HELLENIC REPUBLIC
UNIVERSITY OF MACEDONIA

ΔΗΜΟΚΡΙΤΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΡΑΚΗΣ
DEMOCRITUS UNIVERSITY OF THRACE

Master of Science in Law and Informatics

The journey to Continuous Compliance

for software development

Styliani Rouzi

The journey to continuous compliance, for software development

Styliani Rouzi

M.Sc. Thesis

submitted as a partial fulfillment of the requirements for the

MASTER OF SCIENCE IN «Law and Informatics»

Supervisor

Psannis Konstantinos

Περίληψη

Με την εφαρμογή διαδικασιών διασφάλισης ποιότητας μπορούμε να προστατεύσουμε το λογισμικό και την ομάδα μας από λάθη εκ παραδρομής, παραλείψεις και παρανοήσεις. Οποιαδήποτε αυθαίρετη παραδοχή σχετικά με τον κώδικα μπορεί να επηρεάσει αρνητικά μια επιχείρηση, έναν οργανισμό και τη φήμη μιας ομάδας ανάπτυξης. Επομένως η ύπαρξη ουσιαστικών διαδικασιών είναι υψίστης σημασίας ώστε να εξασφαλίσουμε την πραγματική επίτευξη των στόχων που τίθενται. Οι αυτοματισμοί αποτελούν ένα σημαντικό εργαλείο για τους προγραμματιστές εδώ και χρόνια, καθώς η χρήση τους συμβάλλει στη βελτίωση του χρόνου ολοκλήρωσης ενός νέου χαρακτηριστικού, αυξάνει την ακρίβεια του κώδικα και προσφέρει έγκαιρη ενημέρωση και αποτελέσματα. Η παρούσα εργασία εστιάζει σε άρθρα σχετικά με πληθώρα θεματικών αναφορικά με τη **διασφάλιση ποιότητας** και τη **συμμόρφωση**, συμπεριλαμβανομένου των αυτοματοποιημένων ελέγχων και της διαδικασίας συνεχούς ενσωμάτωσης καθώς επίσης παρουσιάζει προβληματισμούς και ιδέες γύρω από σχετικά ζητήματα όπως το **κόστος επιδιόρθωσης σφαλμάτων** του κώδικα και την ύπαρξη **προκαταλήψεων στον κώδικα**. Το δεύτερο κομμάτι της εργασίας προσεγγίζει την **πρακτική πλευρά της αυτοματοποίησης** στη διασφάλιση ποιότητας, περιγράφοντας αναλυτικές οδηγίες βήμα-βήμα για μεθοδολογίες με τις οποίες μπορεί μια ομάδα να ξεκινήσει να **εισάγει αυτοματοποιημένους ελέγχους** σε υφιστάμενο λογισμικό και πώς να **αξιολογήσει τη σουίτα αυτοματοποιημένων ελέγχων** ως προς την αποτελεσματικότητά της και με κριτήρια ποιότητας των περιπτώσεων που ελέγχονται (αντί της απλοϊκής προσέγγισης του πόσος κώδικας θεωρητικά ελέγχεται).

Λέξεις Κλειδιά:

Συνεχής συμμόρφωση, συνεχής ενσωμάτωση, αυτοματοποιημένοι έλεγχοι, εργαλεία επαλήθευσης, έλεγχοι συμμόρφωσης, διασφάλιση ποιότητας, διασφάλιση ποιότητας ως προς τη συμμόρφωση, προκαταλήψεις, έγκαιρος εντοπισμός προβλημάτων, διόρθωση σφαλμάτων, τεχνικό χρέος, κανονισμοί

Abstract

By establishing Quality Assurance (QA) procedures we shield ourselves - our software and our team - from mistakes, misconceptions and oversights. Making assumptions about the well-being of the code and the results it produces is detrimental to the well-being of a corporation, a business, an organization, or frankly to the reputation of any kind of engineering team. Therefore, it is crucial that meaningful processes are in place to ensure our goals are met. Automation has been a great addition to the armory of developers for years now; using it can help not only save time in coding a new feature and increase accuracy of the outcome but also report test results sooner. This thesis presents articles related to various aspects of Quality Assurance **(QA) and compliance**, including automated testing and CI process, as well as focuses on concerns and ideas regarding relevant issues, such as the **cost of bug fixes** and **bias introduced to code**. The second part of the thesis is dedicated to more **practical aspects of automation** in QA, by providing detailed step-by-step instructions for methodologies with which teams can approach automated testing and can **start introducing automated tests** for existing software, as well as **how a test suite can be meaningfully evaluated** by employing factors of effectiveness and quality of its test cases (rather than merely examining the coverage it provides, meaning the rather theoretical estimation of how much code is being tested).

Keywords

Continuous compliance, continuous integration, automated testing, verification tools, conformance testing, QA, compliance qa, bias, early error reporting, bug fix, technical debt, audit, policy

Preface - Acknowledgments

There are multiple people who, in the past few years, have played a significant role in my growth and knowledge expansion. I am grateful to the experiences and conversations I was privy to, and to everyone who contributed to my being where I am today, by sharing their expertise, not withholding, untiringly offering a different point of view and explaining in great detail the matter at hand.

I had the privilege to witness first hand how security-oriented teams operate: with precision and a holistic approach, going deep but also being fast paced. And, as part of an engineering team myself, this is a great example to look up to.

A big “Thank you” goes to all “Law and Informatics” personnel for creating an interesting, informative and useful curriculum as well as for providing me with the opportunity to attend.

Special thanks to my supervisor, Konstantinos Psannis, for his trust in me to work on, and complete, this thesis.

I am particularly thankful to all the people who have mentored me throughout the years and shaped me in a way that has led me to this path of knowledge.

I am also grateful for the support and inspiration from many people in my life, including Antonis Eleftheriadis who has also contributed in proofreading this document.

Table of contents

1. Introduction

The goal of this paper is to show the evolution of quality assurance and how it needs to rise to modern standards and satisfy the needs of organizations, corporations, teams and software within the era of constant technological evolution. From writing code to automated testing and security, the industry standards keep rising, and we simply cannot exclude compliance for policies, laws and industry requirements from the SDLC (Software Development Life Cycle). The software development process, as we know it, keeps outgrowing our definitions; there has been a huge shift to security and how SDLC is really SSDLC, as in Secure Software Development Life Cycle, and how we need to build applications which are secure by design. We now need to also address compliance in the same spirit, i.e. create software which is compliant by default and by design.

In this paper we aim to present a range of different topics on the broader topic of how to be compliant within the scope of software development, discuss the unique challenges of compliance, and provide practical methodologies to get started with automated testing (as the backbone of continuous compliance).

Just like security shouldn't be an afterthought, neither should compliance. Software should be **designed** with both **security and compliance already in mind**.

1.1. Problem Statement

Testing, just like security, didn’t always come first in reference with software development, and was often seen as an overhead to software developer teams, or at least as an extra step that occurred only after a certain functionality had been implemented. Staying afloat in a world that is constantly moving, means your software is adapting to the new requirements. Consequently, code changes and what we knew to be working, needs to be re-confirmed.

Thankfully we have significantly progressed to considering security and compliance top priorities not just for legal departments and shareholders’ meetings but for every layer and every team of an organization.

Challenges, however, still arise when it comes to automating processes and verification tools, be it testing or compliance. To achieve high quality software we assume we got the specs right in the beginning. It goes without saying that faulty specs can and will be the root of all sorts of problems, and just like we want to add automated tests for software, we should also properly test and verify the initial goal of a project/feature, its specs, designs, user stories and acceptance criteria – and sometimes even then we still need to provide a good level of collaboration and transparent communication to make sure we have all understood the same thing, so that we truly build what’s expected. Clear goals and open communication, between the various teams of an organization and different people of a team, is certainly another important topic when it comes to teams as well as software development. When referring to “confirmation everything works as expected” we assume that we are all expecting the same result and that the provided specs truly align with the business goals of the organization. Increasing the level of certainty that we indeed have a common understanding of what we want to build, and more important of what we *have* built, is often of critical priority.

1.2. Scope of the study

This paper aims to shed light on the importance of incorporating meaningful processes in the lifecycle of software development, and maintaining a high level of confidence in the end result when making any changes. This will help engineering teams maintain a peace of mind when introducing new code, as well as provide project stakeholders with useful data.

While compliance is a widely used term in a variety of industries, continuous compliance is still new, not widely known and thus not implemented within software development teams. This paper is written from the perspective of not just software engineering but also product and business, and it focuses on a collection of important aspects that both engineers, team leads and higher management should keep in mind. Its added value includes that it aims to provide an overview of issues surrounding the absence of continuous compliance, address challenges related to technical teams when it comes to testing for compliance purposes, and provide a practical approach of how to get started with testing and what to watch out for in order to deploy a meaningful test suite. Gathering all the necessary information together along with a practical approach of getting started, hopefully serves as a passage through uncertainty in order to establish modern methodologies to support and improve compliance within organizations.

1.3. Terminology

The unique terminology used in parts of the document is explained within the text itself, and the important keywords can be found in Annex II - Glossary. Details about testing, automated vs manual testing as well as its role in software development can be found in various resources[1] as do the different types available for testing[2].

As far as the term “Compliance” is concerned, it refers to the verification of conformity with law, policies and other requirements (internal, external, industry specific). We will be focusing particularly on software, and compliant software, meaning software that adheres to policy guidelines.

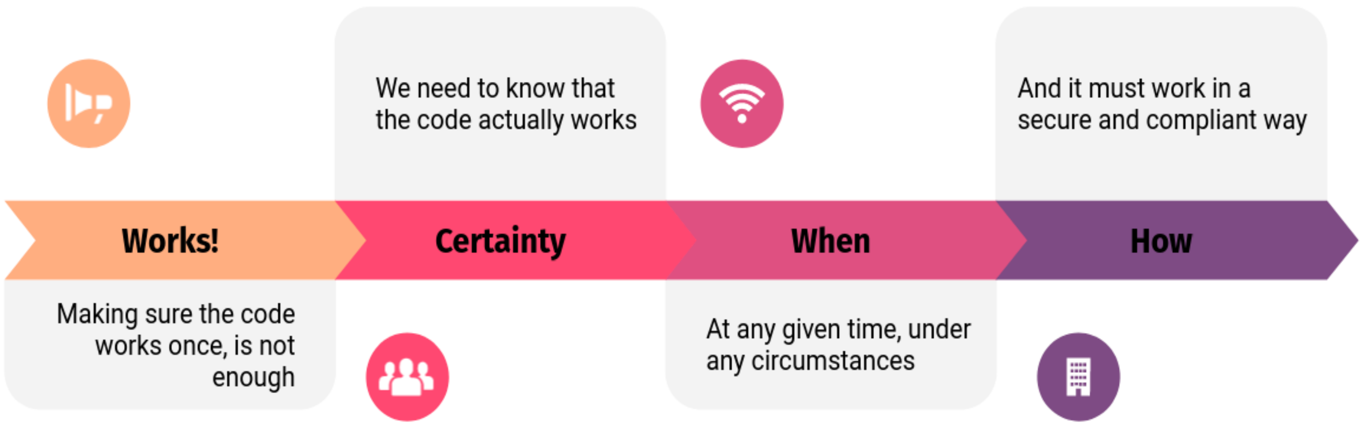
Continuous compliance conveys the meaning that conformity, and design thereof, should be constant and ongoing during all stages of software development.

Continuous compliance conveys the meaning that verification of being conformant with policies is not a task to be executed only every quarter according to internal procedures, but rather every step of the way starting from design until coding, testing and qa.

[1]https://www.researchgate.net/publication/335809902_Role_of_Testing_in_Software_Development_Life_Cycle
[2] https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

2. The evolution of code certainty

Even today it's more often than not for product owners to merely ask developers if the feature is working – a rather doomed question, because the answer is always yes, of course it is working. Any developer's code always works, or at least it works on the developer's machine, at the time, prior to scrutinous testing, and according to the understanding of the specs which the developer has.


Works!
Making sure the code works once, is not enough
Certainty
We need to know that the code actually works
When
At any given time, under any circumstances
How
And it must work in a secure and compliant way

Instead of focusing on whether or not the code currently works, we need to come up with data and metrics that will help us determine if the code can work at any given time. Additionally, we must shift our attention and efforts to being able to make sure that the code continues to work at any later time, as well as confirm that the way it works is the expected way, ie in a compliant and secure[3] way. These can all occur using tools, techniques and methodologies well-known to the software development world, which have their roots in the fundamental concepts of testing and Quality Assurance. Ensuring the above, helps provide a "healthy" software, i.e. a robust system, and significantly higher chances of ending up with a maintainable, easy to understand, more secure and compliant software.

[3] https://www.cisa.gov/uscert/bsi/articles/best-practices/measurement/measuring-the-software-security-requirements-engineering-process

The phases of software development from a robust perspective

Generic steps to go through during software development in regards to code robustness:

1. The coding occurs and the developer confirms the code works
2. The code works and someone tests it manually to confirm it works [manual testing]
3. The code comes with tests, so that even if we make changes to the code base we know that it still continues to work today [this is obviously relevant only for the parts of the code and the scenarios that are being tested in our automated testing suite]. The higher quality of testing we perform, the higher level of certainty we achieve that our code works in the desired way [automated testing]
4. The code works, we have a high level of certainty about it, and we have taken into consideration security and relevant policies during coding and testing, e.g. our test suite includes scenarios as per the regulations that limit the scope of the business.

Following the above, are some of the should-be mandatory best practices for any software development team. More details on how to build software-as-a-service can be found in the twelve-factor app[4] where best practices to achieve continuous processes are outlined – that includes building robust applications, potentially more secure software (following the 12 factor app practices is part of the SSDLC – Secure Software Development Life Cycle), as well as compliant software by default and by definition.

Moving forward we should clarify certain terms and keywords, in order to gain more insights to what each item is about, and how it helps us achieve our end-goals. That includes the terms of continuous and CI, as well as diving into the "why" we should opt for continuous process instead of manual and on-demand actions.

[4] https://12factor.net/

Why continuous?

Continuous processes encourage automation, and in turn their combination improves quality due to multiple factors. It allows for reproducible results and checks, as well as to run those checks in multiple environments. At the same time it saves tons of time from manual labor from a more traditional approach with QA and ensures that error-prone humans do not affect the result much – humans still write the code of the checks, but at least once something is verified to work, it continues to perform as expected and make sure everything works at all times (continuously).

What is Continuous Integration?

What is continuous integration anyway? Continuous integration, or in short CI, is the "practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run"[5] While running tests and builds is not necessarily part of the CI, it's definitely a good practice and nowadays it is implied that once you merge code, a set of actions will follow, such as executing tests, running linter checks, making sure builds are successful, etc.

The CI process

Continuous integration is only the beginning. And it triggers questions like: What else should be part of this process? What else can we use our CI system for? Well, while that depends on the application, using the important rule of debugging "Make no assumptions", the CI process can include anything we assume is true. That should be mostly covered in our test suite if we have a nice and organized test suite, however there are cases where additional CI jobs can be included to eliminate or reduce the need of manual checks after updates, perform security-oriented tasks and checks, and execute compliance-related evaluations. This is beneficial or even critical depending on the nature of the application. The process of CI is no longer just about merging code, it's a whole concept of processes that we must follow to stay on top of things. The traditional definition of CI and automated testing leaves a lot to be desired for pretty much any modern application, and it is even inadequate for any software that must be monitored more closely to stay compliant with laws and policies.

[5] https://aws.amazon.com/devops/continuous-integration/

Continuous Compliance

According to “Continuous compliance” article[6] continuous compliance is proposed as a means of guarantee that the codebase stays compliant on each code change. Continuous compliance increases assurance and reduces costs.

The testing starts with having some kind of tool to run tests with[7] -- ideally automatically as part of the Continuous Integration process which would help us move towards a more Continuous Testing era and support the teams for a Shift Left approach -- even if not TDD (Test Driven Development) itself is in place, it certainly provides a great insight to the code, and consequently to the software being developed, in terms of what is actually working and what not.

Moving away from more traditional flows for testing, the next logical step would be to incorporate security requirements in the test process and implement relevant checks. That way we will ensure a higher certainty that our software remains secure[8].

Ideally that helps us build a high level of certainty that each new code push is still keeping us compliant!

Continuous compliance could be seen as an umbrella of all the above, in the essence that we want to be compliant with the architectural requirements of our project – even after a certain feature is fully developed, we certainly do not want the next code push to slightly break that functionality! – and remain compliant with not just policy and legal requirements but also security aspects.

The term, continuous compliance, embraces a lot of its predecessors such as CI and everything it stands for today, testing and the idea of continuous, as in automated tasks running continuously. More on benefits of continuous practices can be found in the next paragraphs.

[6] “Continuous compliance” by Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. In ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering, (Melbourne, Australia), Sep. 2020. https://homes.cs.washington.edu/~mernst/pubs/continuous-compliance-ase2020-abstract.html

[7] See also: Winchester, Hilary. (2018). How quality assurance codes change: beyond ‘bells and whistles’ and ‘code by catastrophe’?. Quality in Higher Education. 10.1080/13538322.2018.1460900.

[8] https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=34698

Why must it be continuous?

The process of continuously checking and verifying comes with a plethora of advantages on multiple aspects:

a. Not human error prone
b. Company reputation
c. Cost reduction
d. Verify everything, not just sample

Continuous practices make action not human error prone which is already a long way to ensure results, as well as the organization's reputation which depends on those findings. The cost reduction from running processes in an automated way is a crucial factor to selecting this approach – which helps save money, time, and effort. Last but not least, it is equally important to keep in mind that with automated processes we can run and re-run cases and test scenarios at any time. Additionally, it makes it realistic to check all items, instead of relying on the tests of sample data!

Automated and manual QA

Quality assurance in the modern world

Manual testing is an integral part of the software life cycle, and is heavily performed even today, usually within a dedicated team of QA engineers. However, it should not be implied that manual testing could in any way substitute automated testing and the CI process. On the contrary, it is complementary to the automated test suite and aims to reveal additional issues, such as UX issues that prohibit users from having a smooth experience with the software, or, for a website, provide an extensive check of visual elements and how they integrate to the overall feeling of the UI. Everything else that may still be tested manually could and should be incorporated within the automated testing suite.

One of the drawbacks of the manual process is that it's usually performed after the code has been implemented and possibly even merged. By that time, it's very possible that the developer has moved on to another task or even project, and going back to solve an issue requires significantly more time than it would if the issue was detected during development or immediately after it. More on the shift-left approach and the cost of bug fixing can be found in the next chapters.

Having an automated test suite that runs before any change is integrated in the code base is crucial to the success of any project. Developers, now more than ever, focus on writing tests and implementing a CI/CD process within their project. This provides a structured and organized way of handling the needs of software development in any given project, be it a small one or a set of multiple repositories/projects that need to interconnect and be deployed to multiple environments.

Nowadays, every framework for software development comes with testing tools which can be used to represent the functionalities of the software in an, often, offline environment with predetermined test scenarios. The quality of the test suite is as good as the available scenarios. And testing can - and should - be part of any agile[9] methodology a software development team is following.

For more details about Automated VS Manual testing, and why not just go manual, check the article "Automated vs Manual Testing: Which Should You Use, and When?" in the next chapter.

[9] https://www.researchgate.net/publication/348992540_A_Comparison_Study_of_Software_Testing_Activities_in_Agile_Methods

3. Literature review

For the purposes of this thesis, multiple articles and posts were studied to provide multiple aspects of the issues at hand, which industry and teams face. Below is a selection of such articles that contribute to the topics discussed later on. The following articles are also evaluated per different criteria.

Evaluation criteria

Here are the evaluation criteria and the reasoning behind their choice:

1. Use of scientific methods to conduct research and determine results

   The validity of the methodologies with which the research is performed is of crucial importance to determining its effectiveness and the quality of the results.

2. Use of open source tools

   Using tools which are available to everyone not only enables peer review and verification of a paper or research but also makes it more approachable to everyone – students, researchers, contributors, startups – making it possible for anyone to put it in use for their own situation. Thus making the methodology broadly available, also makes it more beneficial to the community.

   Moreover, using open source tools not only gives back to the community but allows us to have a higher level of certainty regarding the validity of our results, since we can have a deep understanding of the tool used to provide us with those data – which data may drive important decisions.

3. Use of established methods and standards

   Using an acknowledged methodology adds validity to the research itself as well as its results, and in the same time helps grow awareness.

4. Real-life application and/or scenarios that apply to the unique needs of an industry

   Being able to measure how compliant we are, can be critical for organizations. Research based on realistic challenges and issues which an industry faces, can truly make a difference and have a great deal of impact.

5. Direct relation to organizations and teams that are involved with QA and compliance issues

Compliance is very important to businesses as well as other forms of organizations. Articles that address the technical aspect of things, provide an invaluable help to those organizations to put in practice the content of the research, instead of merely being presented with theoretical data.

6. Detailed methodology

   How easy and effectively can someone reproduce the research / methodology used in order to produce results for another use case. Step-by-step analysis of the action plan makes it easy for others to reproduce and apply the same method, in order to validate the results and find use of it to other scenarios by any team that wishes to employ the same methodology in its own industry.

Summary and evaluation of articles

In the next few pages we proceed with the evaluation of selected articles, providing a short summary, necessary details of the article, the evaluation rating based on the aforementioned factors, as well as relevant remarks as to why the rating was evaluated as such.

How to compare and exploit different techniques for unit-test generation

Bacchelli, Alberto & Ciancarini, Paolo. (2009). How to compare and exploit different techniques for unit-test generation.

Focused on unit-testing specifically, it presents a comparison between different tools for unit testing (JUnit, JCrasher, Randoop) which generate unit tests automatically.

The article claims to provide a "novel comparison methodology that can be used to analyze the effectiveness of different unit-test creation techniques", and at the same time provide critical details about measuring test coverage. Aside from the useful tool comparison provided within this article, there are very important points made as to how we measure the quality of our test suite – and it is not by coverage. "Coverage can be used as a negative metric" to showcase the parts of the code which are not tested. However, when it comes to how much we have actually tested, we are better off using other metrics such as the percentage of methods that are included in the test suite (method coverage) or whether or not we test both values (true or false) of a boolean condition (decision or branch coverage).

It also focuses on how to get started with testing, such as add test cases for defects or start testing with leaf classes or functions, which are easier to understand and all classes that are using them can benefit from them being tested. As a next step it is also interesting to read more about automation in testing[10]

Evaluation Details

Use of scientific methods	★★★★☆
Use of open source tools	★★★★★
Use of established methods and standards	★★★★★
Real life application and scenarios	★★★★★
Direct relation to relevant organizations and teams	★★★★★
Detailed methodology	★★★★☆

- There is an organized and detailed approach to the issue
- All tools used are open source including the operating system and machine software
- The article uses existing tools and methodologies
- The testing takes place on an existing codebase
- The results are directly related and of interest to any Java software development team
- The process is documented step-by-step providing details on how each stage is implemented

[10] https://www.researchgate.net/publication/362517283_Automation_Testing_In_Software_Organization

A comparison between manual testing and automated testing

Kumari, Bhawna & Chauhan, Naresh & Syed, Habeebullah Hussaini. (2018). A COMPARISON BETWEEN MANUAL TESTING AND AUTOMATED TESTING. SSRN Electronic Journal. 5. 323-331.

A fundamental and easy to understand comparison between manual and automated testing, including listing the benefits and presenting the characteristics of a good test case.

The article provides an overview of the fundamentals around testing. It outlines different characteristics of good test cases and how they affect the quality of our code, focusing particularly on unit testing. Additionally it lists a variety of benefits that come with automated testing, in contrast with manual testing, such as no human interaction being needed for the test execution, better speed due to lack of human factor - in fact execution can be 70% faster. Except for those rather obvious arguments, a few other advantages of automated testing are accuracy of results and reliability of the test case being executed as it was designed. Automated testing also offers a wider test coverage of application features, even though coverage is not necessarily the best metric to measure the effectiveness of our test suite.

Evaluation Details

Use of scientific methods	★★☆☆☆
Use of open source tools	★★★★★
Use of established methods and standards	★★★★☆
Real life application and scenarios	☆☆☆☆☆
Direct relation to relevant organizations and teams	★★★☆☆
Detailed methodology	★★★☆☆

- The article is focused on automated testing methodology which is not itself a recognized scientific method
- An open source tool is used for the automated testing
- The tool used is a common choice for such kind of testing among software developers and QA engineers
- The paper is not based on any specific real scenarios
- The results directly affect software development teams
- While there are steps of how to use Selenium, the tool for automated testing presented in the paper, it is more theoretical than practical enough to be immediately applicable

A comparative analysis of quality page object and screenplay design pattern on web-based automation testing

D. Yuniasri, T. Badriyah and U. Sa'adah, "A Comparative Analysis of Quality Page Object and Screenplay Design Pattern on Web-based Automation Testing," 2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE), 2020, pp. 1-5, doi: 10.1109/ICECCE49384.2020.9179470.

The article presents two kinds of design patterns used for automation testing and evaluates each one of them according to a variety of metrics (flexibility, reusability, functionality, understandability, extensibility, effectiveness). The research applied the QMOOD (Quality model of object oriented design) approach to get quality value from each design pattern. The results presented in this paper identify the object design pattern to present with better quality for metrics of functionality and effectiveness, while screenplay design has better quality for reusability, flexibility, understandability, and extensibility.

Evaluation Details

Use of scientific methods	★★★★☆
Use of open source tools	★★★★★
Use of established methods and standards	★★★★★
Real life application and scenarios	★★★★☆
Direct relation to relevant organizations and teams	★★★☆☆
Detailed methodology	★☆☆☆☆

- There is a solid process followed to conduct the comparative analysis and reach conclusions
- Prestashop is used as an example, which is open source code available to anyone
- The paper is based on existing methods and processes
- While there is no explicit reference to the prestashop instance that it was used, the article's test scenarios are clearly based on real life use cases
- The results' usefulness might be limited to users of that specific software, however the metrics provided are certainly of interest for any team, organization and kind of software
- The methodology is not immediately applicable to another software and it is not clear if and how it could apply to another industry

Automated Testing of AI Models

Haldar, Swagatam & Vijaykeerthy, Deepak & Saha, Diptikalyan. (2021). Automated Testing of AI Models.

The paper is a follow-up on "Model-Based Test Modeling and Automation Tool for Intelligent Mobile Apps"[11] and it uses the tool AITEST and adds testing capabilities for tabular data, image and speech-to-text models aiming to make the tool a comprehensive framework for testing AI models. The additional functionalities include a) interpretability testing of tabular AI models, b) comprehensive sets of image transformations for testing black-box image classifiers, and c) a comprehensive set of audio transformations for testing fairness and robustness properties for speech-to-text models. The suggested implementation is limited to black-box testing which needs to be configured but can then be re-used to other similar models.

The tool is part of IBM's Ignite quality platform.

Evaluation Details

Use of scientific methods	★★★★☆
Use of open source tools	☆☆☆☆☆
Use of established methods and standards	★☆☆☆☆
Real life application and scenarios	★★☆☆☆
Direct relation to relevant organizations and teams	★★★★☆
Detailed methodology	★★★★☆

- The article provides definitions of parameters used and provides detailed results.
- It is not indicated whether or not the tool is open source, or how it can be used by others
- The tool is a new innovative approach to the issue
- While it aims to address very realistic scenarios, the article itself does not present a real life use case
- It directly affects the AI industry
- It provides a lot of information about the process that was used

[11] https://ieeexplore.ieee.org/document/9564374

Automated System-level safety testing using constraint patterns for automotive operating systems

Choi, Yunja & Byun, Taejoon. (2015). Constraint-based test generation for automotive operating systems. Software & Systems Modeling. 16. 10.1007/s10270-014-0449-6.

Automated test generation for safety testing of automotive operating systems. The constraint-based method of testing presented an advantage for testing illegal behaviors.

The automotive industry is a representative example of a typical safety-critical system. According to the article, existing techniques do not focus on interface testing of such operating systems, which is the gap that it aims to fill, using operational constraints defined in the specifications, to produce configuration-dependent and state-dependent constraint patterns. The existing methods of testing present important disadvantages, namely concolic testing does not explicitly test illegal behaviors, and OSEK OS scenario- and specification-based testing checks the functional correctness but not the robustness of the system in case of illegal behavior. For the industry it is imperative to be testing the safety of the system in any environment, and include both illegal and correct behaviors. The experiment included generated examples for all constraint states executing both legal and illegal calls. The constraint-based testing method increases the efficacy of testing for failure detection by systematically executing test cases related to illegal behaviors.

The article presents an example of a system and an industry that requires robust testing techniques, and showcases that even if testing is being performed, existing methods may or may not be suitable to cover all industry and system needs.

Evaluation Details

Use of scientific methods	★★★★★
Use of open source tools	★★★★☆
Use of established methods and standards	★★★★☆
Real life application and scenarios	★★★☆☆
Direct relation to relevant organizations and teams	★★★★☆
Detailed methodology	★★★☆☆

- The paper shows a systematic and scientific approach to conduct the testing
- The software used, trampoline, is the open-source implementation of OSEK OS, according to the article, however it is not clear in the article if non open source tools are needed as well

- The paper depends on established methods and procedures
- This is directly related to automotive industry, however the paper itself is not based on a specific real case
- This would be of importance to relevant teams of this particular industry
- The steps are well documented, but results are not immediately reproducible

On the Challenges of Automated Testing of Web Vulnerabilities

L. F. de Lima, M. C. Horstmann, D. N. Neto, A. R. A. Grégio, F. Silva and L. M. Peres, "On the Challenges of Automated Testing of Web Vulnerabilities," 2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2020, pp. 203-206, doi: 10.1109/WETICE49692.2020.00047.

Evaluation of 5 penetration testing tools against 7 web applications for detention and exploitation of vulnerabilities, using the PTES (Penetration TEsting Execution Standard) methodology.

This IEEE WETICE article sheds a light into the workflow of penetration testing (pentesting). To achieve this 5 tools were tested across 7 web applications to document both the tools as well as the challenges which testers face. The methodology used is PTES (Penetration Testing Execution Standard), according to which pentesting tools perform two functionalities: scanning and exploitation. Scanning aims to identify the vulnerabilities in the application under test and exploitation allows testers to gain access to the application (by exploiting one or more vulnerabilities detected in the first step). Results are publicly available in https://github.com/lfl-repository/PestestingReports. According to the OWASP edition from 2013 to 2017 the most frequent vulnerabilities are those related to input validation. The vulnerabilities chosen for this article's research were the OWASP top 10 from 2017

The importance of pentesting lies with *assurance* on software security. Pentesting is a widespread technique to determine the level of security of systems and applications, as well as provide relevant data and statistics for an organization's security level and potential vulnerable points. In multiple cases, penetration testing is also a requirement, either enforced by law or internal procedures, to determine the readiness of an application for production or confirm its security over time.

Evaluation Details

Use of scientific methods	★★☆☆☆
Use of open source tools	★★★★★
Use of established methods and standards	★★★★★
Real life application and scenarios	★★★★☆
Direct relation to relevant organizations and teams	★★★★☆
Detailed methodology	★★★★☆

- The paper presents a practical approach for verifying tools' effectiveness
- All tools and methodologies used are open source and widely available

- The methodologies and tools used are known and used in the industry
- The paper is based on testing applications which exist for the purpose of testing, it would be interested to perform the same pentest in real applications
- The vulnerabilities being tested are realistic and existent in modern software, thus the results are highly related to any software development team
- The paper provides a comprehensive analysis of the steps followed, allowing replication of the process

Towards Software Compliance Specification and Enforcement Using TOSCA

Mubarkoot, Mohammed & Altmann, Jörn. (2021). Towards Software Compliance Specification and Enforcement Using TOSCA. 10.1007/978-3-030-92916-9_14.

“.. the size and complexity of software systems continue to increase over time and, simultaneously, if not maintained rigorously, the quality decreases” at the same time changes in regulations and policies add extra complexity with compliance management. The article provides a methodology for handling changes in policy documents and industry requirements and stresses the importance of compliance management throughout the software development lifecycle (SDLC).

Evaluation Details

Use of scientific methods	★★☆☆☆
Use of open source tools	★★★★★
Use of established methods and standards	★★★★★
Real life application and scenarios	★☆☆☆☆
Direct relation to relevant organizations and teams	★★★☆☆
Detailed methodology	★★☆☆☆

- A scientific method seems not to be applicable in this case. The overall presentation is methodic.
- All of the tools used are open source
- An existing framework and well-known tools are used
- The paper did not present details of the specific business and software it was used
- The issue at hand as well as a possible solution directly affects engineering and compliance teams, and a variety of different organizations
- A schematic representation is offered in regards with the example topology, but more details would be required for application to another software

Acquiring Software Compliance Artifacts from Policies and Regulations

Breaux, Travis & Antón, Annie. (2022). Acquiring Software Compliance Artifacts from Policies and Regulations.

Provides a methodology to extract technical requirements from legal and policy documents. Explains the problems which developers face when implementing compliant software and outlines the necessity for an organized method to identify and classify such compliance requirements.

“..organizations must be able to demonstrate that they have verifiable procedures in-place to implement the restrictions imposed (..)” while the software engineers face the challenge of reading, understanding, interpreting and extracting rules and test scenarios out of policy and regulatory documents written primarily by domain experts in non-engineering fields (law, medicine, finance, etc). The methodology presented in the article provides technical people with a process for extracting requirements artifacts regarding the actions and obligations which are permitted. The article aims to “provide auditors with a reproducible and certifiable chain of evidence that shows how software systems comply with the law”, through analyzing the language in such documents and creating relevant restrictions and rules which developers can understand. Additionally it helps stakeholders discern between discretionary and mandatory requirements for their software.

Evaluation Details

Use of scientific methods	★★★★★
Use of open source tools	-
Use of established methods and standards	★★★✩✩
Real life application and scenarios	★★★★★
Direct relation to relevant organizations and teams	★★★✩✩
Detailed methodology	★★✩✩✩

- There is a detailed analysis of the method used
- It is not clear which tools were used to implement what’s described in the article
- The article depends on previous research of the same group of people
- The use case provided is of HIPAA policy

- While that’s realistic to the challenges a lot of organizations and teams face when it comes to policy documents, it is quite specific to the US health sector
- There are details provided in theoretical level, however it is unclear how to practically apply this to another policy document

Software Security Requirement Engineering for Risk and Compliance Management

D, Kavitha & .S, Ravikumar. (2021). Software Security Requirement Engineering for Risk and Compliance Management. International Journal of Innovative Technology and Exploring Engineering. 10. 11-17. 10.35940/ijitee.E8628.0210421.

Mapping of security requirements with compliance mandates in an effort to document and visualize their connection aiming to reduce the risk of non-compliances. The article also provides useful explanations and definitions for security terms and concepts.

Security requirements must be defined and handled at the early stages of the software development lifecycle. The article provides an object mapping for risk assessment and compliance management based on category theory and the CORAS risk assessment modeling technology.

Evaluation Details

Use of scientific methods	★★★★☆
Use of open source tools	-
Use of established methods and standards	★★★★☆
Real life application and scenarios	★★★★☆
Direct relation to relevant organizations and teams	★★★☆☆
Detailed methodology	★★☆☆☆

- The article shows a methodical approach to the issue
- It’s unclear which tools were used for this analysis
- The article depends on category theory and the CORAS modeling technology. It would be interesting to see which other tools were used
- The modeling is applied in safety-critical and security-critical software
- The results could benefit other teams and organizations
- It is not clear how we can apply this to another use case

Bug Prediction Capability of Primitive Enthusiasm Metrics

Gál, Péter. (2021). Bug Prediction Capability of Primitive Enthusiasm Metrics. 10.1007/978-3-030-87007-2_18.

Explores the use of Primitive Enthusiasm metrics (method based metrics) to improve bug prediction. The metrics measured

While bugs are unavoidable in software, finding them can be tricky and also ineffective, particularly if only manual methods are used. The goal of the research is to determine if primitive enthusiasm (PE) metrics[12] improve cross-project and across-versions bug prediction. Data types can be primitive (boolean, integer, char, etc) or complex (classes, structs, etc). “Primitive Obsession is the excessive use of primitive data types. The programmer does not create small objects for small tasks, instead s/he is obsessed with the use of primitive data types.”. According to the results, PE metrics can be helpful in identifying bugs and their use is certainly viable and worthwhile in both cases, as “(Primitive obsession is) a symptom for the existence of overgrown, chaotic code parts”

Evaluation Details

Use of scientific methods	★★★★★
Use of open source tools	★★★☆☆
Use of established methods and standards	★★★★☆
Real life application and scenarios	★☆☆☆☆
Direct relation to relevant organizations and teams	★★★☆☆
Detailed methodology	★★☆☆☆

- There is a solid approach to the issue
- Publicly available datasets were used
- The paper is based on scientific and statistical methods
- The algorithm was not directly used on business software running on production
- The results could provide useful insights across other teams
- The analysis is not immediately applicable to another software

[12] Primitive Enthusiasm is a metric designed to highlight possible Primitive Obsession infected code parts – Edit Pengo and Peter Gal, Grasping Primitive Enthusiasm Approaching Primitive Obsession in Steps, https://www.scitepress.org/papers/2018/69188/69188.pdf

Code Smells and Detection Techniques: A Survey

Menshawy, Rana & Hassan Yousef, Ahmed & Salem, Ashraf. (2021). Code Smells and Detection Techniques: A Survey. 78-83. 10.1109/MIUCC52538.2021.9447669

"Refactoring Code smells help the developers to improve the code quality in a significant way"

The paper makes reference to the different detection approaches (manual, history-based, fully automated, etc.) and concludes that the majority of the available tools support Java, and so are the most popular datasets. ML algorithms are widely used, however they depend on those training datasets, restricting research around a specific programming language. An interesting point brought up for further investigation is how refactoring a code smell can in fact introduce a new one, and the relation between them. Also see 18.

Evaluation Details

Use of scientific methods	★★★★★
Use of open source tools	★★★★☆
Use of established methods and standards	★★★☆☆
Real life application and scenarios	★☆☆☆☆
Direct relation to relevant organizations and teams	★★☆☆☆
Detailed methodology	★★★★☆

- The article provides a literature review with specific hypotheses in mind
- Tools used to produce this article are accessible
- There aren't any standards per se, but there is an organized representation of information
- This being a literature review, it does not directly correlate to an industry, organization or specific software
- Insights about causes of bugs and ways for better and faster detection are certainly useful
- The article presents a detailed set of steps and good documentation of findings

Worst Smells and Their Worst Reasons

D. Falessi and R. Kazman, "Worst Smells and Their Worst Reasons," 2021 IEEE/ACM International Conference on Technical Debt (TechDebt), 2021, pp. 45-54, doi: 10.1109/TechDebt52882.2021.00014.

Survey with developers to identify details surrounding the creation of technical debt. Provides a statistical analysis and its results for 5 hypotheses.

“Code bad smells are symptoms of poor design and implementation”[13] and more often than not software development teams are not mindful about technical debt, don’t measure it, don’t acknowledge it in retrospectives, and consequently do not address it.

“Similarly we know that developers (intentionally or unintentionally) violate design and coding rules and best practices, and so technical debt inevitably accumulates over time, making the system harder to maintain”

The article aims to identify worst code smells, i.e. bad smells that never have a good reason to exist, determine the frequency, change-proneness and severity associated with the worst smells and last but not least identify the worst reasons for introducing them. The survey included 71 developers across multiple enterprises. And its findings present that 80 out of 314 cataloged code smells should never exist in any code base. Possibly even worse, the survey also concludes that technical debt usually starts small and over the life of a project as new features are added they become “god classes”. Additionally referenced in the article[14], most smells are introduced at the creation of an artifact, and not as a result of its evolution – which makes you wonder if software development teams really need to get in front of this, at the very time when code is being written, and shut down unnecessary technical debt before it even comes to life. One of the interesting results of the survey reveals that there is a significant difference on severity levels between worst and non-worst smells, as well as that worst and non-worst code smells cannot be correlated to the number of changed lines of code. Last but not least, a complete replication package is provided to encourage replicability and validation of results by others (peer review)[15]

---

[13] M. Fowler, “Refactoring: Improving the design of existing code,” in
Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002. Berlin, Heidelberg: Springer-Verlag, 2002, p. 256.

[14] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk, “When and why your code starts to smell bad (and whether the smells go away),” IEEE Transactions on Software Engineering, vol. 43, no. 11, pp. 1063–1088, 2017

[15] https://zenodo.org/record/4270178#.Ye0ziFtBxhE

Evaluation Details

Use of scientific methods	★★★★★
Use of open source tools	☆☆☆☆☆
Use of established methods and standards	★★★★☆
Real life application and scenarios	★★★★★
Direct relation to relevant organizations and teams	★★★★☆
Detailed methodology	★★★★☆

- The article presents a methodical approach for survey, performing a statistical analysis and presenting results
- It is not clear if open source software was involved. The survey was based on bugs identified by sonarCloud which does not seem to be open source
- Use of well-known methods for statistical analysis, however the research is based on bugs identified by another software
- Participants of the survey were engineers and software developers found from the mailing list of IEEE Technical Council on Software Engineering (TCSE). Additionally the use of sonarCloud for the identification of bugs is an enterprise software that can be potentially used by any software team
- The results are important for any software development team as well as management
- There is a significant amount of information for the survey to be replicated, even though the results still depend on the subjective opinion of each participant. It would be interesting to compare results between senior developers, junior and mid developers as well as team leads or other management positions

A legal cross-references taxonomy for identifying conflicting software requirements[16]

Maxwell, Jeremy & Antón, Annie & Swire, Peter. (2011). A legal cross-references taxonomy for identifying conflicting software requirements. Proceedings of the 2011 IEEE 19th International Requirements Engineering Conference, RE 2011. 197 - 206. 10.1109/RE.2011.6051647.

This paper aims to document the specific challenges that arise due to cross-references in compliance requirements and which directly affect engineers. According to the article there are 6 different types of cross-references which were identified in the research using HIPAA (US Health Insurance Portability and Accountability Act). Additionally the paper offers 4 strategies that can help engineers resolve conflicting requirements:

1. Comply with the most restrictive law
2. Store data separately
3. Obligations supersede legal privileges
4. Consult legal domain experts

Evaluation Details

Use of scientific methods	★★★★★
Use of open source tools	★☆☆☆☆
Use of established methods and standards	★★★☆☆
Real life application and scenarios	★★★☆☆
Direct relation to relevant organizations and teams	★★★☆☆
Detailed methodology	★★★★☆

- The article uses scientific methods to analyze and answer the hypotheses posed
- No open source tools mentioned
- The article is based on solid methodologies
- The text analyzed is the US HIPAA, which covers one certain industry
- It is unclear if it can be directly applied to other kind of policies
- While there is a detailed analysis of the methodology, the algorithm is not available for re-use

16 https://www.researchgate.net/publication/224263869_A_legal_cross-references_taxonomy_for_identifying_conflicting_software_requirements

What Is Technical Debt And Why QA Testers Should Be Concerned About It?

https://www.softwaretestinghelp.com/technical-debt-and-qa/, January 2022

"Most QA managers impulsively view tech debt as the reasonable consequence of focusing all your energy on the current sprint alone, which leads to achieving the test coverage somehow through manual means, and completely ignore automation. This is known as the quick and dirty approach which has been covered in a blog by Martin Fowler, the author of the technical debt quadrant."

The article provides a comprehensive explanation around the causes of technical debt and the challenges that surround it. As a definition it mentions "Technical debt is the difference between what is expected and what is delivered". Additionally, it presents six (6) reasons why technical debt is created, one of which is the absence of continuous integration. Even with automated testing, in their example they identified that there was only 40% coverage and the technical debt resulted in 60% increase of effort for testing, in increased costs for hiring testers to support the project, slow release testing and difficulty in keeping the test cases up to date as the project grew more complex. Last but not least, it emphasizes that delivering too many features (more than the team's capacity) inevitably results in technical debt which will need to be addressed, and for that the team must "slow down to start covering the technical debt it left behind". The reasons presented as the root cause behind the creation and propagation of technical debt are:

1. Improper documentation identified in the article is indeed a common denominator in miscommunication, all sorts of issues, often even delay in release of features
2. Inadequate testing and bug fixing – these are both items often overseen and forgotten by teams and particularly team leads and other management members
3. Lack of coordination between teams, which can lead to delays in deploying software, preparing new infrastructure in time, and have a common understanding of what we are building and what are the next steps moving forward
4. Legacy code, which can grow in volume and technical debt and increase the improvement effort needed
5. Delayed refactoring, for which one of the reasons could be old legacy code which may or may not be needed or understood by the current product and team
6. Absence of continuous integration, so that we know each step of the way what's the state of our codebase and consequently our project and product
7. (more factors which the users/organizations cannot control)

Evaluation Details

Use of scientific methods	★★★☆☆
Use of open source tools	★★★☆☆
Use of established methods and standards	★★★★☆
Real life application and scenarios	★★★★★
Direct relation to relevant organizations and teams	★★★★★
Detailed methodology	★★☆☆☆

- A big part of this article is based on real life experience, still there is a basis for the claims as as well as statistics and graphs to support the data presented
- Use of tools is not relevant to this article, however its findings relate to open source projects
- There is definitely business experience involved, but the results are based on known approaches
- The article is based on real life software
- Its findings apply to a wide range of software development teams that can benefit from its insights
- While there is very useful information included, it is not immediately replicable for another use case

Automated vs Manual Testing: Which Should You Use, and When?

A. (2020, May 7). Automated vs Manual Testing: Which Should You Use, and When? Apica. [*https://www.apica.io/blog/difference-between-automated-manual-testing*](https://www.apica.io/blog/difference-between-automated-manual-testing)

A business approach on testing and which type to choose, along with a concise overview of different kinds of testing and their definition. The comparison between manual and automated testing is based on multiple factors and from a both technical and hands-on perspective. Specifically automated testing should be preferred for regression testing (particularly necessary when there are frequent code changes, which is often the case) and performance testing (in order to simulate thousands of concurrent users) while manual testing may be considered in order to facilitate usability testing (how user-friendly the interface is, or its convenience to end-users) or some kind of ad-hoc testing where we want to get the insights of the tester about an unplanned testing that needs to be executed.

The comparison factors include repetition, accuracy and human error possibility, time consumption, investment required as well as the possible need for human observation (which is best served by manual testing).

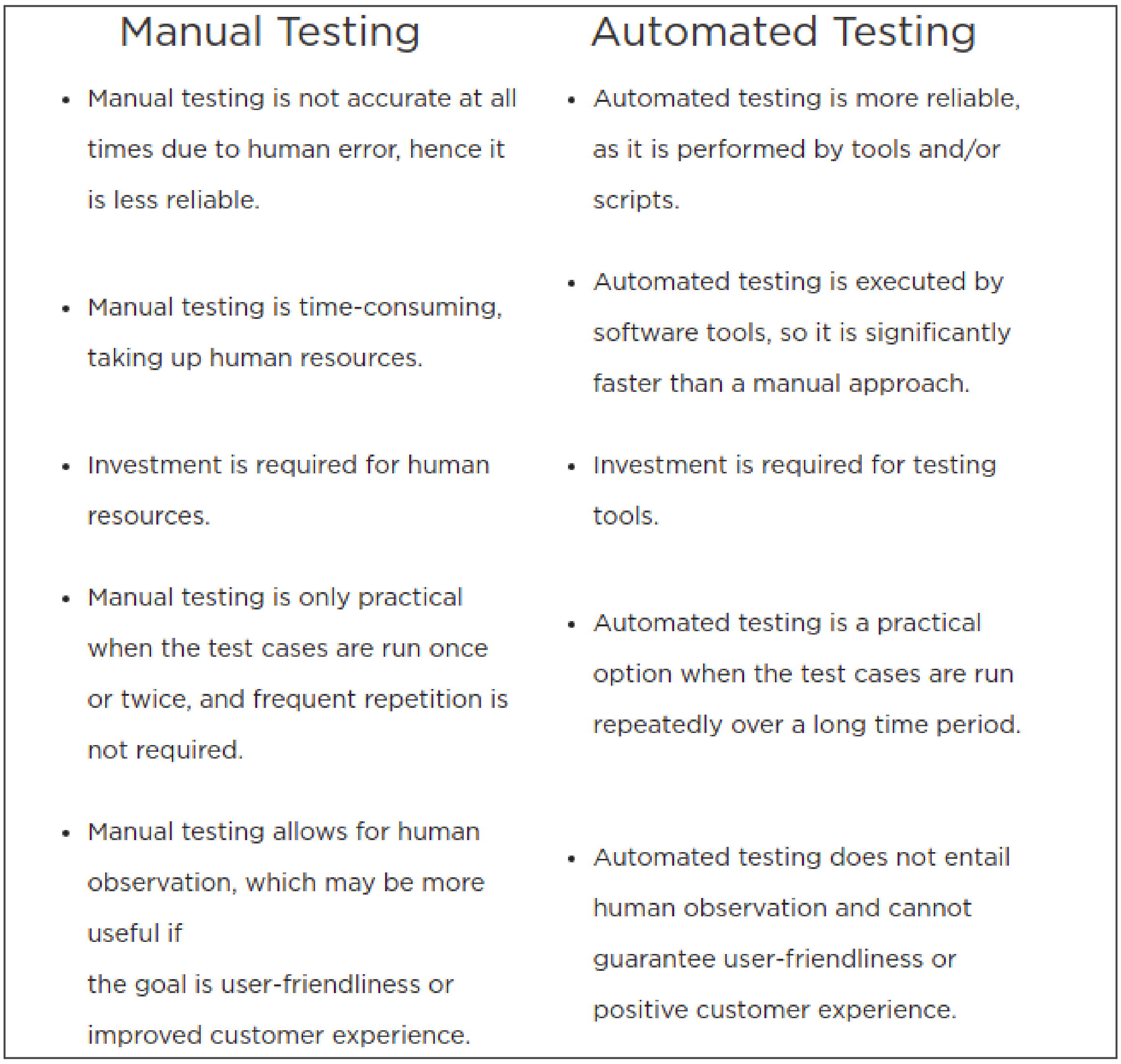
The following table from the article itself, gathers comparison factors in an effort to determine which test is best advised according to the goals and capabilities of each team and organization.

Evaluation Details

Use of scientific methods	★★☆☆☆
Use of open source tools	★★★☆☆
Use of established methods and standards	★☆☆☆☆
Real life application and scenarios	★★★★★
Direct relation to relevant organizations and teams	★★★★★
Detailed methodology	★☆☆☆☆

- The article references different kinds of testing and provides a comparison table
- Open source tools are not directly related to this article, however all findings apply to any open source tool for performing automated tests as well as open source projects

- The article provides explanations for different testing methods and their purpose, as well as the comparative analysis between them
- The findings of the article derive from real life application of testing methods
- And it can certainly help teams and organizations learn about automated testing as well as make more educated decisions about their approach to responsible software development
- The article provides insights on different testing methodologies which can be employed, but there is not a way to replicate the findings for the comparative table between automated and manual tests



Manual Testing	Automated Testing
• Manual testing is not accurate at all times due to human error, hence it is less reliable.	• Automated testing is more reliable, as it is performed by tools and/or scripts.
• Manual testing is time-consuming, taking up human resources.	• Automated testing is executed by software tools, so it is significantly faster than a manual approach.
• Investment is required for human resources.	• Investment is required for testing tools.
• Manual testing is only practical when the test cases are run once or twice, and frequent repetition is not required.	• Automated testing is a practical option when the test cases are run repeatedly over a long time period.
• Manual testing allows for human observation, which may be more useful if the goal is user-friendliness or improved customer experience.	• Automated testing does not entail human observation and cannot guarantee user-friendliness or positive customer experience.

Why Low-Code Isn't No-Test

Bridgwater, A. (2022, January 13). Why Low-Code Isn't No-Test. Forbes. *https://www.forbes.com/sites/adrianbridgwater/2022/01/12/why-low-code-isnt-no-test*

This Forbes article is based on a corporate business stand-point and discusses technical debt and automation oversight, providing data to support the necessity of automated testing and the consequences of lack thereof.

The author chooses the term "lower-code" rather than "low-code", because the amount of coding involved is not insignificant. Both lower-code and no-code techniques can be useful but potentially also detrimental to software development best practices. Mitigations to this risk include a proper policy which is enforced by the organizations that use the software, corporate compliance guidelines, and proper testing before the application goes live.

"With this rapid innovation comes the need for scalable end-to-end testing. A recent Forrester report confirms organizations using Low-Code platforms like Salesforce's Low-Code tools can't afford to ignore automated testing," insists McQueen (senior vice president of growth at Copado) and team

It's not uncommon to release software to production because there is not enough time, or manpower when it comes to manual testing, to conduct the necessary testing or structure the new test cases for the features or bug fixes implemented – but for that the quality suffers, and costs increase as defects are detected later on.

Now more than ever, with the increasing use of Low-Code/No-Code platforms and market competitiveness, teams need to prioritize their strategy for automated testing, which will in turn provide companies with the required business agility that leads to commercial success. Manual testing is no longer an option because not only is it expensive, but it also does not scale.

The article also provides useful statistics regarding software development and automated testing, such as the following:

- Teams that use automated testing have 50% more frequent releases
- Teams that use automated testing experience less than half production failures yearly

Automations applied in software development – use of AI, Low-code and No-code methodologies – should be extended in testing methodology as well, meaning a world of more automated tests.

Evaluation Details

Use of scientific methods	★★☆☆☆
Use of open source tools	★★★☆☆
Use of established methods and standards	★☆☆☆☆
Real life application and scenarios	★★★★★
Direct relation to relevant organizations and teams	★★★★★
Detailed methodology	★☆☆☆☆

- While there are some statistics offer to support the claims of the article, they don't depend on a methodical research and the article does not present details about a scientific approach
- There is no reference of open source tools used for the purposes of the article, however the topic relates to open source software
- The content of the article is based on empirical methods
- Claims within the article derive from business experience
- And can be beneficial to any other software development team
- While useful, data in the article cannot be reproduced for a different use case

4. Testing: a choice or a necessity?

Lots of teams treat testing like a choice or even a burden, rather than the key to certainty.

For a lot of teams writing tests is an extra step they need to tackle. Sometimes it's even seen more like a burden that must be addressed after development is completed, rather than a step integrated within the development process.

Even worse, it's often the case that developers don't know how to get started with testing, or what exactly to test, and maybe they are even confused by the different kinds of testing. That's already too overwhelming and testing becomes the perfect candidate to be put at the bottom of the ToDo list - and that's how some software development teams simply never get to it.

There is a simple approach to making testing less of a stress and more of a *factor of certainty*[17] – even for teams that are already working on existing projects and are in the middle of developing new features. The key here is to start small. If we think of the hundreds of different test cases we should examine, no wonder we cannot get started and our team feels paralyzed by the volume of all that information.

Moving forward in this paper there are simple, yet powerful, techniques to get every team started on testing – because in testing every little bit matters (more about this in section "Practical Testing" and the 1% in testing). Getting started with testing, as well as sticking with it are very important factors about the well-being of your software, and automated testing can directly contribute to the software being secure and compliant, with less bugs, and overall more robust. Moreover, the provided resources later on help teams and developers identify meaningful tests, actionable test cases, as well as possible areas of improvement (in the test suite itself, or in the process of testing).

[17] https://homes.cs.washington.edu/~mernst/pubs/continuous-compliance-ase2020-abstract.html

What does automated testing offer anyway?

Understanding the benefits offered by automated testing can be crucial in getting the whole team on board. Overall, the less bugs and pain-points there are for customers and the support team, the better the experience for everyone – and happy customers are returning customers.

1. Robust code

   A code that handles everything that there is to handle, provides a stable execution outcome, and an overall better product quality.

2. High quality of services

   Automated testing can help catch issues, bugs, faulty logic and inaccurate user stories before the feature hits production (and customers).

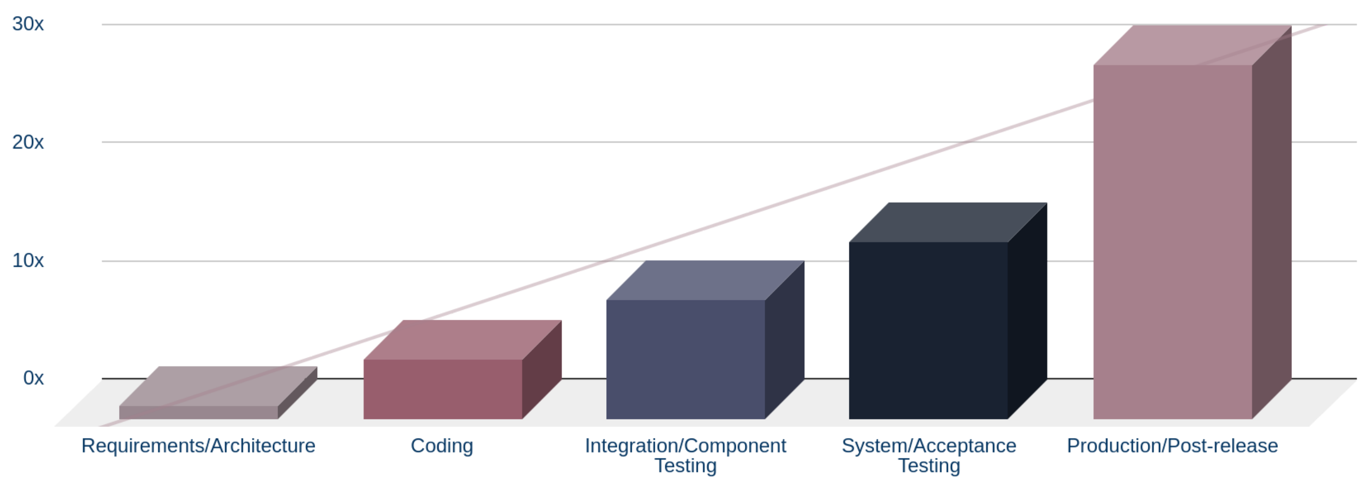
3. Legal protection

Additionally, if automated testing is done right, it already provides some level of legal protection in the essence that it provides certainty that our software is behaving as expected – assuming the design was done properly, and we raise the level of verification regarding its implementation through automated testing.

Can you afford *not* to detect bugs early?

The graph below shows the immense increase in costs as we delay discovering an issue in our software. It can be a wake up call for teams as well as managers that there is this hidden cost as the software development progresses, and if not done right the qa process – or rather acting on its results – can prove to be a lot more expensive than initially budgeted. For that reason, teams including higher management should be made aware of the risks and mitigate them as they see fit. Once the information and data is available, it's possible to make educated decisions, and foresee potential issues.

Let's take a look at the cost of bug fixing in multiple stages of the software lifecycle[18], which is part of the article "What is shift left testing" in BrowserStack's guides. The graph also appears in other articles, as it is an interesting topic that should concern multiple team members within organizations.



Relative cost to fix bugs, based on time of detection

Why does the cost increase as we move away from architecture and writing code?

There are multiple reasons that contribute to the cost going up if you need to take a look at if everything works at a later stage of the SDLC[19]. Except for the communication overhead and developer frustration which arise when a supposedly completed feature is sent back to be reworked on, more practical reasons for the development cost going up is the simple fact that

[18] https://www.browserstack.com/guide/what-is-shift-left-testing and https://deepsource.io/blog/exponential-cost-of-fixing-bugs/

[19] https://www.researchgate.net/publication/255965523_Integrating_Software_Assurance_into_the_Software_Development_Life_Cycle_SDLC

1. Refactoring an existing code means we once spent time writing the code, and now we will spend more time to re-write that code, essentially doing double work if that's a new feature -- and we may even need to go into a complete do-over and rework the feature from scratch!
2. The time it takes to bring our new feature into production, i.e. to our customers, is significantly higher when we need to return back to the first steps of the process. That alone can cause customer dissatisfaction

When is it too soon to deliver a new feature?

Code that works? Or code that works well?

"It's never too early to deliver your task" is possibly what many, including higher management, would say. And while it holds some truth to it, particularly when we are working within iterative processes with a well-defined roadmap and planned milestones, when it comes to certainty and compliance it couldn't be further from the truth. Delivering too soon most probably comes with shortcuts and omissions -- of tests, comments, best practices, etc. And what gets really delivered is a partial task and a whole bunch of technical debt.

Time is not the only factor that matters, unless people are literally dying. Unless you are part of a team which has consciously decided to ship something at a very early stage on purpose – such as a startup which merely wants to showcase a concept and not actually release it – there are some interesting questions to be aware of, before deciding the software is "good enough" or that the deadline has passed. When extra time is needed, it's of the utmost importance to evaluate what has happened with the extra time, even if it was over the estimated time of the task. Was that time dedicated to reviewing the code one more time, or trying to see how everything glues together after an important change? Did it provide meaningful insights to the feature via documentation and code comments? Were useful tests written?

However less tangible or seemingly less countable, these are the things that can contribute to the well being of software and help the whole team in the long run. Merely seeing something working as expected, is not enough to determine how well-written the code is, or how many assumptions are included. And since making an assumption is the root cause of most bugs, being mindful about all aspects of the feature, not just time until delivery, should probably be very high in priority for everyone.

Does the next day matter? The next day, after the code gets merged. That's production deployment. And where any and all well-hidden bugs will surface. **Quality over speed** might in fact result in faster outcomes.

5. Practical testing

“...timely delivery of assigned tasks should not be the sole criterion” – Chaitanya Belwal, Ph.D.[20]

What to test

A thorough enumeration and explanation of all the different test types is beyond the scope of this paper. What we will focus on is the quality of the tests, their traits and structure which build a meaningful automated test suite designed to protect your software.

Each function added to the software must be accompanied by a unit test. Unit tests apart from confirming the function is working as expected, they also provide a good insight on how the function works, what it implements, and offers specifics around its input and its output. While code comments should already make this clear, often going through a test case and seeing example values of what goes in and what result is to be expected, it’s a very helpful process to gain deeper knowledge about the method and its purpose.

Unit tests are only one of the many kinds of tests that can - and should - be incorporated into any software. Each kind of testing provides additional certainty as well as a different approach to the issue at hand, for example “Black box testing is conducted to evaluate the compliance of a system with specified functional requirements and corresponding predicted results.“[21]

Adding integration and end-to-end tests can help identify issues with the communication between parts of the application, or use of other services within our codebase. After all, having a function that works properly – tested via unit tests – which, however, cannot be called from other parts of our application, is still problematic and should definitely be fixed, sooner than later. Besides, building a function is only relevant if it can be useful somehow – such as incorporated into our codebase, or used by our software and our team. Unless it provides some kind of value to the team, the software we are building, or the product we are shipping, it may be great practice developing that code but it is not relevant to the current common goals of the team.

[20] https://www.linkedin.com/pulse/think-you-have-rock-star-your-software-team-chaitanya-belwal-pmp-csm/

[21] https://www.researchgate.net/publication/268419508_Different_Approaches_To_Black_box_Testing_Technique_For_Finding_Errors

How to test

Explicitly

Tests must be specific. It's not too hard to get derailed when writing tests, in fact it is fairly easy to end up testing that the truth is in fact true. Which is why tests must come with explicit values. Do you need a user record? Create one. You need to make sure the function returns the correct email address? Create a user with a specific email address, and use that value within the test.

Organized

Tests must be structured. And DRY still applies. Tests can be grouped when they address a certain case, e.g. when an error is raised, when the user has a specific role (admin, guest, etc.)

How to get started with testing

Testing is a factor of certainty. And if testing is a factor of certainty that our code works as expected, Compliance QA is a factor of certainty that our code and product is compliant with the legally binding regulations and policies our organization must adhere to.

Whether for a solo software developer or a whole engineering or Testing/QA team, there are simple ways to get started with coding, even if you have an existing product or service with ongoing development and even with pressing deadlines.

Below there are 2 simple ways that help any developer bootstrap testing in their team. It doesn't necessarily need to get the CTO or Lead Engineer involved – any team member can get things rolling.

Methodology 1: Start small

The "start small" approach is a simple workflow that anyone can use, including junior developers who haven't tested anything before. For existing projects without a test suite, it can seem as an enormous endeavor to start writing tests. The solution to this is: Just do it. Small. – Simply get started and commit in small increments.

Here is an overview of simple steps to follow:

1. Setup test suite
2. Write 1 test
3. Another team member writes 1 more test
4. The team commits to writing 2 tests each for the next week

Methodology 2: Bug-based testing

This approach helps teams avoid having the same bugs reoccur – besides there so many other bugs out there, no reason to stick to the same ones!

This approach may require a slightly better understanding of debugging and root cause analysis, in order to come up with specific and accurate reproduction steps that are necessary to setup the test scenarios – remember the previous section about having explicit test cases.

Steps to get you started:

1. Setup the test suite
2. Write 1 test
3. When the team is handling a bug, write a test for this bug
4. Repeat for every bug you encounter

A common step is, unsurprisingly, to write one test. This is what gets you started, and keeps you moving towards testing more. Writing the one test can provide momentum for individuals and teams and it's a lot easier to aim to write one test rather than more generic goals such as increasing coverage by X percent. Let's see in the next chapter how to tackle this first, yet important, step.

"Set small goals along the way and
don't be overwhelmed by the process."

– Kara Goucher

How to write one test

Diving into the most important step, writing a test. Each time. How to write the one test:

1. Write 1 test case. Anywhere. Anyhow.
2. Put it in the right place, ie if this is a unit test of a particular model, it should be in the test file of this model
3. Make sure the test is actually testing
4. Write one more test case, and repeat
5. Find similarities in the existing test case, which would warrant using shared examples to DRY things up

Make sure the test is actually testing

Sometimes we get carried away with testing and particularly with stubs and mocks, that we end up testing that the truth returns true. That's not at all what we are aiming for with our test suite though.

If you have a test that checks if the returned value is true, change your test to expect a false value, and make sure it fails. If it fails, you can revert it back to true, and proceed. However, if it does not fail, that means that it is returning true – as you were initially expecting – but for the wrong reason! You need to dig into that reason and figure it out.

Here are some ideas to get you started with your investigation to properly set up your test:

- ○ Your test might be testing the wrong thing
- ○ Or, quite probably, you haven't set up the test case properly – to run a test we assume we have several pieces of data already available and to recreate a particular test scenario our data needs specific values
- ○ Somewhere in there there is an assumption

You should absolutely not just leave a test passing, if it wouldn't fail with the opposite expected value, because it's not really testing what you wanted to be testing – that's a testing trap.

Testing 1% can make a difference

Why does it make sense to start small with testing and try the above mentioned methodologies?

1% of code is probably useless for any application, but 1% of tests could potentially catch the 1 case where there is a bug in the code. When we are writing code, 1% of a feature means nothing, and there is nothing we can use from 1% or even 10% of the code which is required to develop a feature or an application. We literally need all the code for the particular task we are working on. However, that's not quite the case with tests. The thing with testing is that, if you have 100 different cases to test, even if you only test 1 case, you still have added some value to your code, and that's because 1% of tests is way better than none. 1% of tests could include 1 bug that would get caught *before it makes it to production* -- and that's a win any day! It might not be much, but it is a whole lot more than nothing. Testing 1% can make a difference, might be a small one, but it is a difference nonetheless.

The Shift Left approach for Testing

"This shift left in the agile development process means testing starts much earlier in the application lifecycle. As a result, developers should own QA, which means developers write, execute, and maintain tests for the code they produce; it encourages everyone on the team to be engaged in delivering high-quality products to the customer quickly." – Alexsandro Souza[22]

What does shift left mean?

To be put simply, as described by BrowserStack[23]:

- Tests being run by developers themselves before they push their individual code unit to version control.
- The Shift Left Testing approach implements a process that lets developers detect bugs early and often because these code units are small and infinitely more manageable

[22] https://dev.to/apssouza22/we-are-testing-software-wrongly-and-it-costs-a-lot-of-money-23o5
[23] https://www.browserstack.com/guide/what-is-shift-left-testing

The problem

According to BrowserStack, when testing is paused until the end of development, any bugs that do show up will usually be *more difficult to fix*. Add to that the extra cost of bug fixing at later stages of the software development cycle, as described in The cost of bug fixing in multiple stages of the software lifecycle and it should be a no-brainer that we should focus on creating and running tests as early as possible in our software development process.

Metrics

What are metrics? Why do we care? Isn't coverage all that matters?

Tricentis[24] blog post presents a collection of metrics for software testing. The list includes the obvious, such as coverage and cost per bug fix, but it also comes with a more meaningful approach such as being mindful about "How bad are the bugs?" and "How many bugs did the test team not find?" which can provide great insights to how the team works and how the testing process evolves. The ultimate goal is to determine what we are doing right and what we can **improve**.

Here's a list of metrics all developers should ask themselves:

1. How long does it take our team to write tests for a feature?
2. Are the tests of good quality?
3. How many times did a test catch a bug in new code?
4. How bad are the bugs?
5. How many bugs were not spotted before a feature was shipped to production?

Coverage is one of the most mentioned metrics when it comes to automated testing, yet there are so many more aspects to be mindful about.

[24] https://www.tricentis.com/blog/64-essential-testing-metrics-for-measuring-quality-assurance-success/

Coverage VS Good Tests

Is it enough if we have 90% coverage? Absolutely no! Hands down that's the worst approach to determining the effectiveness of the testing process. As Martin Fowler[25] points out, test coverage is a good way to identify untested code. However it doesn't tell us much about the quality of the code, or if we have covered all the dozens of different subcases.

According to https://dzone.com/articles/we-are-testing-software-incorrectly-and-its-costly reporting test coverage and using it as a threshold for shipping software

- contributes to lower quality tests just to increase the numbers, and
- tends to distract from the use cases that should drive the software development process

How to interpret coverage

That said, having 100% test coverage is awesome! The key here, though, is interpretation. 100% coverage does not mean that you are testing everything you possibly could, in fact it doesn't even mean you are testing everything you *should*. But it does show that you have bootstrap your testing suite for all relevant parts of the software, which is awesome and super important specially if you are now starting with testing. That means that developers can **focus** on the specific test cases they want to add rather than configuring the test suite and adding completely new components.

Once we reach a comfortable level of tests within our software, it is not long until we wonder whether or not our tests should be checking for more – more assumptions, more specs, more indirect acceptance criteria. Staying on top of what may be considered "obvious" by the team can help identify even more test cases, and even aspects of the software which should be working a certain way but have never been tested. Staying aware of different assumptions between different teams is also valuable when identifying more items for testing. For instance, the security team might find it crucial to add tests that check for possible SQL injections – and that's just one idea, talk to your team to find out more! At the same time, the business or legal teams may assume that the engineering team is always checking that the software complies with internal policies or external requirements – even though that's not always easy to split up into test cases which run automatically. Again, close collaboration between teams with clear and transparent communication can do wonders in revealing assumptions, producing new testing requirements and improving the overall software and service being offered.

[25] https://martinfowler.com/bliki/TestCoverage.html

Why continuous compliance?

“We propose continuous compliance to guarantee that the codebase stays compliant on each code change using lightweight verification tools. Continuous compliance increases assurance and reduces costs.“ -- https://homes.cs.washington.edu/~mernst/pubs/continuous-compliance-ase2020-abstract.html

According to the paper, continuous compliance can be summed up as the process in which teams “build verification tools for compliance controls, and each commit runs the verifiers in continuous integrations”. This approach enables both increased verification towards compliance of the software as well as fast reporting directly to developers – which can enable faster fixes, less communication and organizational overhead between teams that need to review code, report back the findings to the engineering team which needs to receive the results, understand them and schedule them into their sprints for development. This alone can positively impact costs; additionally it reduces the amount of effort required and eliminates a huge part of human error on the auditors’ side.

Techniques such as bug-based testing can be used in continuous compliance as well, to ensure that any non-compliant cases become part of the verification tools in place, so that the same defect does not occur again.

There are industries which are inherently obliged to comply with regulations, such as the medical sector, financial software, aerospace, services related to critical infrastructure, as well as software related to security, cryptography and trust services. Yet there are policies that may apply even if the software does not fall into these categories, GDPR is one very evident example.

Is being compliant enough?

Making sure the software is compliant, and particularly doing so continuously and in an automated way, is essentially part of the bigger sector of automated testing, and should be seen as part of automated testing responsibilities. However, being compliant, meaning adhering to policies and regulations does not guarantee there are no bugs in the software. Testing for bugs, or generally writing unit tests, is the fundamental step in creating an automated testing suite and environment. There are, of course, many more aspects, such as integration tests and end to end checks, which are also very useful and provide a more holistic approach to the software, based on the overall product. Furthermore, such tests are a lot more in the vicinity of the experience of actual users. On the other hand, making sure we reduce bug occurrence by testing an application, and even testing it well, does not do much about keeping the application compliance, because whether or not there are bugs is not directly relevant to being compliant. Certainly the overall well-being of a software contributes to it being compliant, in the sense that the absence of bugs might cover parts of the regulations, however the software automated testing process is not primarily concerned with compliance matters, making it somehow difficult to make claims about the software's adherence to policies just by evaluating the test suite. Both compliance testing and traditional software automated testing are required in order to determine production-readiness, evaluate compliance and make relevant business decisions. In fact, it calls for a more dedicated team for compliance testing & QA to run, as doing it effectively assumes knowledge of the product from its business perspective as well as the user experience we are aiming for and also understanding of regulations that must be followed. That's why it's often useful to combine compliance teams and software development teams or QA testers, and create a combined workgroup to tackle the continuous compliance through testing explicitly for compliance matters.

Compliance VS Testing

Compliance, particularly when paired with software development, does bring challenges when technical people try to understand and accurately interpret legal documents, extract necessary information into actionable tasks and stay up to date with changes.

The world of compliance consists of policies and legal terms, while the testing refers to more technical aspects such as issues, sprint goals and product requirements (often showcased through user stories).

Compliance	**Testing**
● Policies	● Technical issue
● Industry standards	● Business requirements
● Complex documents	● User stories and flows
● Legal requirements	● Sprint goals

6. Bias in software, and the role of testing

Assumptions made during any stage of the Software Development Life Cycle, and particularly implementation and testing phases, can be detrimental to user experience, developer motivation, business costs, as well as the overall state of the software.

Aiming at a robust code is a high priority, even though «building robust systems that encompass every point of possible failure is difficult because of the vast quantity of possible inputs and input combinations»[26], it is always useful to keep in mind that users will most likely do the 'wrong' thing when using an application, which will cause an error. Handling those errors gracefully[27] and guiding the user to the proper use of software not only will provide end-users with the desired outcome but will also reduce the amount of complaints and cases that need to be investigated by support and tech teams.

When it comes to testing, it is imperative to predict and test for as many edge cases and wrong inputs as possible, as this will reveal weak code and protect us from having to deal with the errors in production.

Assumptions not only negatively affect the robustness of the codebase, but also slow down the overall progress of the software, as the team's focus shifts to bug fixing and users' issues – which, among other things, can lead to developer frustration. Losing focus from creating awesome software that scales, can cause poor analysis and missing cases during requirements break-down, which in turn leads to more issues, causing a vicious cycle that, additionally to exceptions and errors, can also result in introducing vulnerabilities in the code.

"Software is created by people with specific ideas in mind"

– Valentin Jeutner

The human brain is awesome at generalizing, but any ideas our mind has is also a source of assumption and bias. Software is shaped by human choices, all of which can be questioned and debated. While AI is not necessarily built with bias, as it can easily be applied on larger scale, it amplifies the biases of its creators[28]

Acknowledging the issue at hand, and being proactive are already the first steps to improving software and reducing bias. "American research shows that the most advanced AI facial recognition tools make mistakes

[26] https://en.wikipedia.org/wiki/Robustness_(computer_science)
[27] Read more at https://uxmag.com/articles/failing-gracefully
[28] https://www.coursera.org/learn/ai-law/lecture/plqQD/law-and-ai-hardware

much more often in cases with black women than in cases with white men" – Ulrika Andersson[29]. Keeping this in mind while developing and testing software, can significantly impact the end result – like any other conscious or unconscious decision and choice.

Developers of AI systems are not responsible for what the AI system does, but they do share responsibility for bias inserted during development and/or training, meaning that developers, and anyone else involved in the software development life cycle, must be mindful and educated, so as not to introduce their own biases and assumptions into the system.

Automated testing systems are a great tool to provide specific checks against a wide range of different scenarios after every change in the code. Still they only test what they are told, or what the developer/tester has written as test cases, or they behave according to the training set that was used. Carefully crafting the use cases introduced in automated testing and training systems, can increase the possibility of a more accurate outcome, if not prevent the occurrence of a bug altogether. Thinking outside the box is certainly desired when it comes to coming up with scenarios and creating complex cases to test how the software will handle them.

Julia Reinhard raises a great point about not releasing software to people before it is tested[30] because of what it really means to release a software, i.e. let it loose where people will use it and be affected by its use. The provided example about mood detection systems is a great opportunity to take a step back and think long and hard about how **software affects people**, and how easy it is to become non-compliant even if having the best intentions in mind. According to the podcast, mood detection systems can put people into categories and reinforce cliches, which doesn't necessarily comply with non-discrimination laws. From a more business perspective, this is really a wake-up call of how being laser-focused on delivering extra features can cause legal problems, and how **a business can fundamentally benefit from having a well-defined development process that includes enough time and resources for proper QA**.

Left-handed people and the role of testing as a preventive measure

Left-handed people are more inclined to click on options we make available on the left side of the screen, but developers don't necessarily use those options much, and consequently they are less likely to test for those functionalities during manual testing – unless they have a well-organized list of cases to test for, in which case they might as well incorporate them into an automated testing system. Including alternative user

[29] https://www.coursera.org/learn/ai-law/lecture/TidPQ/ai-and-criminal-law
[30] https://podcasts.apple.com/us/podcast/julia-reinhardt-what-do-gdpr-and-ai-regulations-mean/id1506212617?i=1000531062544

flows into end-to-end testing scenarios is relatively easy to do, once you think about it. And once written, a test continues to be available and executed to test for those different journeys our users take. This is easier to implement as the functionalities are being introduced and coded, rather than later on. TDD methodology can come handy for an additional reason, as it can act as a preliminary and preventive measure for untested options.

Test Driven Development and the shift-left approach call for testing software early. The initial stages of SDLC provide a unique opportunity to reflect on how a new feature is incorporated into the existing software and the different ways it can be used. Before coding and before we make up our minds about what the user workflow should be, and before we get caught up in the business logic of the application itself, we have a window of opportunity to look at this with fresh eyes and uncover more ways the new option can be used from a user perspective.

Writing down test cases beforehand can help prevent missing out on edge cases and less usual ways of user interaction with the software, precisely because we don’t start testing after we have seen the end result, so the final implementation of a new component is still a lot less tangible, which leaves a lot of room for interpretation of the initial idea and designs, and allows us to get more creative with the test scenarios – while at the same time our intuitive thinking might as well introduce our own biases!

AI that is safe, and unbiased – Reality or wishful thinking?

Are we delusional to think that AI systems from Siri to autonomous automobiles and even human-looking robots[31] can provide a safe and inclusive environment for humans and technology to coexist in peace?

Recent events[32] might leave us hanging in hesitation or even fear, but legislators are continuing to improve the legal frame in which AI can be developed and most importantly provided for use.

European Union publication for “Liability for artificial intelligence and other emerging digital technologies” suggests that any such technology “must come with **sufficient safeguards**, to minimize the risk of harm these technologies may cause, such as bodily injury or other harm.”[33]

[31] Like very vividly presented in the movie Ex Machina a few years back https://www.imdb.com/title/tt0470752/

[32] https://www.autoevolution.com/news/tesla-model-s-hits-the-back-of-a-bus-in-newport-beach-autopilot-may-be-involved-176486.html Tesla crashes into (possibly stationary) bus

[33] https://op.europa.eu/en/publication-detail/-/publication/1c5e30be-1197-11ea-8c1f-01aa75ed71a1/language-en

Another article raises concerns for autonomous cars, particularly about the need of sufficient standards and regulations to be in place.[34]

Non profit organization also supports, in the ANSI approved UL Standard, that the **reliability** of hardware and software is **necessary**[35] and that traditional safety practices may require changing to accommodate autonomy, eg. fault mitigation actions to be in place when there is lack of a human operator.

Still technology can provide the means for a new era of possibilities in various industries, from the medical field[36][37] to Network systems[38], to ensuring cyber attacks don't leave us hanging[39] [40], to presenting software creators with tools to work faster and smarter[41] and operations personnel (devops) with solutions[42], options[43] and tools[44]. Compliance QA can help make it happen by providing the framework of verifying outcomes, testing and checking every step of the way and validating results, before, during, and after they are used in real life. At the same time it is technology evolution that contributes to testing improvement[45].

[34] https://theconversation.com/autonomous-cars-five-reasons-they-still-arent-on-our-roads-143316
[35] ANSI/UL 4600, Standard for Safety for Evaluation of Autonomous Products
https://www.shopulstandards.com/ProductDetail.aspx?productid=UL4600
[36] https://www.researchgate.net/publication/361190449_Exploitation_of_Emerging_Technologies_and_Advanced_Networks_for_a_Smart_Healthcare_System
[37] https://www.researchgate.net/publication/336349466_An_Enhanced_and_Secure_Cloud_Infrastructure_for_e-Health_Data_Transmission
[38] https://ieeexplore.ieee.org/abstract/document/9844054
[39] https://www.researchgate.net/publication/356653094_Automated_Testing_of_Data_Survivability_and_Restorability
[40] https://www.sciencedirect.com/science/article/abs/pii/S2210537918300490
[41] https://dl.acm.org/doi/abs/10.1145/3520304.3528952
[42] https://www.researchgate.net/publication/337954540_Testing_microservice_architectures_for_operational_reliability
[43] https://www.researchgate.net/publication/345389310_Automation_Testing_in_DevOps
[44] https://www.researchgate.net/publication/359924657_ExVivoMicroTest_ExVivo_Testing_of_Microservices
[45] https://ieeexplore.ieee.org/document/9805972

7. Contribution and future work

In the current paper we explored the evolution of automated testing to testing for compliance in a continuous manner, its challenges and the related costs, as well as practical methodologies for teams to get started and improve. The paper presents a variety of articles around related topics, such as automated testing and QA, defects and technical debt, compliant industries and their needs, and challenges regarding converting policy documents to technical issues.

A sector that equally affects the tech industry as well as the legal world, is AI. With its rapid growth and innovative path that leads businesses and automation, legislation is trying to catch up as the tech world is continuously evolving. Whether or not it will be able to sustain policies and regulations up to a good standard that allows for even more research and new ideas, remains to be seen. Its particular needs for not just testing technical features but also compliance with regulations, is rather unique, as it affects its business more than any other industry. Furthermore there are interesting aspects of testing when it comes to decentralized systems such as blockchain[46]. Other existing techniques such as intrusion detection systems (IDS) can boost research and development of methods for testing and monitoring for security and compliance related incidents – such systems[47] could be adapted and used to get real-time alerts for possible issues. More research towards the impact of automated testing and continuous compliance from a both technical and business/financial perspective would be of interest and definitely work exploring. Perhaps the necessary safeguards required for AI can come in the form of a process towards continuous compliance.

[46] https://dl.acm.org/doi/10.1109/ICPC.2019.00048
[47] http://ikee.lib.auth.gr/record/329274

8. Final thoughts

Understanding the code as a non-programmer is challenging, often unnecessary and almost always beyond the expertise and responsibilities of a lot of people who are called to determine if the code works as expected, including whether or not it is secure and compliant.

Making assumptions is the primary reason why bugs occur, therefore we don't want to resort to hypotheses, especially when we can have tangible data. For that reason, and to avoid additional problematic situations, after the code is written, we must focus on making our code as bulletproof as possible.

Hoping for bug-free code is utopian, and the only question is whether or not we know we have a bug. The path to success is to test and determine with certainty that we have the desired outcome within the restrictions of our sector.

TLDR there is no such thing as completely bulletproof code, but it is the unattainable goal we try to reach, in an effort to surpass our own selves and elevate the quality of our code.

"What gets measured, gets improved"

– Peter Drucker

References

1. Honest, Nirali. (2019). Role of Testing in Software Development Life Cycle. International Journal of Computer Sciences and Engineering. 7. 886-889. 10.26438/ijcse/v7i5.886889.
2. Sten Pittet. The different types of software testing. https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing
3. Measuring The Software Security Requirements Engineering Process. Cybersecurity and infrastructure security agency (CISA)
4. Kumari, Bhawna & Chauhan, Naresh & Syed, Habeebullah Hussaini. (2018). A COMPARISON BETWEEN MANUAL TESTING AND AUTOMATED TESTING. SSRN Electronic Journal. 5. 323-331.
5. Bacchelli, Alberto & Ciancarini, Paolo. (2009). How to compare and exploit different techniques for unit-test generation.
6. Brian Chess (Fortify Software), Julia H. Allen (2009. An Alternative to Risk Management for Information and Software Security. Carnegie Mellon University
7. Mahajan, Prasad & Harshal, India & Bharati, Shedge & Uday, India & Bharati, Patkar & Coe, & Patkar, Uday. (2022). Automation Testing In Software Organization. https://www.researchgate.net/publication/362517283_Automation_Testing_In_Software_Organization
8. Barraood, Samera & Haslina, Haslina & Baharom, Fauziah. (2021). A Comparison Study of Software Testing Activities in Agile Methods.
9. Choi, Yunja & Byun, Taejoon. (2015). Constraint-based test generation for automotive operating systems. Software & Systems Modeling. 16. 10.1007/s10270-014-0449-6.
10. D, Kavitha & .S, Ravikumar. (2021). Software Security Requirement Engineering for Risk and Compliance Management. International Journal of Innovative Technology and Exploring Engineering. 10. 11-17. 10.35940/ijitee.E8628.0210421.
11. Breaux, Travis & Antón, Annie. (2022). Acquiring Software Compliance Artifacts from Policies and Regulations.
12. Mubarkoot, Mohammed & Altmann, Jörn. (2021). Towards Software Compliance Specification and Enforcement Using TOSCA. 10.1007/978-3-030-92916-9_14.
13. Gál, Péter. (2021). Bug Prediction Capability of Primitive Enthusiasm Metrics. 10.1007/978-3-030-87007-2_18.

14. Haldar, Swagatam & Vijaykeerthy, Deepak & Saha, Diptikalyan. (2021). Automated Testing of AI Models.
15. D. Yuniasri, T. Badriyah and U. Sa'adah, "A Comparative Analysis of Quality Page Object and Screenplay Design Pattern on Web-based Automation Testing," 2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE), 2020, pp. 1-5, doi: 10.1109/ICECCE49384.2020.9179470.
16. L. F. de Lima, M. C. Horstmann, D. N. Neto, A. R. A. Grégio, F. Silva and L. M. Peres, "On the Challenges of Automated Testing of Web Vulnerabilities," 2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2020, pp. 203-206, doi: 10.1109/WETICE49692.2020.00047.
17. D. Falessi and R. Kazman, "Worst Smells and Their Worst Reasons," 2021 IEEE/ACM International Conference on Technical Debt (TechDebt), 2021, pp. 45-54, doi: 10.1109/TechDebt52882.2021.00014.
18. Dawson, Maurice & Burrell, Darrell & Rahim, Emad & Brewster, Stephen. (2010). Integrating Software Assurance into the Software Development Life Cycle (SDLC). Journal of Information Systems Technology and Planning. 3. 49-53.
19. A, J. (2022, January 3). What is Technical Debt and Why QA Testers Should be Concerned About It? Software Testing Help. https://www.softwaretestinghelp.com/technical-debt-and-qa/
20. Khan, Mohd. (2011). Different Approaches To Black box Testing Technique For Finding Errors. International Journal of Software Engineering & Applications. 2. 10.5121/ijsea.2011.2404.
21. A. (2020, May 7). Automated vs Manual Testing: Which Should You Use, and When? Apica. https://www.apica.io/blog/difference-between-automated-manual-testing/
22. Bridgwater, A. (2022, January 13). Why Low-Code Isn't No-Test. Forbes. https://www.forbes.com/sites/adrianbridgwater/2022/01/12/why-low-code-isnt-no-test/?sh=641d647a2238
23. Menshawy, Rana & Hassan Yousef, Ahmed & Salem, Ashraf. (2021). Code Smells and Detection Techniques: A Survey. 78-83. 10.1109/MIUCC52538.2021.9447669, 2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)
24. Maxwell, Jeremy & Antón, Annie & Swire, Peter. (2011). A legal cross-references taxonomy for identifying conflicting software requirements. Proceedings of the 2011 IEEE 19th International Requirements Engineering Conference, RE 2011. 197 - 206. 10.1109/RE.2011.6051647.

25. Winchester, Hilary. (2018). How quality assurance codes change: beyond 'bells and whistles' and 'code by catastrophe'?. Quality in Higher Education. 10.1080/13538322.2018.1460900.
26. Sharma, Tushar & Kechagia, Maria & Georgiou, Stefanos & Tiwari, Rohit & Sarro, Federica. (2021). A Survey on Machine Learning Techniques for Source Code Analysis.
27. Liability for artificial intelligence and other emerging digital technologies https://op.europa.eu/en/publication-detail/-/publication/1c5e30be-1197-11ea-8c1f-01aa75ed71a1/language-en
28. Autonomous cars: five reasons they still aren't on our roads https://theconversation.com/autonomous-cars-five-reasons-they-still-arent-on-our-roads-143316
29. ANSI/UL 4600, Standard for Safety for Evaluation of Autonomous Products https://www.shopulstandards.com/ProductDetail.aspx?productid=UL4600
30. Julia Reinhardt, What do GDPR and AI regulations mean for Silicon Valley?, Voices of the Data Economy (Apple Podcasts #14) https://podcasts.apple.com/us/podcast/julia-reinhardt-what-do-gdpr-and-ai-regulations-mean/id1506212617?i=1000531062544
31. AI and Criminal Law, LUND University, Coursera https://www.coursera.org/learn/ai-law/lecture/TidPQ/ai-and-criminal-law
32. Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. (2020), Continuous Compliance, ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering, (Melbourne, Australia), Sep. 2020, pp. 511-523. https://homes.cs.washington.edu/~mernst/pubs/continuous-compliance-ase2020-abstract.html
33. Ted O'Meara, Failing gracefully, https://uxmag.com/articles/failing-gracefully
34. Tricentis (2016), 64 Essential testing metrics for measuring quality assurance success, https://www.tricentis.com/blog/64-essential-testing-metrics-for-measuring-quality-assurance-success
35. Martin Fowler, Test Coverage, https://martinfowler.com/bliki/TestCoverage.html
36. Memos, Vasileios & Psannis, Kostas & Goudos, Sotirios & Kyriazakos, Sofoklis. (2021). An Enhanced and Secure Cloud Infrastructure for e-Health Data Transmission. Wireless Personal Communications. 117. 10.1007/s11277-019-06874-1.
37. Alexsandro Souza (2021), We are testing software wrongly. And it is costly. https://dev.to/apssouza22/we-are-testing-software-wrongly-and-it-costs-a-lot-of-money-23o5
38. Shreya Bose (2020), Shift Left Testing: What It Means and Why It Matters BrowserStack https://www.browserstack.com/guide/what-is-shift-left-testing

39. Christos Stergiou, Kostas E. Psannis, Brij B. Gupta, Yutaka Ishibashi, Security, privacy & efficiency of sustainable Cloud Computing for Big Data & IoT, Sustainable Computing: Informatics and Systems, Volume 19, 2018, Pages 174-184, ISSN 2210-5379, https://doi.org/10.1016/j.suscom.2018.06.003.
40. Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. Evolutionary-based automated testing for GraphQL APIs. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '22). Association for Computing Machinery, New York, NY, USA, 778–781. https://doi.org/10.1145/3520304.3528952
41. Pietrantuono, Roberto & Russo, Stefano & Guerriero, Antonio. (2019). Testing microservice architectures for operational reliability. Software Testing, Verification and Reliability. 30. 10.1002/stvr.1725.
42. Jain, Prateek & Consulting, Jupitor It And Research. (2020). Automation Testing in DevOps.
43. Gazzola, Luca & Goldstein, Maayan & Mariani, Leonardo & Mobilio, Marco & Segall, Itai & Tundo, Alessandro & Ussi, Luca. (2022). ExVivoMicroTest: ExVivo Testing of Microservices. Journal of Software: Evolution and Process. 10.1002/smr.2452.
44. Muller, Sylvain & Bryce, Ciar´an. (2021). Automated Testing of Data Survivability and Restorability. 111-126. 10.5121/csit.2021.111810.
45. Jianbo Gao, Han Liu, Yue Li, Chao Liu, Zhiqiang Yang, Qingshan Li, Zhi Guan, and Zhong Chen. 2019. Towards automated testing of blockchain-based decentralized applications. In Proceedings of the 27th International Conference on Program Comprehension (ICPC '19). IEEE Press, 294–299. https://doi.org/10.1109/ICPC.2019.00048
46. Sanket (2019), The exponential cost of fixing bugs, https://deepsource.io/blog/exponential-cost-of-fixing-bugs/
47. Minopoulos, G.M.; Memos, V.A.; Stergiou, C.L.; Stergiou, K.D.; Plageras, A.P.; Koidou, M.P.; Psannis, K.E. Exploitation of Emerging Technologies and Advanced Networks for a Smart Healthcare System. Appl. Sci. 2022, 12, 5859. https://doi.org/10.3390/ app12125859
48. P. Alcock, B. Simms, W. Fantom, C. Rotsos and N. Race, "Improving Intent Correctness with Automated Testing," 2022 IEEE 8th International Conference on Network Softwarization (NetSoft), 2022, pp. 61-66, doi: 10.1109/NetSoft54395.2022.9844054.
49. C. S. Spahiu, L. Stanescu, R. Marinescu and M. Brezovan, "Machine Learning System For Automated Testing," 2022 23rd International Carpathian Control Conference (ICCC), 2022, pp. 142-146, doi: 10.1109/ICCC54292.2022.9805972.

Appendix I – Articles list

	Title / Link		Short overview
1	A comparison between manual testing and automated testing	December 2018 SSRN Electronic Journal 5(12):323-331[48]	A fundamental and easy to understand comparison between manual and automated testing, including listing the benefits and presenting the characteristics of a good test case.
2	How to compare and exploit different techniques for unit-test integration	January 2009[49]	Focused on unit-testing specifically, it presents a comparison between different tools for unit testing (JUnit, JCrasher, Randoop) which generate unit tests automatically.
3	Automated System-level safety testing using constraint patterns for automotive operating systems	January 2015[50] DOI: 10.1007/s10270-014-0449-6	Automated test generation for safety testing of automotive operating systems. The constraint-based method of testing presented an advantage for testing illegal behaviors.
4	Software Security Requirement Engineering for Risk and Compliance Management	March 2021[51] DOI: 10.35940/ijitee.E8628.0210421 International Journal of Innovative Technology and Exploring Engineering 10(5):11-17	Mapping of security requirements with compliance mandates in an effort to document and visualize their connection aiming to reduce the risk of non-compliances. The article also provides useful explanations and definitions for security terms and concepts.
5	Acquiring Software Compliance Artifacts from Policies and Regulations	Breaux, Travis & Antón, Annie. (2022). Acquiring Software Compliance Artifacts from Policies and Regulations. [52]	Provides a methodology to extract technical requirements from legal and policy documents. Explains the problems which developers face when implementing compliant software and outlines the necessity for an organized method to identify and classify such compliance requirements.
6	Towards Software Compliance Specification and	December 2021[53] DOI: 10.1007/978-3-030-92916-9_14	How to handle changes in policy documents and industry requirements, the importance of compliance management throughout the software development lifecycle (SDLC).

[48] https://www.researchgate.net/publication/349636718_A_COMPARISON_BETWEEN_MANUAL_TESTING_AND_AUTOMATED_TESTING
[49] https://www.researchgate.net/publication/228777686_How_to_compare_and_exploit_different_techniques_for_unit-test_generation
[50] https://www.researchgate.net/publication/282480447_Automated_System-level_Safety_Testing_of_Automotive_Operating_Systems
[51] https://www.researchgate.net/publication/350567020_Software_Security_Requirement_Engineering_for_Risk_and_Compliance_Management
[52] https://www.researchgate.net/publication/228818973_Acquiring_Software_Compliance_Artifacts_from_Policies_and_Regulations
[53] https://www.researchgate.net/publication/356890651_Towards_Software_Compliance_Specification_and_Enforcement_Using_TOSCA

	Enforcement Using TOSCA		
7	Bug Prediction Capability of Primitive Enthusiasm Metrics	September 2021[54] DOI: 10.1007/978-3-030-87007-2_18 In book: Computational Science and Its Applications – ICCSA 2021	Explores the use of Primitive Enthusiasm metrics (method based metrics) to improve bug prediction. The metrics measured
8	Code Smells and Detection Techniques: A Survey	2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)	Different detection approaches and the language restrictions due to available tools.
9	Automated Testing of AI Models	October 2021[55]	A testing tool for image and speech-to-text models. This framework provides black-box testing, which, once configured,can be applied to a large number of similar models.
10	A comparative analysis of quality page object and screenplay design pattern on web-based automation testing	June 2020[56] DOI: 10.1109/ICECCE49384.2020.9179470	Comparison analysis between screenplay design and page object design patterns for automated testing.
11	On the Challenges of Automated Testing of Web Vulnerabilities	2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), September 2020[57]	Evaluation of 5 penetration testing tools against 7 web applications for detention and exploitation of vulnerabilities, using the PTES (Penetration Testing Execution Standard) methodology
12	Worst Smells and Their Worst Reasons	Falessi, Davide & Kazman, Rick. (2021). Worst Smells and Their Worst Reasons [58]	Survey with developers to identify details surrounding the creation of technical debt. Provides a statistical analysis and its results for 5 hypotheses.
13	What Is Technical Debt And Why QA Testers Should Be Concerned About It?	https://www.softwaretestinghelp.com/technical-debt-and-qa/, January 2022	A comprehensive post about the causes of technical debt and the challenges that surround it.

[54] https://www.researchgate.net/publication/354520302_Bug_Prediction_Capability_of_Primitive_Enthusiasm_Metrics
[55] https://www.researchgate.net/publication/355141671_Automated_Testing_of_AI_Models
[56] https://ieeexplore.ieee.org/document/9179470
[57] https://ieeexplore.ieee.org/document/9338518
[58] https://www.researchgate.net/publication/350131861_Worst_Smells_and_Their_Worst_Reasons

14	Automated vs Manual Testing: Which Should You Use, and When?	https://www.apica.io/blog/difference-between-automated-manual-testing/, 2022	Post on business portal which provides a concise overview of different kinds of testing along with their definitions and when each type of testing is advised. Offers clear definitions on automated and manual testing from a technical and hands on perspective, and it focuses on their differences and factors to consider when choosing one or the other.
15	Why Low-Code Isn't No-Test	https://www.forbes.com/sites/adrianbridgwater/2022/01/12/why-low-code-isnt-no-test/?sh=641d647a2238, January 2022	Technical debt, automation oversight when it comes to testing, from a business standpoint. The article is based on corporate insights and data to support the necessity of automated testing and the consequences of lack thereof.
16	A legal cross-references taxonomy for identifying conflicting software requirements	Proceedings of the 2011 IEEE 19th International Requirements Engineering Conference, RE 2011. 197 - 206. 10.1109/RE.2011.6051647, October 2011[59]	The impact of cross-references in legal documents and strategies for engineers to resolve conflicting requirements.

[59] https://www.researchgate.net/publication/224263869_A_legal_cross-references_taxonomy_for_identifying_conflicting_software_requirements

Appendix II - Terminology/Glossary

Bug / code smell	When the software is behaving in a different way than intended
Compliance	Verification that software follows industry policies and requirements
CI (Continuous Integration)	To regularly merge all code contributions into a central repository (where build and test suite run automatically)
CI/CD	The fundamental 2-part automation for software development, 1. CI -- Continuous Integration, one of the main purposes it serves is to automatically run tests at every push (that includes merging a PR/MR) 2. CD -- Continuous Delivery or Continuous Deployment, are 2 methods to make the software available, ie deploy to a production server (or other ENV)
Pull Request (PR) / Merge Request (MR)	Propose changes to be merged into the repository in a way that is easy for the team to review the suggested changes Both terms refer to the same thing: ● Github uses the term PR ● Gitlab uses the term MR
SDLC - Software Development Life Cycle	A detailed plan to describe the development process, including a stage for testing
Shift Left Testing approach	Start testing as early as possible, instead of waiting for features to be completed by the dev team and passed on to the QA team
TDD (Test Driven Development)	The methodology of writing tests before writing code
Technical debt	When trying to implement more code than the team's capacity, which results in best practices not being applied
Test suite	A collection of test cases
Version control	Track and manage changes to code. Useful to keep code safe, usually using services such as Gitlab (and others), to promote teamwork and collaboration on software development to facilitate immediate and easy code reviews on suggested changes