tests should be defined and run before any implementation is present. Several studies have attempted to measure the effect of the *test first* practice on the quality of the software produced and time taken, but the results presented are inconclusive.

The testing practice of XP encompassed more than simply *test first*. System testing is automated, incremental, regular, and early. User acceptance testing is similar although often less or not at all automated [1]. But these features can be used independently of *test first* and perhaps with some success. This raises our research question:

How does the practice of testing affect a team following XP – or if test first is not followed then do the other practices still influence the way testing is performed and the external quality?

accurately. We found that the teams had a high degree of variation in external quality that cannot be easily explained by the teams' testing practice alone or the practices of XP alone. Instead we find that testing must become part of the culture of the team, and can do so in at least two different ways.

II. Literature Review

Test first (TF) is an established development technique which is essentially the same as *test driven development* (TDD). In both cases the aim is to write tests before writing the functional code. This should in theory aid development as the tests form the basis of the specification, design and functional tests, whereas testing after the code is written is regarded as only a testing technique [1; 2]. Whilst TF and TDD are well defined, the traditional or *test last* technique is interpreted differently by the studies in the literature investigating this phenomenon. The studies' definitions of *test last* can be divided into roughly four categories, which we will refer to as TL 1 to 4:

TL-1: Unspecified traditional method. Most experiments provided at least a few hints as to the method that the non-TF teams should follow, however some did not [3-5]. The following statement was typical of the broad definitions that these papers used: "the control group which followed the traditional process" ([5], p132). As no further discussion was made about the process followed we can't make generalizations about what affected their performance.

TL-2: Specified traditional method. This method is typified by manual or ad-hoc testing and was used in three studies [6-8]. The method was defined thus: "no automated tests were written and the project was developed in traditional mode with a large up-front design and manual testing after the software was implemented." ([7], p71) and

Third International Conference on Software Testing, Verification, and Validation Workshops

Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers

Mauricio Finavaro Aniche, Marco Aurélio Gerosa
Department of Computer Science - University of São Paulo (USP) - Brazil
{aniche,gerosa}@ime.usp.br

Abstract

Test-driven development (TDD) is a software development practice that supposedly leads to better quality and fewer defects in code. TDD is a simple practice, but developers sometimes do not apply all the required steps correctly. This article presents some of the most common mistakes that programmers make when practicing TDD, identified by an online survey with 218 volunteer programmers. Some mistakes identified were: to forget the refactoring step, building complex test scenarios, and refactor another piece of code while working on a test. Some mistakes are frequently made by around 25% of programmers.

1. Introduction

Test-driven development (TDD) is an important practice in Extreme Programming (XP) [1]. As agile practices suggest, software design emerges as software grows. In order to respond very quickly to changes, a constant feedback is needed and TDD gives it by making programmers constantly write a small test that fails and then make it pass. TDD is considered an essential strategy in emergent design because when writing a test prior to code, programmers contemplate and decide not only the software interface (e.g. class/method names, parameters, return types, and exceptions thrown), but also on the software behavior (e.g. expected results given certain inputs) [13].

TDD is not only about test. It is about helping the team to understand the features that the users need and to deliver those features reliably and predictably. TDD turns testing into a design activity, as programmers use

Kent Beck sums up TDD as follows: 1) quickly add a test; 2) run all tests and see the new one fail; 3) make a little change; 4) run all tests and see them all succeed; 5) refactor to remove duplication [18]. In order to get all benefits from TDD, programmers should follow each step. As an example, the second step states that programmers should watch the new test fail and the fifth step states to refactor the code to remove duplication. Sometimes programmers just do not perform all steps of Beck's description. Thus, the value TDD aggregates to software development process might be reduced.

This article presents some of the most common mistakes that programmers make during their TDD sessions, based on an online survey conducted during two weeks in January, 2010, with 218 volunteer programmers. The survey and its data can be found at http://www.ime.usp.br/~aniche/tdd-survey/.

This article is structured as follows: Section 2 presents some studies about the effects of TDD on software quality; Section 3 shows the most common mistakes programmers make based on the survey; Section 4 discusses about the mistakes and ideas on how to sort them out; Section 5 presents threats to validity on the results of this article; Section 6 concludes and provides suggestions for future works.

2. The effects of TDD on software quality

Empirical experiments about the effects of TDD have been conducted generally with two different groups: graduate students at universities and professional developers at the industry. Most of them show that TDD increases code quality, reduces the defect density, and provides better maintainability.

In practice, few organizations apply the strict TDD process in the form of the repetition of the sequence of steps described above. The real insight has been test-first development and, more specifically, the idea that any new code must be accompanied by new tests. It is not even critical that the code should come only after the test (the "F" of TFD): what counts is that you never produce one without the other.

This idea has come to be widely adopted —and should be adopted universally. It is one of the major contributions of agile methods.

(Meyer 2014)

TDD Empirical Study at SERL in AUT

Full citation: Buchan, J., Li, L., & MacDonell, S.G. (2011) Causal Factors, Benefits and Challenges of Test-Driven Development: Practitioner Perceptions, in Proceedings of the 18th Asia-Pacific Software Engineering Conference (APSEC 2011). Hochiminh City, Vietnam, IEEE Computer Society Press, pp.405-413.
doi: 10.1109/APSEC.2011.44

Causal Factors, Benefits and Challenges of Test-Driven Development: Practitioner Perceptions

Jim Buchan, Ling Li, Stephen G. MacDonell

SERL, School of Computing and Mathematical Sciences
AUT University, Private Bag 92006
Auckland 1142, New Zealand
jim.buchan@aut.ac.nz, angela.linz@gmail.com, stephen.macdonell@aut.ac.nz

Abstract

This report describes the experiences of one organization's adoption of Test Driven Development (TDD) practices as part of a medium-term software project employing Extreme Programming as a methodology. Three years into this project the team's TDD experiences are compared with their non-TDD experiences on other ongoing projects. The perceptions of the benefits and challenges of using TDD in this context are gathered through five semi-structured interviews with key team members. Their experiences indicate that use of TDD has generally been positive and the reasons for this are explored to deepen the understanding of TDD practice and its effects on code quality, application quality and development productivity. Lessons learned are identified to aid others with the adoption and implementation of TDD practices, and some potential further research areas are suggested.

Keywords: test-driven development (TDD), TDD benefits, TDD challenges, causal network

I. INTRODUCTION

Test-driven development (TDD), which emphasizes a mind-set that functional code should be changed only in response to a failed test, is considered "proven practice" by many contemporary software development practitioners and text-book writers. Although it is a technique that has been practiced for decades [1], it has recently gained more visibility with the rise in use of Agile methodologies such as Extreme Programming (XP), where it is a core practice [2].

Proponents of TDD have reasoned that its use should result in improvements to code quality [3], testing quality [4], and application quality [5], compared to the traditional Test-Last (TL) approach. It has also been claimed to improve overall development productivity, encourage early understanding of the scope of requirements (user stories), as well as potentially leading to enhanced developer job satisfaction and confidence [3].

In contrast, critics claim that the frequent changes to tests in TDD are more likely (than in TL) to cause test breakages, leading to costly rework and loss of productivity [6]. Boehm and Turner [6] also note that with TDD the consequences of developers having inadequate testing skills may be amplified, compared to the consequences for a TL approach. Other critics note that TDD may not be appropriate for all application domains [7].

While these claims and criticisms provide some basis for evaluating the possible utility of TDD, practitioners and researchers have recognized the need for stronger evidence as a basis for investing – or not – in the effort to adopt and implement this set of practices. Recently, empirical researchers have been investigating the claimed benefits, constraints, and applicability of TDD in a variety of industrial and academic settings to build up a body of evidence. This empirical evidence is mixed in its results regarding the benefits of TDD (covered in more detail in Section VI).

This report adds further evidence regarding TDD in practice by describing the experiences of a software development team that has used TDD for three years in a specific project. This project (referred to as the "TDD project") adopted Extreme Programming (XP) as a development methodology. This was the first project to adopt TDD in this organization

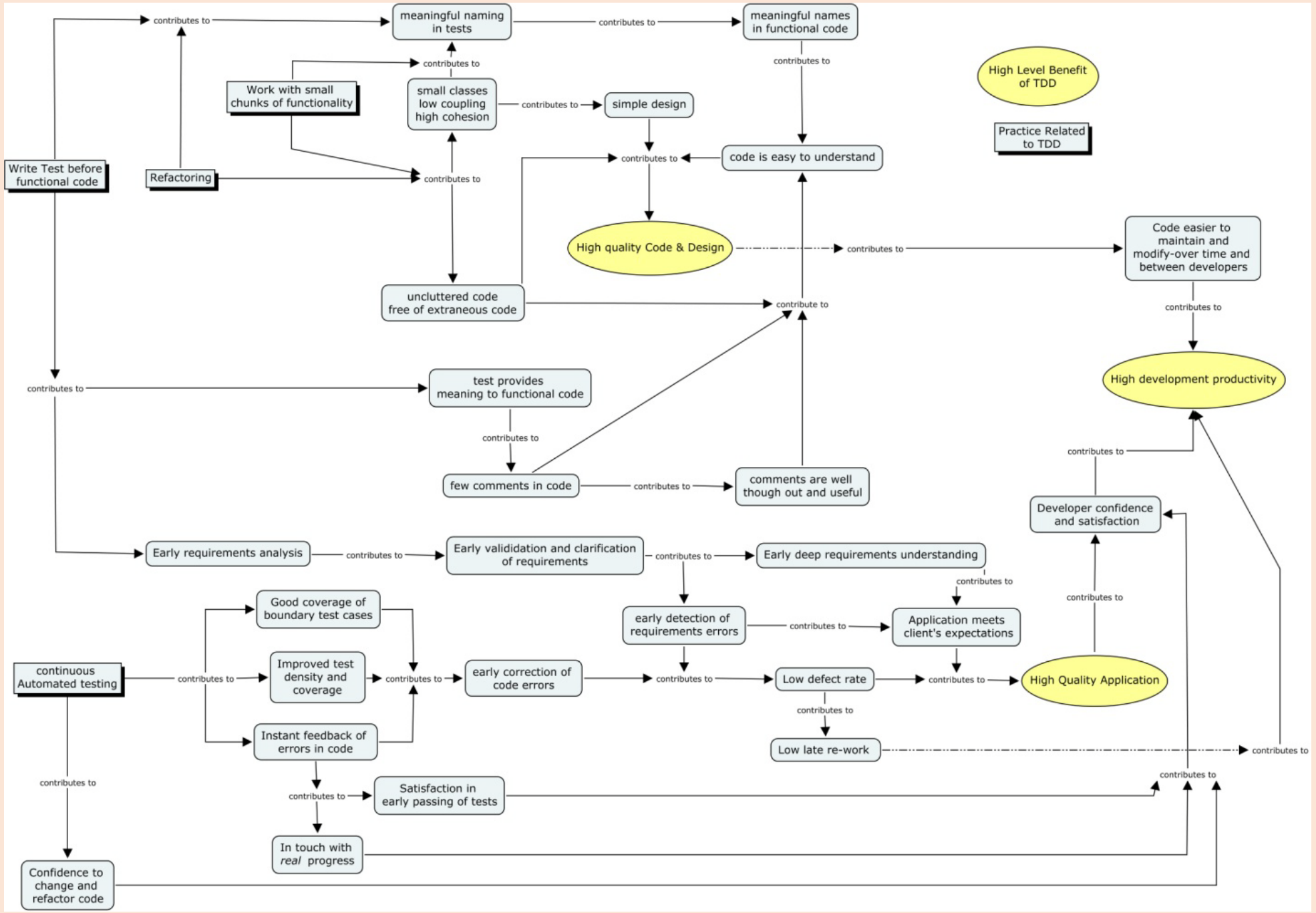


Figure 2. Causal network of TDD benefits and contributing factors.

TDD Research

Table 1. Summary of TDD research in industry.

Study	Type	Number of companies	Number of programmers	Quality effects	Productivity effects
George[8]	Controlled experiment	3	24	TDD passed 18% more tests	TDD took 16% longer
Maximilien[9]	Case study	1	9	50% reduction in defect density	Minimal impact
Williams[10]	Case study	1	9	40% reduction in defect density	No change

Table 2. Summary of TDD research in academia.

Controlled experiment	Number of programmers	Quality effects	Productivity effects
Kaufmann[11]	8	Improved information flow	50% improvement
Edwards[12]	59	54% fewer defects	n/a
Erdogmus[13]	35	No change	Improved productivity
Müller[14]	19	No change, but better reuse	No change
Pančur[15]	38	No change	No change

(Hossein 2005)

Empirical Software Engineering (2020) 25:2630–2660
https://doi.org/10.1007/s10664-020-09818-7

How software engineering research aligns with design science: a review

Emelie Engstro¨ m[1] · Margaret-Anne Storey[2] · Per Runeson[1] · Martin Ho¨ st[1] · Maria Teresa Baldassarre[3]

Published online: 18 April 2020

Abstract

Background Assessing and communicating software engineering research can be challenging. Design science is recognized as an appropriate research paradigm for applied research, but is rarely explicitly used as a way to present planned or achieved research contributions in software engineering. Applying the design science lens to software engineering research may improve the assessment and communication of research contributions.

Aim The aim of this study is 1) to understand whether the design science lens helps summarize and assess software engineering research contributions, and 2) to characterize different types of design science contributions in the software engineering literature.

Method In previous research, we developed a visual abstract template, summarizing the core constructs of the design science paradigm. In this study, we use this template in a review of a set of 38 award winning software engineering publications to extract, analyze and characterize their design science contributions.

Results We identified five clusters of papers, classifying them according to their different types of design science contributions.

Conclusions The design science lens helps emphasize the theoretical contribution of research output—in terms of technological rules—and reflect on the practical relevance, novelty and rigor of the rules proposed by the research.

Keywords Design science · Research review · Empirical software engineering

Empirical Studies of SE and How to communicate findings with practitioners?

Engström, E., Storey, M.-A., Runeson, P., Höst, M., & Baldassarre, M. T. (2020, 2020/07/01). How software engineering research aligns with design science: a review. *Empirical Software Engineering, 25*(4), 2630-2660. https://doi.org/10.1007/s10664-020-09818-7

VASE: Visual Abstracts for Software Engineering

Slides and materials from a tutorial given at CBSoft 2019 (held in Brazil) describe how to use the design science template in action are posted here: https://github.com/margaretstorey/cbsoft2019tutorial/ (the tutorial was also about research methods, see slides 64 of the slides deck https://github.com/margaretstorey/cbsoft2019tutorial/blob/master/CBSoft%20Tutorial%202019%20Slides.pdf).

Slides from talk at ESEM are available here: https://www.slideshare.net/mastorey/using-a-visual-abstract-as-a-lens-for-communicating-and-promoting-design-science-research-in-software-engineering

Related blog post is here: http://margaretstorey.com/blog/2017/11/09/visual-abstracts/

Empirical software engineering research aims to generate prescriptive knowledge that can help software engineers improve their work and overcome their challenges, but deriving these insights from real-world problems can be challenging. We promote design science as an effective way to produce and communicate prescriptive knowledge while we propose using a visual abstract template to communicate design science contributions and highlight the main problem/solution constructs of this area of research, as well as to present the validity aspects of design knowledge.

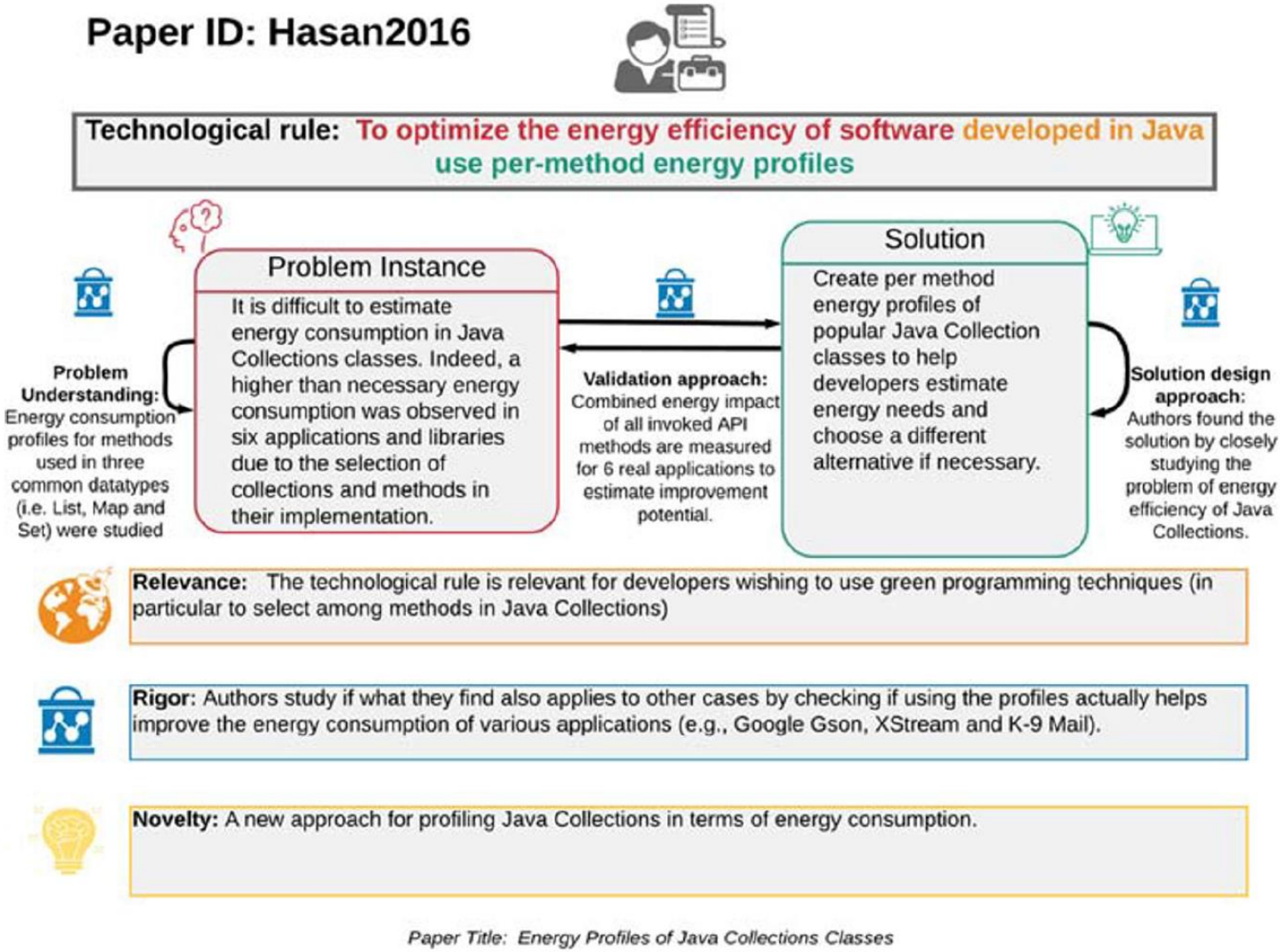
We have posted a template of this visual abstract, and we welcome comments as well as examples of how this abstract can be applied to your research!

dfucci Davide Fucci

Sharing findings with practitioners
DSSE.org
https://github.com/margaretstorey/VASE

Sharing findings with practitioners

DSSE.org
https://github.com/margaretstorey/VASE



Questions and Comments….