INTRODUCTION & MOTIVATION

" Software is everywhere today. It leads every our step. It is a part of everything we do. The software makes our everyday work easier, simplifies our daily lives*"* [1]. This is the tagline of Unicorn, a European software company based in the Czech Republic that aims to provide complex information systems and solutions in the field of tech. Upon thinking, their statement is correct.

Software *is* indeed everywhere as it is used in every aspect of our daily lives from the devices we use to the services we rely on. For example, to communicate we use messaging apps like WhatsApp and Facebook Messenger, for directions we rely on GPS services like Google Maps or Waze, we book and manage our doctor's appointments directly through smartphones, to practice our linguistic skills we use Duolingo and similar apps and to manage our finances we use banking apps. All these examples show how widely software affects our day-to-day activities and improves our standard of living.

However, despite its immense potential, software often contains flaws, requiring considerable resources for maintenance and repairs. For instance, in 2021, a massive Facebook outage [2] shook the internet and disrupted communication due to mistakes made during maintenance on the company's network.

This incident cost around $65 million in lost advertising revenue and is a stark reminder that even the most prominent platforms are not immune to software bugs.

Other noteworthy catastrophic occurrences caused by software bugs include the catastrophic explosion of the Ariane 5 rocket [3], just seconds after liftoff, and the chaotic malfunctioning of the baggage handling system in 2008, during the opening of Heathrow Terminal 5 [4] in London.

The latter one in particular, led to the loss of thousands of passengers' belongings, with approximately 42,000 bags misplaced by British Airways over ten days.

At times, software flaws can pose actual real threats to people's lives.

In 1982, the software controlling the Therac-25 radiation therapy machine (used for chemotherapy) *"*contained bugs which proved to be fatal*"* [5]. The lethal radiation doses emitted by this device, unfortunately, led to the death of multiple cancer patients and left some with lifelong injuries. Additionally, in October 1992, the London Ambulance Service [6] encountered software problems, disrupting their emergency responses and resulting in delays in dispatching ambulances to people in critical need of medical assistance.

These events highlight how critical it is to find and address bugs early on to guarantee the safety, efficiency, and reliability of software in our daily lives. They serve as evidence of the potential devastation that software bugs can cause, impacting people and economies on a large scale, and often losing public trust in these systems.

Maintaining the health of their codebase while also striving for efficiency and resilience is a challenging task for developers as it requires continuous dedication to quality assurance.

The examples of software defects and their widespread consequences mentioned above have made it evident that the root cause often lies within the code itself.

Code takes shape within IDEs, hence the ability to visualize and comprehend information within these environments is crucial.

In response to these needs, I present the Code Health Dashboard (full wireframe provided in appendices – however screenshots are presented within the report as well).

The data to build this dashboard can directly be gathered from the IDE itself or with the use of external tools such as JaCoCo (code coverage tool for Java applications).

This report will discuss the design concept for the dashboard and provide justification using the popular IDE, IntelliJ IDEA, as an example.

DESIGN ANALYSIS

While many IDEs include tools to assist developers in finding and fixing bugs and mistakes, they often fail to consider how important it is to visually represent this data. IDEs like IntelliJ provide tools for testing, detecting errors, improving code and debugging. However, these tools may not always be immediately straightforward or visually intuitive to developers, often requiring them memorizing the various actions to be taken.

1. Test coverage visualisation

For example, in IntelliJ, to run tests, a developer must right-click on a test class or method, select the specific test to run or choose to run all tests. Similarly, when examining test coverage, developers need to go through a multi-step process: navigate to the "run" menu, select "edit configurations," and set up a new configuration (e.g., JUnit). Additionally, they must ensure that the "enable code coverage" option is checked in the relevant section.

Both of these processes can be viewed as counter to Jakob Nielsen's usability heuristics [7], particularly "Visibility of System Status" and "Recognition rather than recall." They place unnecessary cognitive and operational burdens on developers (requires remembering sequences of actions, which can be burdensome impacting the workflow efficiency). The lack of an intuitive, one-click solution for running tests and for examining test coverage, means that developers may not have a clear and immediate understanding of the system's status.

To address these issues, the dashboard will feature a test coverage visualization in the form of a pie chart (see Fig. 1 below), providing a visual representation of the overall project's test coverage. This pie chart consists of distinct segments and "sub-segments", each denoting the percentage of code that is tested and untested.
For example, as shown in Fig. 2, 60% of the code is covered by tests, with Apple.java having 80% of its methods tested and 20% untested as shown in the red segment.

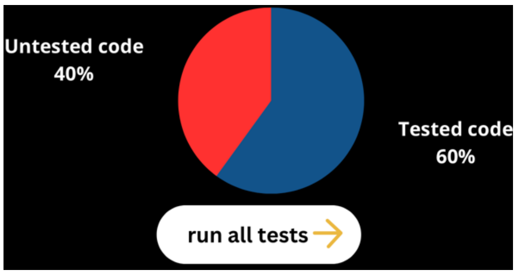


Figure 1

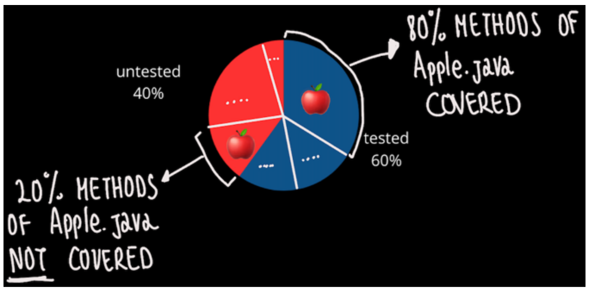


Figure 2

What makes this visualization particularly effective is its interactivity. Users can click on the segments, directly navigating to the specific methods or classes covered or not by tests.

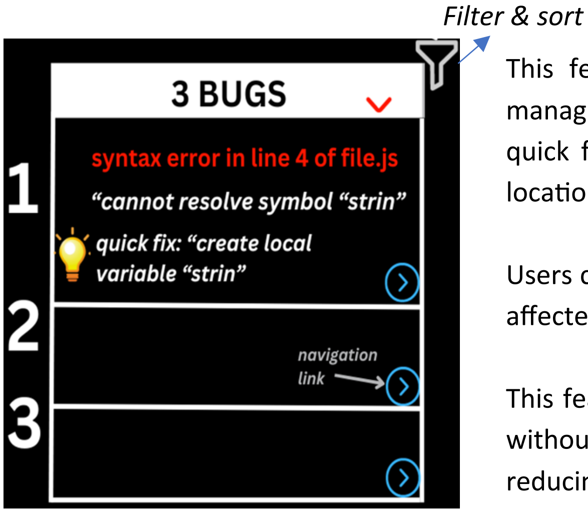
Additionally, for user convenience, a one-click 'Run All Tests' button (see Fig.1) will be provided below the chart, simplifying the initiation of testing procedures and further enhancing the overall user experience.
The pie chart provides an immediate overview of the codebase's testing status and it encourages developers to pay more attention to this critical aspect of code quality, reducing the potential for bugs.

2. Error navigator

When a syntax error occurs in IntelliJ, the IDE visually distinguishes it in red.
Additionally, a corresponding underline appears on the left side of the code editor, indicating the file where the error is located.
However, clicking on the file only opens it without actually directing you to the specific line where the error is located.

This challenge becomes more pronounced when working with extensive codebases, as identifying a single, small red-highlighted word can be time-consuming.
While IntelliJ does offer a unified system for listing these issues and providing direct navigation to each one, it may not be as visible or prominent due to its size (small red circle with an exclamation mark on the top right corner of the page). Moreover, it doesn't have a prioritization mechanism for the listed bugs, and any solution suggestions are offered at the line level, not within this list.
As a result, finding and addressing errors, even with this existing feature, can be time-consuming.

To address this, my dashboard will include an "error navigator" (see Fig. 3) in the form of a dropdown menu. Total number of bugs present in the project will also shown on the top.



syntax error in line 4 of file.js
"cannot resolve symbol "strin"
quick fix: "create local variable "strin"

This feature will serve as a centralized location for viewing and managing all code errors, including the type, description, line number, quick fixes and navigation links next for direct access to the exact location of the error.

Users can filter and sort errors based on criteria like type, severity, or affected class, allowing for focused attention on specific issues.

This feature allows developers to quickly identify and address issues without having to manually go through multiple files one by one, reducing the risk of oversight.

3. Code smell detector

The dashboard will also include a "code smell detector".
According to Martin Fowler, a renowned author and software engineer, "a code smell is a surface indication that often hints at a deeper problem in the system" *[8]*. Hence, code smells serve as early warnings, prompting developers to address underlying issues before they escalate. For the purpose of this report, I will only focus on one code smell as an example: duplication.

Duplicated code in a software project not only affects code quality but also increases the technical debt associated with it [9] cause maintaining duplicated code becomes increasingly harder throughout the project.

In IntelliJ, you can locate duplicated code through the main menu by selecting "Refactor" and then choosing the "Find and Replace Code Duplicates" option. However, this method lacks a clear visual representation of where the duplicated code exists in the codebase. It only prompts a popup message asking if you wish to replace a piece of duplicated code, hence developers may struggle to understand the full extent of duplication as they'll only rely on these messages.

To address this, the dashboard will not only include the total number of duplicated but also represent the duplicated code as a heatmap (see Fig. 3). In this heatmap, each code file is represented as a block (multiple files can be represented by one block), and the colour intensity of each block corresponds to the degree of code duplication in that file or module. This allows developers to prioritize their efforts by focusing on blocks with more intense colours, indicating higher levels of duplication (i.e. red block in Fig. 3 indicates highest number of duplications in the project: 7 duplications in Grape.java and Apple.java).

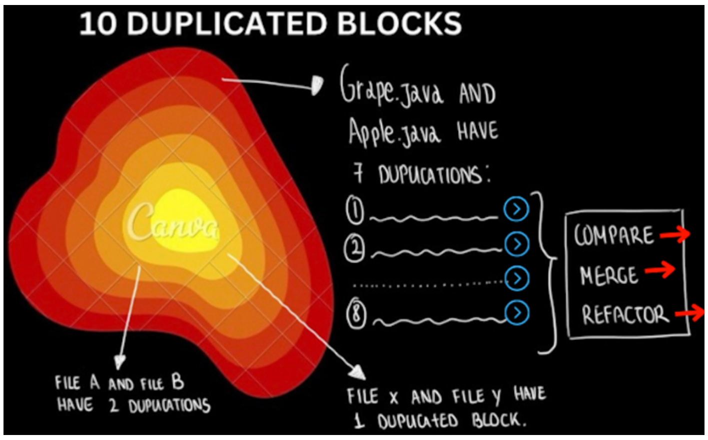

10 DUPLICATED BLOCKS
Grape.java AND Apple.java HAVE 7 DUPLICATIONS:
①
②
⑧
COMPARE
MERGE
REFACTOR
FILE A AND FILE B HAVE 2 DUPLICATIONS
FILE X AND FILE Y HAVE 1 DUPLICATED BLOCK.

Figure 3

The heatmap will also be interactive, allowing developers to click and zoom on a specific block for closer examination of the corresponding files and take actions to compare, merge, or refactor directly from the heatmap.

This is a valuable tool as it allows developers to visually pinpoint areas with duplicated code, facilitating the consolidation of duplicated code, and ultimately enhancing code quality and maintainability.

4. *To-do list*

Figure 4

Developers often have to multi-task.
To organise their work, set priorities and stay on top of their tasks without using any other tool (i.e. calendars, planners etc.), I decided to include a to-do list on the dashboard (see Fig. 4).

This enhances information visualisation as it makes it easier for developers to visually prioritize their tasks and track progress, with a clear overview of completed and pending tasks.

5. *Customisation*

Finally, the dashboard will incorporate two more features that align with Nelson's "Flexibility and Efficiency of Use" heuristic [7].
Firstly, developers can create project-specific metrics, so that the dashboard caters to their software development objectives i.e. they may want to have an interactive mind map of all dependencies in their dashboard (see wireframe in appendices).
Secondly, they can personalize the appearance and layout, according to their unique preferences.
These features allow a flexible UI that can be tailored in the way developers want.

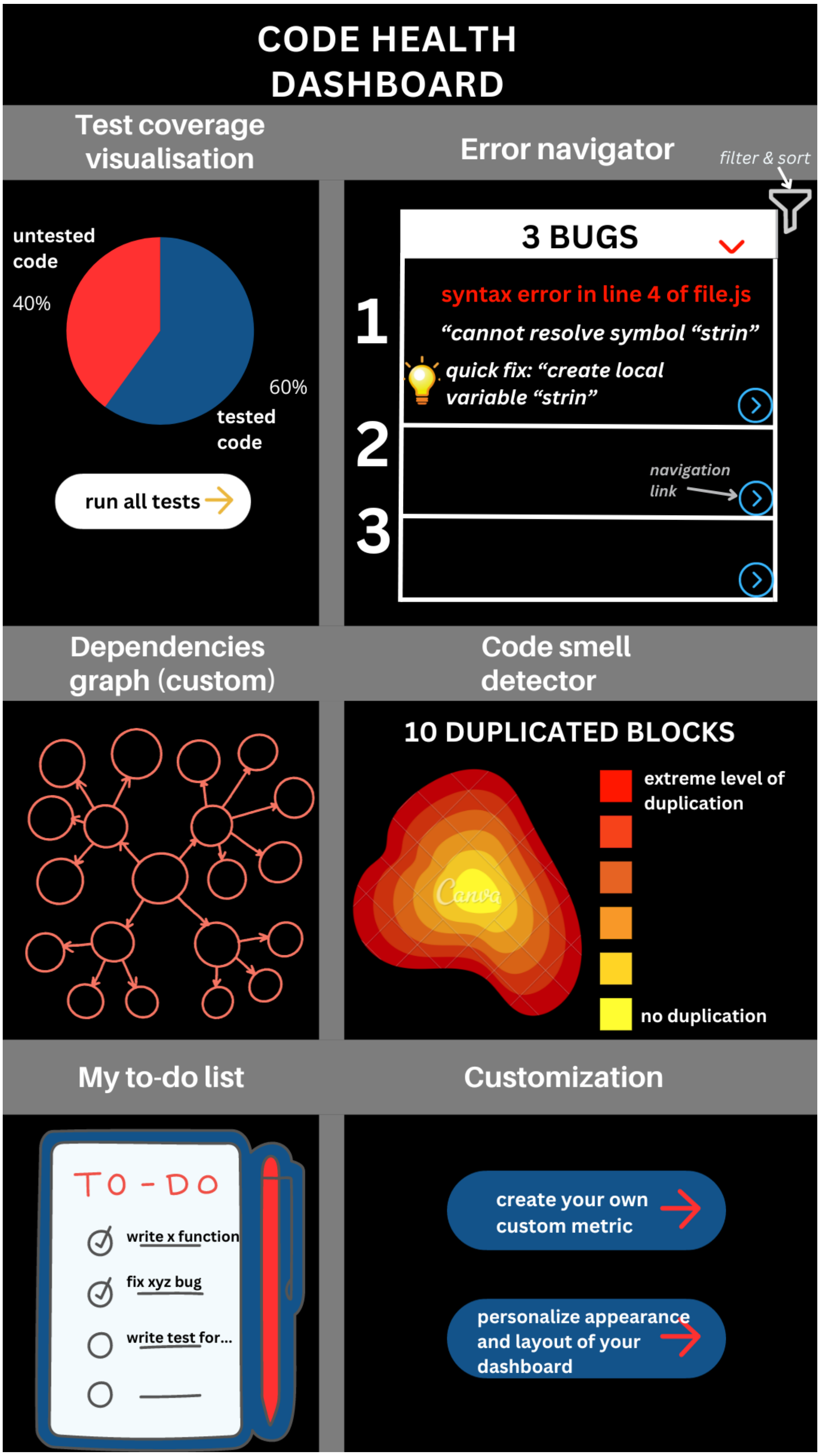
CONCLUSION

In conclusion, the Code Health Dashboard visualises critical information on a single page.
While there are similar tools available in the market, they mostly present such information with numbers, which can be difficult to interpret.
My idea, in contrast, uses different kinds of visual elements such as a test pie chart, an error navigator, a code smell heatmap and a to-do list, which have been proven to be great tools for presenting complex information.
Studies have in fact shown that "90% of the information transmitted to the brain is visual as visuals are processed 60.000X faster in the brain than text" [10].

Overall, these figures/visual aids will help identify and address errors and vulnerabilities, prioritise tasks and address developers' unique needs and preferences through the customisation options.
Hence, the primary aim is to assist developers in improving the quality and resiliency of their codebases, thereby preventing incidents and disruptions, similar to those mentioned earlier.

APPENDICES

Full wireframe of the code dashboard


CODE HEALTH DASHBOARD
Test coverage visualisation
untested code
40%
60%
tested code
run all tests
Error navigator
filter & sort
3 BUGS
1
syntax error in line 4 of file.js
"cannot resolve symbol "strin"
quick fix: "create local variable "strin"
2
navigation link
3
Dependencies graph (custom)
Code smell detector
10 DUPLICATED BLOCKS
extreme level of duplication
no duplication
Canva
My to-do list
TO-DO
write x function
fix xyz bug
write test for...
Customization
create your own custom metric
personalize appearance and layout of your dashboard

References

1. *Software Everywhere* (no date) *Unicorn*. Available at: https://unicorn.com/en/software-everywhere

2. Raygun Blog. (n.d.). *How much could software errors be costing your company?* [online] Available at: https://raygun.com/blog/cost-of-software-errors/

3. Holzman, E. (2017). *The Worst Computer Bugs in History: The Ariane 5 Disaster*. [online] BugSnag. Available at: https://www.bugsnag.com/blog/bug-day-ariane-5-disaster/

4. The Independent. (2008). *Disastrous opening day for Terminal 5*. [online] Available at: https://www.independent.co.uk/news/uk/home-news/disastrous-opening-day-for-terminal-5-801376.html

5. Fabio, A. (2015). *Killed By A Machine: The Therac-25*. [online] Hackaday. Available at: https://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25/

6. Long, T. (2009). *Oct. 26, 1992: Software Glitch Cripples Ambulance Service*. [online] Wired. Available at: https://www.wired.com/2009/10/1026london-ambulance-computer-meltdown/

7. Nielsen, J. (1994). *10 Heuristics for User Interface Design*. [online] Nielsen Norman Group. Available at: https://www.nngroup.com/articles/ten-usability-heuristics/.

8. martinfowler.com. (n.d.). *bliki: CodeSmell*. [online] Available at: https://martinfowler.com/bliki/CodeSmell.html.

9. Codegrip. (2019). *What is duplicate code?* [online] Available at: https://www.codegrip.tech/productivity/what-is-duplicate-code/

10. Admin (2021) *Studies Confirm the Power of Visuals to Engage Your Audience in eLearning*, *www.shiftelearning.com*. Available at: https://www.shiftelearning.com/blog/bid/350326/studies-confirm-the-power-of-visuals-in-elearning#:~:text=Visuals%20have%20been%20found%20to,to%20process%20the%20information%20faster