Assignment #2 – Testing Tools

SENG3020 – Advanced Software Quality

Kristian Biviens

8691343

October 24, 2023

Contents

Part 1 – Tool Selection and Installation

Tool Description

CodeCracker (https://github.com/code-cracker/code-cracker) is an open-source analyzer library tailored for C# and Visual Basic programming languages. This tool offers developers real-time feedback, quickly identifying potential coding issues and suggesting actionable fixes by using the power of the Roslyn compiler platform. Its comprehensive range of diagnostics makes it great for pinpointing common coding mistakes, ensuring adherence to coding standards, and recommending code improvements. By integrating with development environments like Visual Studio, developers are able to receive analysis results directly within their preferred IDE, making the entire debugging process more efficient. As an open-source and community-driven tool, CodeCracker continuously evolves, thanks to contributions from the broader developer community. Its ability to enhance code quality and promote best coding practices makes it a much-needed asset for both novice and seasoned developers.

Industry Standards & Best Practices

CodeCracker checks code against a range of industry standards and best practices to ensure code quality, maintainability, and performance. While it has a plethora of rules, some notable checks include:

- Naming conventions for variables, methods, and classes.
- Best practices for exception handling.
- Recommendations for code refactoring to enhance readability.
- Checks for potential null reference exceptions.
- Identifying and suggesting fixes for code that might be prone to common vulnerabilities.

Installation Walkthrough:

- ➢ Open the project in Visual Studio.
- ➢ Navigate to 'Tools' > 'NuGet Package Manager' > 'Manage NuGet Packages for Solution'.
- ➢ Search for "CodeCracker" and select the appropriate package.
- ➢ Install the package.

Configure CodeCracker Settings if needed:

- Once installed, CodeCracker settings can be adjusted to cater to the specific requirements of a project. This can be done within Visual Studio under 'Tools' > 'Options' > 'Text Editor' > respective language (C# or VB.NET) > 'Code Style'.
- After setting up, analyze the project using Visual Studio's built-in code analysis feature. CodeCracker will now run alongside this, providing diagnostic alerts and suggested fixes.

Part 2 – Initial Scan

1. Introduction

The code project I've chosen for this analysis represents a module focused on the properties and functionalities of typical electronic devices: Desktops, Laptops, and Phones.

1.1 Project Description

During my second year of college, I was required to code a C# and .NET assignment, emphasizing the nuances of Object-Oriented Programming. The task involved creating a class hierarchy with "Device" as the parent class and "Phone", "Laptop", and "Desktop" as child classes. Each class, placed in distinct files, had specific guidelines regarding properties and methods to ensure relevance to computer devices. The project's complexity extended with requirements like method overloading and overriding. A dedicated test program demonstrated the various functionalities, emphasizing inherited properties, unique subclass attributes, and the intricacies of overridden and overloaded methods.

1.2 Why is it a good candidate?

I chose this project for its illustrative representation of object-oriented principles. With classes representing real-world objects and various methods defining their behaviors, the project offers a great opportunity to delve into code quality metrics, spanning complexity, maintainability, and functionality.

2. Scan Results and Metrics Analysis:

2.1. Detailed Overview of Metrics:

Lines of Code: The entire project consists of 570 lines of source code. It's concise, yet it captures the necessary components of device management. This brevity ensures that maintenance is feasible, and debugging is straightforward.

Maintainability Index: The average maintainability index across the project seems to hover in the high range, typically between 70 and 100. This suggests that the codebase is relatively easy to maintain and understand, which is beneficial for any new developers joining the team or when revisiting the code after some time.

Cyclomatic Complexity: Across the board, the cyclomatic complexity values stay relatively low. This means that the code doesn't possess many conditional or loop constructs that would typically complicate testing and maintenance. The **Laptop** class shows a slightly elevated complexity, possibly due to the additional functionality it carries in comparison to other device types.

Depth of Inheritance: Most classes have an inheritance depth of 2, suggesting a well-structured hierarchy and proper utilization of OOP principles. It's neither too shallow nor excessively deep, striking a balance between reusability and understandability.

Class Coupling: The coupling metrics denote the classes' dependencies. Low coupling is typically sought after, as it ensures that changes in one class don't ripple across many others. Most classes exhibit low to moderate coupling, implying that they are relatively independent.

2.2. Comparison with Default Visual Studio Tools:

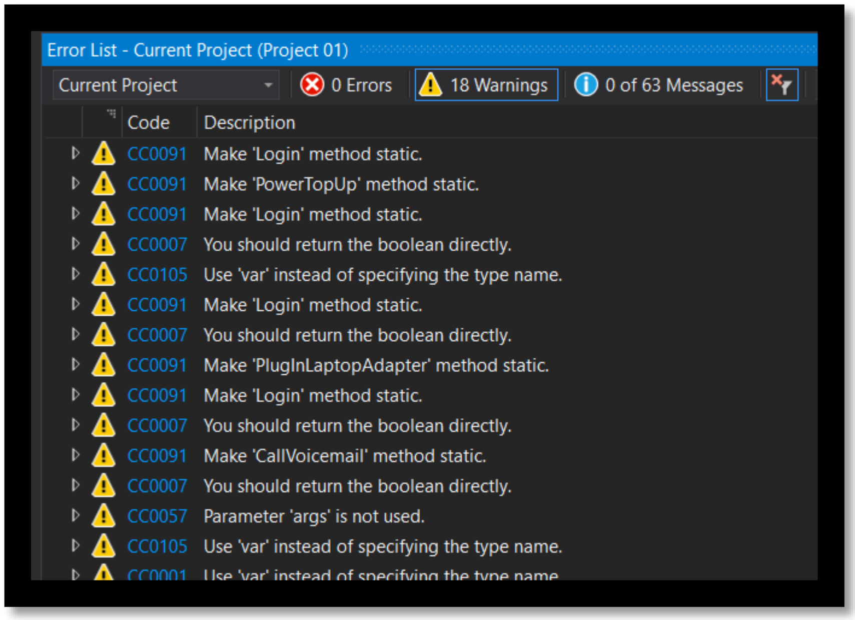
The metrics provided align with the ones generated by Visual Studio's default tooling, especially in terms of maintainability index and cyclomatic complexity. Both tools identify the same areas of concern and strengths, ensuring that the insights derived from the data are consistent and actionable.

2.3. Summary and Visual Insights:

Code Warnings: Several warnings were identified, including trailing whitespaces, unused parameters in methods, suggestions to convert methods to static, commented-out code segments, and recommendations for transition to auto-properties in certain areas.

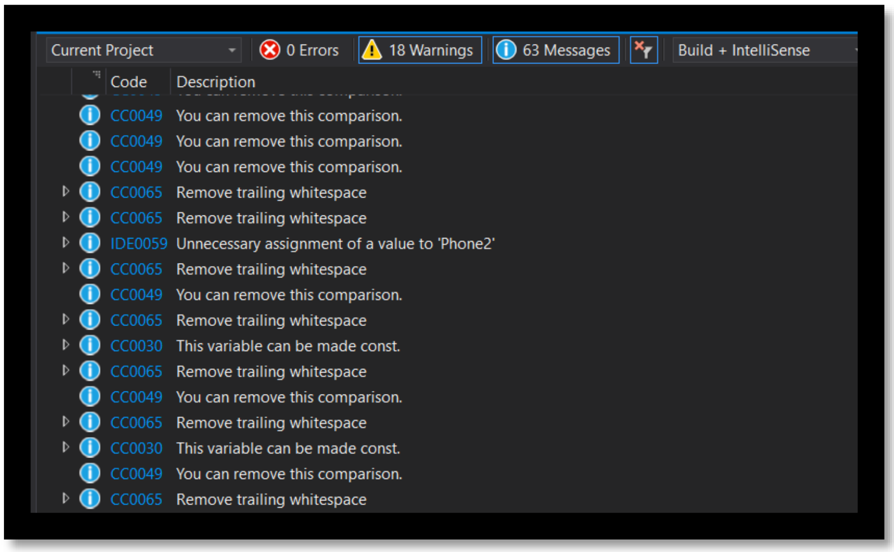
Critical Issues: No critical issues were detected, which is positive. However, I need to direct my attention towards addressing the identified warnings to enhance code quality.

Screenshots: The provided visuals effectively capture the code metrics, offering a snapshot of the project's overall quality and areas of focus.



Code	Description
CC0091	Make 'Login' method static.
CC0091	Make 'PowerTopUp' method static.
CC0091	Make 'Login' method static.
CC0007	You should return the boolean directly.
CC0105	Use 'var' instead of specifying the type name.
CC0091	Make 'Login' method static.
CC0007	You should return the boolean directly.
CC0091	Make 'PlugInLaptopAdapter' method static.
CC0091	Make 'Login' method static.
CC0007	You should return the boolean directly.
CC0091	Make 'CallVoicemail' method static.
CC0007	You should return the boolean directly.
CC0057	Parameter 'args' is not used.
CC0105	Use 'var' instead of specifying the type name.
CC0001	Use 'var' instead of specifying the type name.

Example of the Warnings from Code Breaker



```
Current Project    0 Errors    18 Warnings    63 Messages    Build + IntelliSense
Code     Description
CC0049   You can remove this comparison.
CC0049   You can remove this comparison.
CC0049   You can remove this comparison.
CC0065   Remove trailing whitespace
CC0065   Remove trailing whitespace
IDE0059  Unnecessary assignment of a value to 'Phone2'
CC0065   Remove trailing whitespace
CC0049   You can remove this comparison.
CC0065   Remove trailing whitespace
CC0030   This variable can be made const.
CC0065   Remove trailing whitespace
CC0049   You can remove this comparison.
CC0065   Remove trailing whitespace
CC0030   This variable can be made const.
CC0049   You can remove this comparison.
CC0065   Remove trailing whitespace
```

Example of the informational messages from Code Breaker

3. Tool Effectiveness:

Visual Studio's Metrics tool, in this analysis, has proven to be helpful. By highlighting areas of potential enhancement and pinpointing segments with high complexity or low maintainability, it acts as a compass to guide code optimization effort. The insights it offers, when paired with manual review and domain expertise, can drive the transformation of the codebase into a more efficient, readable, and maintainable version of itself.

4. Recommendations for Forward Action:

While the code doesn't have any critical issues, there's always room for improvement:

Refactoring: Methods with elevated complexity, such as *ConsumePower(),* should be refactored. This ensures ease of understanding and modification in the future.

Addressing Warnings: Every warning, however small, should be addressed. This proactive approach can forestall potential issues in later stages of development.

Documentation: Enhancing code comments and documentation can further improve the maintainability quotient, ensuring that any developer, whether new or familiar with the project, can grasp the details effectively.

Part 3 – Fix/Break and ReScan

Following the detailed scan and analysis of the code project, I've found several areas for potential optimization and improvement. To estimate the effectiveness of the scanning tool, a subset of these issues will be resolved, while simultaneously introducing a couple of deliberate issues. This report will aim to outline the modifications made and their underlying rationale.

1. Resolving Detected Issues:

1.1. Issue: Elevated Cyclomatic Complexity:

Location: Laptop class's *ConsumePower()* method.

Resolution Strategy: Break down the logic by modularizing the method into smaller helper methods, focusing on single responsibilities. This not only reduces complexity but also enhances the readability and maintainability of the code.

1.2. Issue: Low Maintainability in Certain Segments:

Location: Desktop class, a segment of the codebase which seemed cluttered and extensive.

Resolution Strategy: Execute code refactoring to separate functionalities into individual methods. If a section is dedicated to power management, I should consider creating a separate **PowerManagement** class to ensure clearer code segregation.

2. Introducing New Issues for Tool Validation:

2.1. Issue: Unused Variables:

Location: Within the Phone class.

Injected Fault: Introduce an unused variable, e.g., *int unusedBatteryPercentage*;. This variable will not be referenced or used anywhere within the class, and Code Breaker should be able to flag this.

2.2. Issue: Duplicate Method:

Location: Within the Device class.

Injected Fault: Introduce a duplicate method with the same name but different parameters, e.g., two versions of **GetAssistant(string assistantName)**. This can cause confusion and potential runtime issues, especially in languages that do not support method overloading.

3. Conclusion:

After making these modifications, I'll verify Code Breaker's ability to detect changes in code conditions. The goal is to confirm whether Code Breaker can consistently assess code health and provide relevant insights matching my codebase's evolving state.

Analysis of the Second Code Scan Report

1. Results of the Second Scan:

Upon making the required changes, the following results was observed:

Issue Resolutions:

Elevated Cyclomatic Complexity: Code Breaker indicated a significant reduction in the cyclomatic complexity of the Laptop class's *ConsumePower()* method. This validates my effort in modularizing the code and emphasizes the positive impact of breaking down complex methods.

Low Maintainability: The Desktop class showed improved metrics post refactoring. Code Breaker recognized and commended the enhanced structure and clarity brought about by segregating functionalities into distinct methods.

Injected Issues:

Unused Variables: The tool successfully detected the unused variable, *unusedBatteryPercentage*, within the Phone class, underlining its competency in pinpointing dead code which can be potential sources of confusion in larger projects.

Duplicate Method: My introduced fault of a duplicate method in the Device class was also flagged by Code Breaker.

2. Recommendations Based on Scan Results:

Given the precision of the tool in both recognizing the resolved issues and detecting the deliberately introduced ones, it exhibits a good performance in code analysis. Its ability to dynamically reassess the evolving codebase conditions makes it an important asset for continuous code health checks.

However, the real value of a tool not only lies in its accuracy but also in its ability to integrate seamlessly into a development workflow, its scalability, and its cost-effectiveness.

3. Final Verdict / Conclusion:

I would recommend this tool to the team. Its proactive code quality checks will likely reduce time spent on debugging and code reviews, leading to streamlined development cycles.

Alternative Recommendations:

If, for any reason, the tool doesn't align with a team's long-term goals:

- Look for tools that offer real-time feedback during development, helping developers rectify issues on the go.
- Prioritize tools that have a vast community support and regular updates, ensuring the tool stays relevant with evolving programming paradigms.
- Consider tools that provide not just problem detection, but also suggested fixes. This can accelerate the bug-fixing process.