Kvalitet i testiranje softvera

- **Način polaganja predmeta:** sistem 70/30, 2 kolokvijuma
- **Predavači iz privrede**
- **Tržište rada**
- **Postoji dosta zajednica za QA u Srbiji.**
- **Studentski vs industrijski softver**
- Američki nacionalni institut za standarde (NIST) procenjuje da softverski bagovi izazivaju 60 milijardi dolara gubitaka u američkoj ekonomiji godišnje.

Plan rada na vežbama

1. Maven
    a. Kreiranje projekta, zavisnosti, paketi, verzije jave itd
2. Junit
    a. Testiranje na nivou klasa
    b. Arange, Act, Assert
    c. DRY
3. Mockito
    a. Object under testing
4. Hamcrest – Full packet
5. TestNg
    a. Xml konfiuguracija
    b. Logovanje – log4j
6. Selenium + TestNg
    a. Linkovi
    b. Forme
    c. Logovanje, cookie
    d. Sadržaj teksta
7. API test
    a. Postman
8. JMeter
    a. Performanse
    b. Plan testiranja
    c. Izveštaji
9. Jenkins + Git – automatsko testiranje na svaki merge grana

a) Software Testing Introduction

1.1 What is software testing and why it's necessary?

Testing is a process rather than a single activity - there are series of activities involved. Test process ensures that each and every component of the software meets the business and technical requirements that guided its design and development and works as expected.

Your software is never going to be 100% right (bug free), and it's ok.

- Testing is necessary because we all make mistakes, all the time.
- Anything that we produce can go wrong.
- If we make mistakes, then we need to check our work.
- Others would see our mistakes better than we do.
- Our logic might miss something, that at first doesn't seem much important, but it can affect the system in means that we have not intended to, and it can be something that isn't strictly specified (in documentation or requirements), but it is expected to be included and handled.
- Some of those mistakes are less important, but some of them are expensive (trucks) or dangerous (autonomous cars, Veoneer, Uber,…).
- Testing was perceived as confirmation that developed system meets its requirements but it's useful as it can save time, money and reduce risks.

Software testing can be divided into two steps:

- **Verification:** it refers to the set of tasks that ensure that software correctly implements a specific function. -- "*Are we building the product right?*"
- **Validation:** it refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. -- "*Are we building the right product?*"

What does this mean? During product development, we might not know every aspect. Things get overseen, from small ones to big ones. Testing gives a fresh perspective on the functionality, look and feel of the product. It is the outside view, and perspective is shifted from the small part focus to a bigger picture. *With fresh eyes, you can find mistakes and errors in the logic of development, missing requirements that needed to be set, but nobody had the idea in the first place that it can affect the product*.

The more time and effort invested in meaningful testing activities, the more reliable the product is. You will be able to guarantee, to some extent, that every part of software (product) and all its integrated parts are working as they should. You can assure your customers that you have made everything that you have committed to make and that everything works as it was intended, that your product has good quality. From this comes the term **quality assurance (QA)**. ISO 9000 standard.

There are a lot of misconceptions about what is software testing and what testers actually do. Testing is a skill and it's constantly evolving, so let's see what software testing should be:

- **Investigation** - We may not always know what the outcome will be, but it's our job to uncover information that helps people make decisions. It is much more than comparing against a specification that has an expected result. We need to think critically, ask difficult questions, pick up on risks, notice those things which at first glance seem inconsequential, yet on close examination are much more important and need investigating further.
- **Exploration** - Regardless of how comprehensive requirements are, they will never be an exhaustive list. You won't know everything the software will do up front. That's where exploratory testing comes in.
- **Mitigation** - one of the reasons we test, is to discover issues, risks, and other information about a software product, enabling action to be taken so that the end user is not unpleasantly impacted by them. This action might be:
    - Fixing bugs
    - Re-assessing and changing the original requirements
    - Providing user assistance within the product
    - Creating user documentation
    - Communicating known issues to stakeholders
- **Valuable** - Software testing is a valuable activity in software development but often misunderstood due to its unpredictable and creative nature. The lack of countable things created by testers is one reason some people like to use test cases as a way of measuring - they are a tangible, countable output. The value of testing extends beyond test cases. The testing carried out during exploratory testing sessions may not necessarily provide a defined set of test cases, however, the tester often finds more interesting bugs by not following a scripted path.
- **Communication** - A massive part of a tester's job is communication. Testers provide information about the quality of a software product, so it's important we communicate this information accurately to enable the right decisions to be made. Someone can start as a tester with few technical skills, but a real ability to communicate with others and be clear about what it is you are saying is vital. As testers, we need to make sure we use the correct words and phrasing so as to not be ambiguous, and to remove the risk of misunderstanding. What you mean to say isn't always what you end up saying, and often assumptions are made, and incorrect actions are taken as a result of poor or insufficient communication.
- **Potentially Infinite** - All testing is a sampling. For every nontrivial product, there is an unimaginable number of parameters with a great number of possible values. How do you know you are testing the important ones? We can't test everything. It's part of our job to make the decisions about what to test, understand the consequences of only testing those things, and be able to explain our decisions.

Software testing shouldn't be perceived only as a task where the tester works through a list of pre-prepared tests or test cases giving a firm pass or fail result. If you have a user story or set of requirements, it is, of course, important to make sure what you are testing adheres to those things, however, it can be helpful to reframe acceptance criteria as 'rejection criteria'. When the acceptance criteria are not met, the product is not acceptable, but if they are met,

that doesn't mean the product has no issues. We will talk more about the acceptance criteria in the upcoming chapters.

- **Acceptance Criteria Definition 1**: "Conditions that a software product must satisfy to be accepted by a user, customer or other stakeholder." (via Microsoft Press)
- **Acceptance Criteria Definition 2:** "Pre-established standards or requirements a product or project must meet." (via Google)

1.1.1 Some of the most notable defects

a) In 2009, one of Google developers added a single slash to a list of malicious sites, which caused every single site to be listed as malicious. You can read a bit more here:

https://googleblog.blogspot.com/2009/01/this-site-may-harm-your-computer-on.html

b) In the case of Nasa's Mars Climate Orbiter, human error led to losing this very expensive piece of technology. The craft was supposed to get into the Mars atmosphere and stabilize its position there. Unfortunately, it crashed. That happened because some of the software calculations were done in the metric system and a part of the software done by a sub-contractor engineering firm didn't convert from English units to metric.

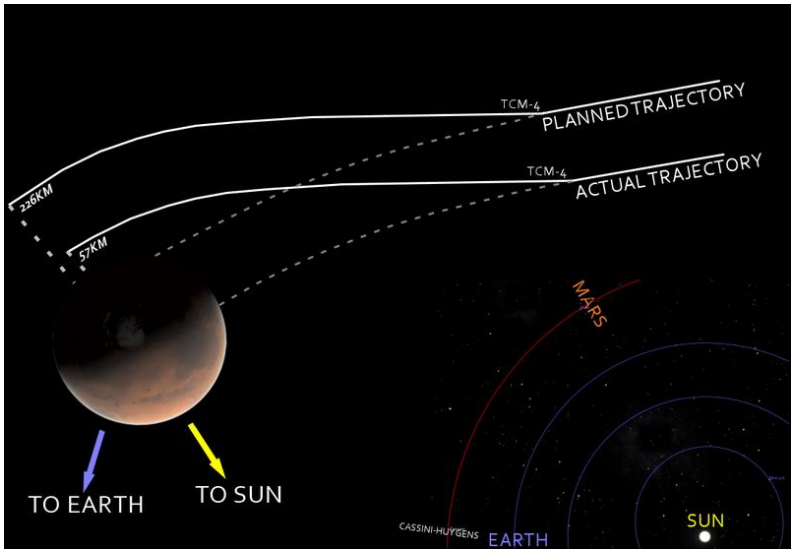


Figure 1. Nasa climate orbiter trajectory

c) Heathrow Airport T5 baggage system was tested with over 12000 luggage bags prior to going live and it worked flawlessly. On the day of the release, it got stacked and caused serious delays. The system didn't know how to handle a simple thing like what to do if a passenger forgot something in the bag and if the bag was removed, and then put back. This caused system to halt.

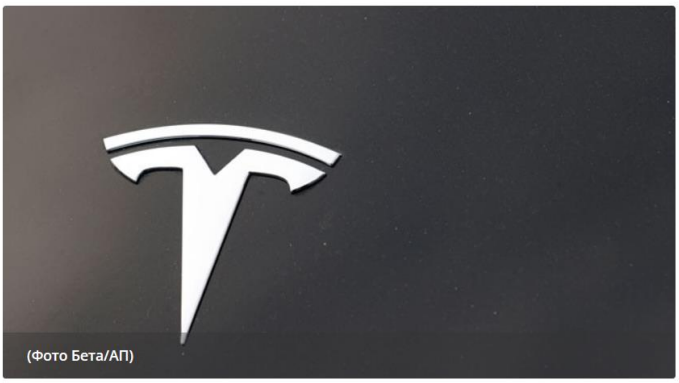
d) While testing we must keep in mind that our system might affect other systems and vice versa. How serious consequences may be, we can see in the EDS child support system. In 2004, EDS introduced a highly complex IT system to the U.K.'s Child Support Agency (CSA). At the exact same time, the Department for Work and Pensions (DWP) decided to

restructure the entire agency. The two pieces of software were completely incompatible, and irreversible errors were introduced as a result. The system somehow managed to overpay 1.9 million people, underpay another 700,000, had US$7 billion in uncollected child support payments, a backlog of 239,000 cases, 36,000 new cases "stuck" in the system, and has cost the UK taxpayers over US$1 billion to date.

As we can see from these few examples, quality testing of individual components, their integration with other components and integration of the connected systems is very important in order to get the desired goals of our product or software.

Четвртак,
22.09.2022. у 23:35

„Тесла" повлачи преко милион возила у САД због грешке



Вашингтон – Компанија за производњу електричних аутомобила „Тесла" повлачи 1,1 милион возила у САД због грешке у систему за аутоматско спуштање прозора, што може довести до тога да возило не открије препреку и да дође до повреде.

Како Танјуг преноси, систем за аутоматско спуштање прозора ради тако што спречава повреде путника у возилу – заустављањем прозора ако се наиђе на препреку.

Tesla povlači 363.000 automobila zbog greške u softveru za autonomnu vožnju

Može da se desi da vozila prođu pravo kroz raskrsnicu, i to iz trake za skretanje, da se ne zaustave ispred znaka STOP ili neoprezno prođu kroz žuto svetlo na semaforu.

IZVOR: KLIX.BA | PETAK, 17.02.2023. | 10:24 -> **10:30**

Свиђа ми с Твитуј Podeli

GeoTech Cues | August 6, 2024

The Great IT Outage of 2024 is a wake-up call about digital public infrastructure

By Saba Weatherspoon and Zhenwei Gao

On July 19, the world experienced its largest global IT outage to date, affecting 8.5 million Microsoft Windows devices. Thousands of flights were grounded. Surgeries were canceled. Users of certain online banks could not access their accounts. Even operators of 911 lines could not respond to emergencies.

The cause? One mere faulty section of code in a software update.

The update came from CrowdStrike, a cybersecurity firm whose Falcon Sensor software many Windows users employ against cyber breaches. Instead of providing improvements, the update caused devices to shut down and enter an endless reboot cycle, driving a global outage. Reports suggest that insufficient testing at CrowdStrike was likely the cause.

However, this outage is not just a technology error. It also reveals a hidden world of digital public infrastructure (DPI) that deserves more attention from policymakers.

According to data from Parametrix, the global IT outage linked to CrowdStrike is likely to have caused at least **$5.40 billion** in direct financial losses for Fortune 500 companies, excluding Microsoft. Cyber insurance is expected to cover only 10% to 20% of these losses.

1.2 Seven Testing Principles

Principles defined here are not the only one that exists. There are new ones that emerged with Agile methodologies. However, these below are still valid, and we will mention others.

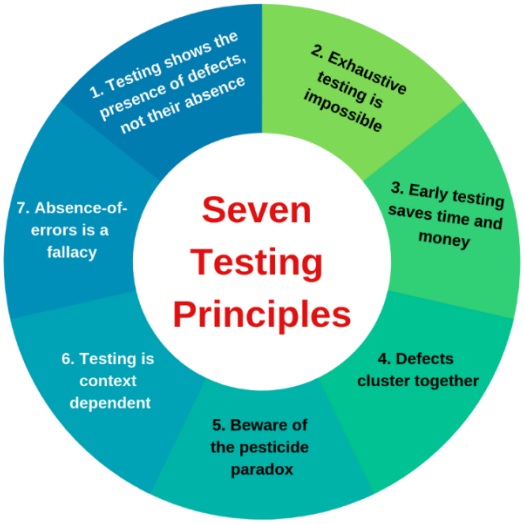


Figure 2: Seven testing principles

1.2.1 Testing shows the presence of bugs

Testing shows that defects are present but cannot prove that there are no defects. This was first coined by Edsger W. Dijkstra in the late 1960s. We can never be 100% sure that our software is bug free. While testing and application or component, we can only show that some part or module isn't working the way it is supposed to, but we can't show that it won't work correctly in every single possible case.

"Simplicity is a great virtue, but it requires hard work to achieve it and education to appreciate it. And to make matters worse: **complexity** sells things better." Edsger W. Dijkstra

1.2.2 Exhaustive testing is impossible

Testing all combinations of inputs and preconditions is virtually impossible when we have in mind that we aren't talking about trivial cases. There are testing technics and risk analyses that we can use to maximize our confidence in the quality of the software or component and prioritize testing efforts, but every single thing can't be tested due to the complexity, time constraints or some other factor we are faced with.

1.2.3 Early testing

Early testing is necessary, or at least good practice for a few reasons. When testing is done early, testers get to know the product early and they can directly impact the development with their insight. Defects and errors found early are easier to fix, and that approach saves money.

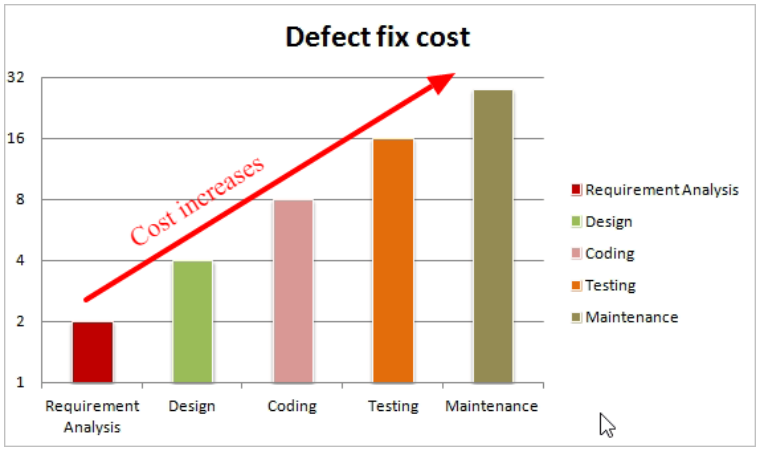


Figure 3. Cost of fixing defects in different stages of development

https://www.softwaretestinghelp.com/7-principles-of-software-testing/

1.2.4 Defect clustering

A small number of modules usually contain most of the defects discovered during pre-release testing or are responsible for most of the operational failures. Predicted defect clusters, and the actually observed defect clusters in test or operation, are an important input into risk analysis used to focus the test effort.

Defect Clustering in Software Testing is based on the Pareto principle, also known as the 80-20 rule, where it is stated that approximately 80% of the problems are caused by 20% of the modules.

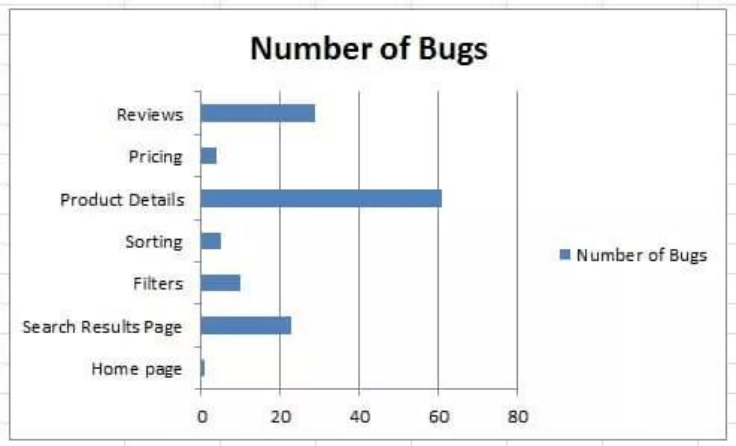


Figure 4: Defect clustering sample

In this image, the principle of defect clustering in software testing can be seen that most of the defects are clustered in particular sections of the application e.g., Product Details, Reviews and Search Results Page for an e-commerce website and the remaining defects in other areas.

1.2.5 The pesticide paradox

The phenomenon that the more you test software, the more immune it becomes to your tests – just as insects eventually build up resistance and the pesticide no longer works. If we write our tests once, don't maintain them and just execute them, eventually, they will stop finding bugs. This is the case where the project evolves, and our tests become outdated. A good practice is to maintain our tests as the project progresses, include new and fresh ideas, we need to try to use new valid and invalid test data to test the behavior of software, documentation or module. Even a new test should be written periodically. Like insects evolve to, and pesticides aren't effective in killing them, so happens with the tests if the project progresses, and the tests aren't maintained.

1.2.6 Testing is context dependent

Testing is done different in different contexts. For example, safety-critical industrial control software is tested differently from an e-commerce mobile app (figure 5). As another

example, testing in an Agile project is done differently than testing in a sequential lifecycle project.

Figure 5. Context depending testing

1.2.7 Absence of errors fallacy

Some organizations expect that testers can run all possible tests and find all possible defects, but principles 2 and 1, respectively, tell us that this is impossible. Further, it is a fallacy (i.e., a mistaken belief) to expect that just finding and fixing many defects will ensure the success of a system. For example, thoroughly testing all specified requirements and fixing all defects found could still produce a system that is difficult to use, that does not fulfill the users' needs and expectations, or that is inferior compared to other competing systems. This can mean that the system is intuitive for use from an engineer's perspective, but for the user, that doesn't have specific knowledge it can seem counter intuitive, or there is more efficient or aesthetically pleasing product on the market.

Figure 6: Absence of errors — user unable to do basic actions

Coming to the absence of errors fallacy, imagine you developed an e-commerce system and tested it completely. All the identified defects have been fixed and retested. Let's say 99% of

the defects have been rectified. Management is pretty sure about the product quality with respect to defects. Now when the system is demonstrated to a client, what if you get feedback saying "Though it is so-called defect-free, still this is not what I wanted. I wanted a simple UI that can handle the user load"?

Everybody in the Test team was confident about the product 'quality' (absence of errors) but in the end, it proved to be false (fallacy) the system is not usable; it doesn't fulfill the client's expectations.