Winter 3-2022

Classifying Dead Code in Software Development

Arman Alavizadeh
Portland State University

Classifying Dead Code in Software Development

by

Arman Alavizadeh

An undergraduate honors thesis submitted in partial fulfillment of the

requirements for the degree of

Bachelor of Science

in

University Honors

and

Computer Science

Thesis Adviser

Warren Harrison

Portland State University

2022

I. Introduction

The ARIANE 5 rocket exploded less than a minute after its launch mid-air on June 4th, 1996[1], promptly piquing public interest and a full-blown investigation was held. Findings found that the failure point was an error in the Inertial Reference System's software which misguided the rocket[1]. Code that was thought to be dead (non-executing during runtime) was being run contrary to the belief of the ARIANE software engineers and had caused the chain of events leading to the explosion[1]. A scenario for the rocket that had seemed highly unlikely during development had occurred at launch, executing the supposedly dead code within the Inertial Reference System[1]. This dead code cost the European Space Agency $370 million USD[2] and development time.

Although the majority of dead code's effects in the software engineering world are not as costly as the ARIANE 5 incident, the incident does highlight the importance of understanding and eliminating dead code, a problem that has evolved in tandem with software development.

But how do we understand dead code from the perspective of a computer scientist or a software developer? At the highest level, dead code can be abstracted as, "[...] any code that's never executed, or if executed, the execution has no effect on the application's behavior"[4]. Such an open-ended definition leaves a lot of room for interpretation. Examining dead code at a lower level of granularity gives us narrower definitions: Code that is never executed in a called function or a function that is never called. Both of these examples would be considered unreachable dead code, "Code that can't be reached or executed on any program path is called unreachable code"[4]. Unreachable code being a subset of dead code.

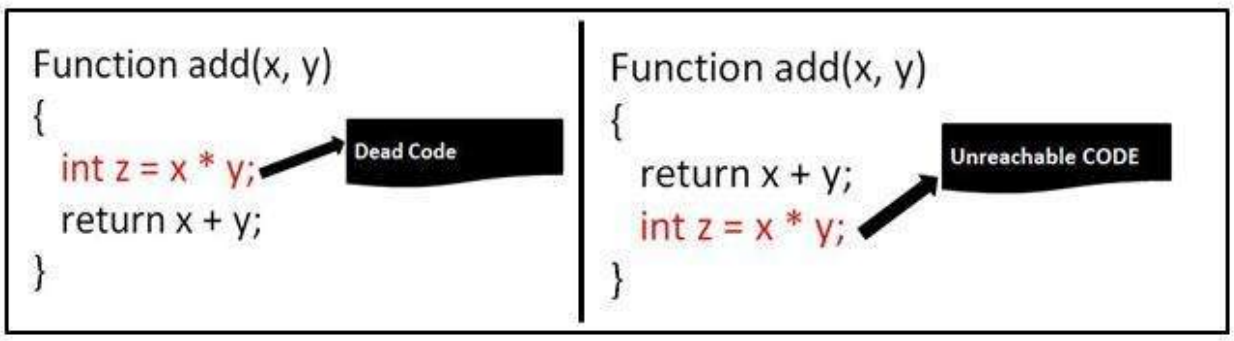

Figure 1 - Unreachable code [5]

As seen in Figure 1, we can see examples of dead code and unreachable code in the guise of a simple addition function. What int z is doesn't matter since it is not relevant to the return value of the add(x,y) function, z could be anything and we would never use it. The difference between these two examples is that z is computed but never used since it is before the return statement. Z is never computed for the unreachable code example because the return statement is before Z's initialization, hence that line of code will never execute under any circumstance.

For a more complex example, there is plenty of method stub code that could be considered dead. Method stubs commonly refers to code that has not yet been fully developed or simulates the behavior of existing code[6]. It's useful to simulate needed functionality while developing other aspects of the codebase.

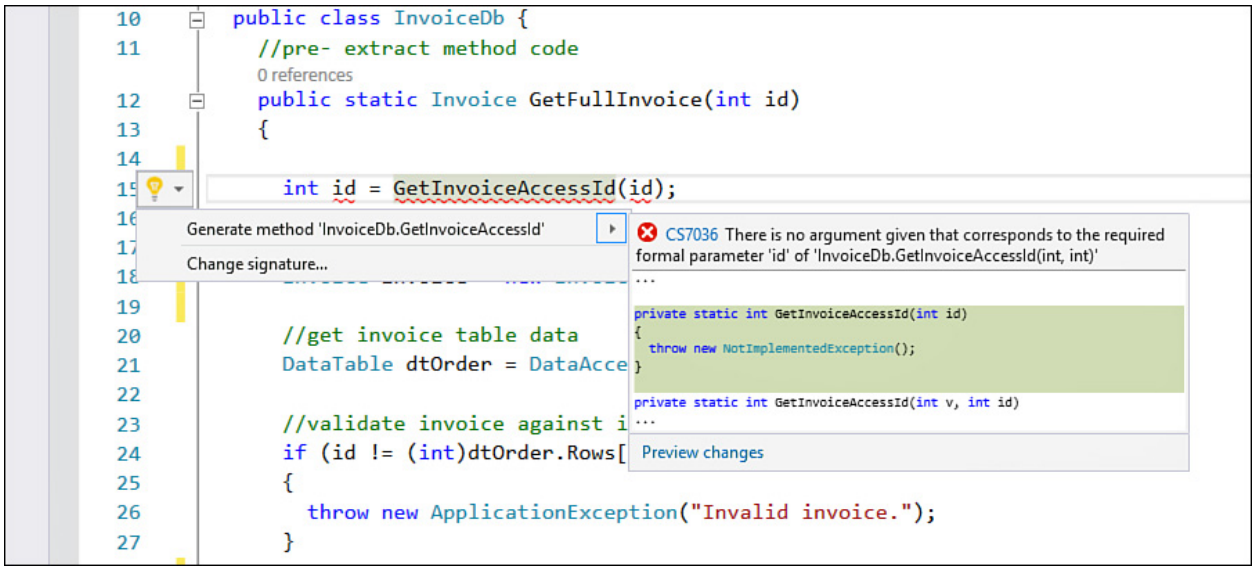

Figure 2 - Method Stub Sample Code [7]

Re-prioritization is common in the software engineering world. Software functionality is dictated by key stakeholders, meaning that required functionality can change with their desires. Say we have a project that has postponed development of certain features since said key stakeholders want something else out of the product. In this scenario, how would we determine whether or not a method stub is considered dead? If there are leftover method stubs from an upended feature still present in the re-prioritized codebase, we could have dead code. Remember that dead code can be code that doesn't contribute value to our software. Everytime we change the deliverables of our software, we must think of how our code is relevant towards those deliverables.

Simone Romano et al noted that around sixteen percent of all methods from the open source Java applications they studied were dead**,** compared to their median of around twelve percent across their studied codebases[8]. Due to the nature of open source projects, these methods may yet be implemented by their respective developers and serve as the coding equivalent of a method stub. It is impossible to know what portion of the aforementioned sixteen percent are waiting to be implemented. Ultimately, a method stub can be a bad code smell since other developers are guessing what your intentions are and whether or not you decide to follow up on them.

In an extreme case of the effects of method stubs, many customers of Chemical bank in New York lost a collective total of 150 million USD overnight due to a withdrawal error[9]. A commented out portion of code updating the cash machine software

was the culprit[9], causing behavior which the developers thought was not possible until it happened and became an issue. It's impossible to know what the development team was thinking at the time and whether or not they thought the commented code could be a problem, but what we do know for certain is that the same code was neglected. Code behavior is not so easily defined until it is ran in a user context and even then only specific scenarios might trigger unexpected behavior.

Everyone can agree on the definition of dead code at the highest level of abstraction. With what we have seen with the previous examples, it's challenging to translate that definition down to lower levels of granularity. Subjectivity often plays a part when identifying dead code. Inconsistencies in dead code classification will happen when we identify dead code with subjective definitions, substantiating the need for more objective dead code sub-definitions. In order to get more objective definitions of dead code sub-types, a taxonomy is needed.

II. Taxonomizing Dead Code

Taxonomizing dead code benefits our understanding of it for several reasons: Systematic breakdown of certain types of dead code help us categorize for a given range of granularity. Taxonomies ensure consistency between definitions which has been lacking at lower levels of granularity for dead code. Being able to classify and categorize dead code across different contexts clarifies dead code beyond its abstract definition. Taxonomies provide a point of reference that can be used in identifying dead code as somewhat of a standard. Standards evolve over time, and our taxonomy would as well, but an initial version must be instantiated first. The goal here is to not be as comprehensive as possible, as that could potentially be multiple papers, each necessitating in-depth research. Finding a method for discovering more specific definitions of dead code is the focus. A way to taxonomize dead code, if you will.

Before classifying dead code subtypes, let's establish a basic taxonomy for dead code from what we currently know:

1. Dead Code
    - A. Unreachable Code
        - 1.A.1 Preceded by an Unconditional Control Flow Change
        - 1.A.2 Preceded by a Conditional Control Flow Change that can never be true
    - B. Unused Code
        - 1.B.1 Commented Out Code
            - 1.B.1.1 Block/Multiline Comments
            - 1.B.1.2 Individual Line Comments
        - 1.B.2 Unused Files
        - 1.B.3 Undefined Functions

Figure 3 - Basic Taxonomy

Our taxonomy provides specific, objective classifications for dead code. When classifying dead code, we have to think about levels of granularity. For example, we know that figure 1 represents unreachable code. The return statement occurs before z is ever calculated, meaning the control flow change was unconditional. If we instead had a conditional to calculate z before the return statement such as "IF (A > B AND A = B) THEN int z = x * y", z would still never be calculated. The difference is that our conditional is always going to be calculated as false, whereas in the original example the line calculating z was never executed. Since we have two different ways of creating unreachable code, we then have two different types of unreachable code for our taxonomy. Hence, a deeper level of granularity. Going forward, we will be adding new classifications to this taxonomy.

Deadcode's effect on a program's behavior is unknown unless an issue is apparent enough to warrant investigation. If there is dead code but there are no side effects to its presence ("benign" dead code), then it may as well not exist until a potential issue arises in the world of web development. On the other hand, even benign dead code is an issue in embedded systems. Many embedded systems (dominantly written in C) rely on extensive code validation and optimization due to many industrial and reliability standards placed on the codebase[10]. Compilers generally are not thorough enough in catching dead code for the aforementioned systems which is why techniques such as abstract interpretation and counterexample guided abstraction refinement are utilized[10]. Code optimization and validation coverage are not as important in the higher level segment of the industry such as web development compared to embedded systems where certain criteria have to be met for the codebase to be considered sound.

Although there are different types of dead code, the usage of the term "dead code" differs between different segments of the industry. On the web development side, it is increasingly common to see libraries pulled in where only a few functions from a given library or framework are used with researchers such as Obbink working on a solution[3]. Obbink et al's foray into web development dead code focused on unused features within JavaScript frameworks by implementing a dead code removal tool that could use any definition of dead code as long as it could make a call graph of the JavaScript codebase[3]. Using bloated frameworks is a source of major overhead for the JavaScript parsing engine and not all web browsers support modules[3]. Web developers are being limited in their coding guidelines due to this issue, such as an aversion to utilizing object reflection[3]. Obbink et al's approach with the call graph was to represent functions as nodes and the caller-callee relationship as edges between nodes, with dead code elimination based upon nodes disconnected from the global scope node[3]. However, it was noted even by Obbink and colleagues that events may not have a specified caller, and therefore many pieces of code that handle the event could be labeled as dead via their approach[3]. Even working on dead code at a specific language and sub-section of the industry presents challenges at a technical level.

How dead code is classified depends on the analysis technique given in the context of the call graph. No matter the technique given, the specific type of dead code to remove remains the same. JavaScript's dynamic typing makes it difficult to

statically analyze since variables do not have to be explicitly typed before use, meaning that dynamic analysis techniques would be dominant in use for Lacuna. Lacuna being the tool in development by Obbink et al. Analogizing this problem to a language like C, we would have library code in our codebase that wasn't in use causing bloat in compilation time. Although there are key differences to JavaScript and C, the type of dead code remains similar conceptually. Technical details of the languages change how we approach this situation. Whereas in C the static analysis capability of an optimized compiler would rout out unused library functions/data members, in JavaScript we cannot rely as heavily on static analysis. Analyzing the effects of unused dead code in both situations, compilation would take longer in C and parsing would take more time for the JavaScript interpreter, with binary size also being unnecessarily bloated for both as well.

Both of the above examples relate to unused imported code not made by the developers themselves. Hence, we can call this type of dead code unused imports, a subset of unused dead code revolving around the usage of libraries/frameworks not written by the developer(s). The approach to solving unused imports is different between our scenarios, but the conceptual understanding remains the same and can be translated between languages. Specifics and technical issues change with the language, but the concept remains the same. Meaning that we can use imported dead code as a classification. We've taken a higher level definition of dead code and defined it at a smaller scale. Let's revisit our taxonomy with unused imports included:

1. Dead Code
    - A. Unreachable Code
        - 1.A.1 Preceded by an Unconditional Control Flow Change
        - 1.A.2 Preceded by a Conditional Control Flow Change that can never be true
    - B. Unused Code
        - 1.B.1 Commented Out Code
            - 1.B.1.1 Block/Multiline Comments
            - 1.B.1.2 Individual Line Comments
        - 1.B.2 Unused Files
        - 1.B.3 Undefined Functions
        - 1.B.4 Unused Imports
            - 1.B.4.1 Unused Library Functions
            - 1.B.4.2 Unused Framework Features

Figure 4 - Taxonomy With Unused Imported Code Added

Our unused imports classification comes in two different types so far based on the research we have evaluated: unused library functions and unused framework features. More subclassifications of unused imports are possible, but our focus is on the method of classification rather than being comprehensive in our classification.

Neubauer et al worked on analyzing code coverage techniques for industrial C code in embedded systems[10]. Their work focused on how to differentiate dead code in the control and data portions of the C codebases, particularly those that are "reactive, control-oriented, and floating-point intensive"[10]. Differentiating dead code is harder in automatically generated embedded code since control code gets more mixed in with data compared to handwritten programs which use a loop as control[10]. A solution they found was to create a model of the codebase via various techniques in order to understand what code did not fit the data/control flow of the program[10]. Effectively, Neubauer et al. mapped out the logical flow of specialized embedded C code and identified dead code as code that disrupted the intended data/control flow of the program. By mapping out logical inconsistencies in the code, they turned dead code into a coverage problem, "how much of our code is inconsistent with our intent/purpose". Although this was done at the inline level, that doesn't mean this approach can't be used at other granularities.

Boomsma et al focused their research on dead code identification within the context of PHP web applications[11]. Due to technical constraints with PHP, they focused on creating dynamic analysis tools in the form of a web application and an eclipse plugin[11]. Boomsma and co determined that the file level of granularity would be best due to the ease of measuring file usage[11]. Hence, Boomsma and co turned their PHP dead code problem into a coverage based code problem as well. Of course some files may be under development (stubs) or they aren't supposed to be invoked often, so Boomsma et al also measured the frequency of file use and not just coverage[11].

Both of the aforementioned pieces of research focus on completely different segments of software development, yet they arrive at a fundamentally similar approach. Code found to not contribute to the intent of the software was labeled as dead. Even at different levels of granularity (inline vs file) the methodology did not change. Hence, we can call this type of dead code logically inconsistent code: Code which does not adhere to or contribute to the main logical flow of the program. Of course, it is not as black and white as the definition makes it sound like. Some files may not be regularly run such as files focusing on setups or specific, in-depth testing suites. However, these files still contribute to the main logical flow of the program, as they serve as a form of verification of the intended software behavior. Method stubs are unfinished and would be flagged as dead, but they aren't meant to contribute to the codebase until they are fleshed out. In reality, developers must still use their better judgment when identifying logically inconsistent dead code. There are methods of identifying logical inconsistencies, but none are perfect, which is why the developer(s) have the last word. Now that we have identified a new classification of dead code, let's see where it falls on our taxonomy:

1. Dead Code

    A. Unreachable Code

        1.A.1 Preceded by an Unconditional Control Flow Change

        1.A.2 Preceded by a Conditional Control Flow Change that can never be true

    B. Unused Code

        1.B.1 Commented Out Code

            1.B.1.1 Block/Multiline Comments

            1.B.1.2 Individual Line Comments

        1.B.2 Unused Files

        1.B.3 Undefined Functions

        1.B.4 Unused Imports

            1.B.4.1 Unused Library Functions

            1.B.4.2 Unused Framework Features

    C. Logically Inconsistent Code

        1.C.1 Logically Inconsistent Files

        1.C.2 Logically Inconsistent Functions

        1.C.3 Logically Inconsistent Individual Lines

Figure 5 - Taxonomy With Logically Inconsistent Code Added

Logically inconsistent code is placed in the second level of granularity in our taxonomy, but this is debatable. Both Neubauer et al. and Boomsma et al. focused on identifying dead code that did not follow the main logical flow of their codebases. Identifying the aforementioned specific type of dead code required monitoring code execution at different levels of granularity, which is similar to looking for unused code. However, logically inconsistent code isn't always unused, so I have put it at the same level of granularity as unused code rather than classifying it under unused code. Unused code comes with the expectation that it is never executed, not sometimes or rarely; never. This is not a definitive placing of logically inconsistent code as more research and discussion would be needed for that to happen since it is similar to unused code. If anything, this shows why dead code is difficult, yet important, to taxonomize.

These were just a few examples of marrying together different dead code identification techniques from research across the software engineering industry. Obviously, more research is warranted in this endeavor which would expand our means of

classifying dead code. Hopefully, this serves as good food for thought for how to taxonomize dead code at a narrower scale. Now our challenge comes in the form of how to utilize our narrower definitions in code analysis.

III. Analyzing Analysis Techniques

Code analysis is essential to any software development project. When mentioning code analysis, two particular types come to mind: static and dynamic code analysis. Static code analysis examines a codebase before the program is ran, usually against a set of rules or a coding standard's specifications[12]. Dynamic code analysis examines source code while the program is running, scanning for any crashes, memory leaks, etc[13].

If we were to catch dead code in a professional software development environment, these forms of analysis would be our first line of defense so to speak. Codebases grow ever larger especially as the intentions and roles of programs grow more grandiose. Developers cannot be tracking dead code manually as a routine, since development time would be wasted which makes manually tracking dead code an opportunity cost. Spotting dead code in dynamic/static code analysis becomes one of the most efficient solutions since it can be automated into a workflow. Static and dynamic analysis will not be assessed holistically, but rather in the context of dead code.

Before we assess dynamic and static code analysis, we must understand their limitations. Static code analysis cannot acknowledge developer intent, such as if a function is behaving as expected[12].

```
1  function square(x)
2  {
3      return x;
4  }
```

Figure 6 - Function Passing Static Code Analysis

In figure 6, we see that the function square is not actually doing anything to its argument x, it just returns the same value it was given. We expect a square function to return a squared value of the argument passed to it, which is not the case here. If you were to run static code analysis on square(x), it would pass since developer intent is undeterminable and subjective. Obviously we know that the function is not behaving as expected, but static code analysis cannot understand those expectations in the first place.

Dynamic code analysis has more nuanced limitations than static code analysis. Automation performing dynamic code analysis is only as good as the rule sets created by developers. Unit testing for example is as extensive as the developer(s) make the testing suite. Even if rule sets are perfect, pinpointing what in the source code is causing an issue can be difficult and open to developer interpretation.

Looking at static code analysis through the lens of dead code, there are two benefits we can observe. Static code analysis can easily detect issues such as unreachable code or dead code in code blocks like figure 1 easily and would correctly flag them as dead code. Static code analysis would be beneficial in identifying imported dead code. Novak et al focused their

work on taxonomizing four commonly used static code analysis tools: StyleCop, Gendarme, FindBugs, and CheckStyle[14]. Tools which can detect unreachable code, unused values and unused functions[14]. All tools support some sort of ruleset they can govern such as naming, performance, maintainability, etc with accompanied methods of configuration[14]. However, these rulesets are meant to enforce objective rules such as syntax for comments, amount of spaces for indentation, camel casing, etc. There is not an easy way to identify all forms of dead code on the spot, hence no ruleset would be able to identify all forms of dead code. Dead code can also take shape in different ways between codebases and languages. Static code analysis tools easily and precisely catch unused values/functions and unreachable code, but rulesets are constrained to code style and optimization. Optimization can include removing dead code, but there is no guarantee that all dead code would be removed. Perhaps static code analysis tools could be given a ruleset that defines logically inconsistent dead code for a given codebase. Set a certain type of dead code for removal and define the ruleset based on that. Such a ruleset would not be as extensible as a coding style standard, but rather be tailored for specific codebases.

Static analysis also has limitations in certain languages such as JavaScript due to the dynamism of the language, something which Obbink and co acknowledged in their research. Revisiting the earlier example of method stubs, static code analysis is not guaranteed to detect all dead method stubs. If a method stub is still being called/used but that piece of functionality is not relevant to the product anymore, it should be marked as dead code. Static code analysis may not mark that method stub as dead. Technical limitations can make static analysis methods inconsistent across languages. A potential ruleset for dead code removal through static analysis would most likely be language, dead code type and codebase specific.

Dynamic code analysis generally comes in the form of unit/component/integration test suites and tools which can lay out the execution flow of your program[13]. However, since dynamic code analysis cannot pinpoint problematic code like its static counterpart, many things are left to the discretion of the developer(s). Boomsma et al's approached dynamic code analysis through the lens of file usage in php web applications[11]. Extensibility and a consistent degree of precision come as a benefit of their approach. However, analyzing at a file level can lead to false positives or false negatives. Even if a file sees a lot of usage, dead code can still be present within the file. Not to mention that files can vary immensely in size and readability. Developers must investigate a flagged file for dead code and identify the culprit code block(s), a task which can vary drastically in timescale. Granularity is important for detecting dead code with dynamic code analysis since dynamic analysis cannot point to a specific line and label it as dead code like static analysis. Developers need to have somewhat of an idea of what to flag for identification when utilizing dynamic code analysis tools. Classifying specific, focused types of dead code as we did throughout the taxonomy section is key to creating dynamic code analysis tools for dead code.

However, if we have a type of dead code in mind to remove as developers, we can specialize a dynamic code analysis tool to remove that type of dead code. Revisiting our method stub example where it represented unnecessary functionality, we could write a dynamic code analysis tool just for those method stubs. A potential dead code flag would be if we find method

stubs that are called in a testing suite but the functionality they were standing in for is no longer present, then the test itself becomes useless.

Static/dynamic code analysis both have their pros and cons in regards to dead code removal. They should be utilized for different types of dead code removal. For example, static code analysis excels at unreachable code removal, but is limited in regards to logically inconsistent dead code. Dynamic code analysis is more flexible but gives less precise indicators for problem areas of code, which in this case means dead code. Both can potentially be effective at removing narrower, quantifiable definitions of dead code instead of targeting dead code holistically.

IV. Conclusion

Dead code remains a pervasive challenge throughout the world of software development. Software development as an industry is booming and the scope of today's software grows alongside it. Identifying dead code with accuracy and efficiency is key for the everyday software that our world runs on. Being able to classify and taxonomize dead code into smaller levels of granularity allows us as developers to have a more objective understanding of dead code. Through finding patterns in research studying real world code bases, we can classify and define these subsections of dead code. Current dynamic/static code analysis tools have limitations in dead code detection but by applying narrower dead code classifications, we may be able to find new ways to use these tools.

Acknowledgement

I would like to thank professor Warren Harrison for his invaluable guidance in writing this paper. He has been a great help in understanding dead code as a subject.

References

[1] J.L. Lions. 1996. Ariane 5 Failure - Full Report. (July 1996). Retrieved October 20, 2021 from http://sunnyday.mit.edu/nasa-class/Ariane5-report.html

[2] Jamie Lynch. 2017. The worst computer bugs in history: The ariane 5 disaster: Bugsnag Blog. (September 2017). https://www.bugsnag.com/blog/bug-day-ariane-5-disaster

[3] Niels Groot Obbink, Ivano Malatova, Gian Luca Scoccia, and Patricia Lago. 2018. An extensible approach for taming the challenges of JavaScript dead code elimination. (April 2018). Retrieved November 19, 2021 from https://ieeexplore-ieee-org.proxy.lib.pdx.edu/document/8330226/authors#authors

[4] Arvind Padmanabhan. 2021. Dead code. (April 2021). Retrieved November 19, 2021 from https://devopedia.org/dead-code#dos-Reis-et-al.-2020

[5] Unreachable Code. Retrieved January 20th, 2022 from

https://www.tutorialspoint.com/software_testing_dictionary/unreachable_code.htm

[6] Method stub. Technopedia. Retrieved January 20th, 2022 from https://www.techopedia.com/definition/3731/method-stub-software-development

[7] Snell, M. and Powers, L., 2015. *Microsoft Visual Studio 2015 Unleashed, Third Edition*. 3rd ed. O'Reilly. Retrieved February 13th, 2022 from https://www.oreilly.com/library/view/microsoft-visual-studio/9780134133164/ch09lev2sec12.html

[8] Simone Romano, Christopher Vendome, Giuseppe Scannielo, and Denys Poshyvanyk. 2018. A Multi-Study Investigation into Dead Code. (June 2018). Retrieved November 19, 2021 from https://ieeexplore-ieee-org.proxy.lib.pdx.edu/document/8370748

[9] Jeff Hecht. March 5, 1994. Technology: Bank error not in your favour - lose Dollar 15 million . *newscientist*. Retrieved January 9, 2022 from https://www.newscientist.com/article/mg14119152-800-technology-bank-error-not-in-your-favour-lose-dollar-15-million/

[10] Felix Neubauer, Karsten Scheibler, Bernd Becker, Ahmed Mahdi, Martin Fränzle, Tino Teige, Tom Bienmüller, and Detlef Fehrer. Accurate Dead Code Detection in Embedded C Code by Arithmetic Constraint Solving. *Ceur-ws*. Retrieved November 29, 2021 from http://ceur-ws.org/Vol-1804/paper-07.pdf

[11] Hidde Boomsma, B.V. Hostnet, and Hans-Gerhard Gross. 2013. Dead code elimination for web systems written in PHP: Lessons learned from an industry case. (January 2013). Retrieved November 19, 2021 from https://ieeexplore-ieee-org.proxy.lib.pdx.edu/document/6405314

[12] Richard Bellairs. February 10, 2020. What is Static Analysis? Static Code Analysis Overview. Retrieved February 13, 2022 from https://www.perforce.com/blog/sca/what-static-analysis

[13] TotalView. July 10, 2020. What is Dynamic Analysis? Retrieved February 13, 2022 from https://totalview.io/blog/what-dynamic-analysis

[14] Jernej Novak, Andrej Krajnc, Rok Zontar. January 2010. Taxonomy of Static Code Analysis Tools. Retrieved February 13, 2022 from https://www.researchgate.net/publication/251940397_Taxonomy_of_static_code_analysis_tools