VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

FACULTY OF FUNDAMENTAL SCIENCE

DEPARTMENT OF INFORMATION SYSTEMS

Boris Kulagin

RESEARCH ON REFACTORING METHODS FOR ENSURING ASYNC CODE IN .NET APPLICATIONS

Master Graduation Thesis

Supervisor: Kęstutis Normantas

VILNIUS 2024

Vilnius Gediminas Technical University	ISBN	ISSN
Faculty of Fundamental Sciences	Copies No.	
Department of Information Systems	Date ..........-.....-.....	

Master Degree Studies **Information Systems Software Engineering** study programme Master Graduation Thesis	
Title	**Research on Refactoring Methods for Ensuring Async Code in .NET Applications**
Author	**Boris Kulagin**
Academic supervisor	**Kęstutis Normantas**

	Thesis language: English

Annotation

The thesis addresses the challenge of refactoring code from the synchronous to the asynchronous paradigm. The main aim of this work is to enhance the asynchronization process by detecting and resolving blocking code issues. Through a comprehensive review of scientific literature and targeted research, existing gaps in current methods are identified. To bridge these gaps, this research extends the focus to the asynchronization of the entire call graph rather than just the call stack. Blocking code issues, which hinder the asynchronization process, are thoroughly examined. In conjunction with the proposed method, an automated refactoring tool is developed. To validate the proposed method, a set of test projects is created and analyzed. Moreover, experiments with the application of the tool by developers to asynchronize a couple of source code projects are conducted, and their results demonstrate the effectiveness of the proposed method.

Structure: Introduction, 3 chapters, Conclusions, References, and 2 Annexes. The thesis consists of: 41 p. text without appendixes, 25 pictures, 18 tables, 41 bibliographical entries. Annexes included.

Keywords: Refactoring, asynchronous code, synchronous code, blocking code

Vilniaus Gedimino technikos universitetas	ISBN	ISSN
Fundamentinių mokslų fakultetas	Egz. sk.	
Informacinių sistemų katedra	Data ..........-.....-.....	

Antrosios pakopos studijų **Informacinių sistemų programų inžinerijos** programos magistro baigiamasis darbas

Pavadinimas	**Kodo pertvarkymo metodų tyrimas asinchroniškumui užtikrinti .NET taikomosiose programose**
Autorius	**Boris Kulagin**
Vadovas	**Kęstutis Normantas**

Kalba: anglų

Anotacija

Baigiamajame darbe nagrinėjama kodo keitimo problema iš sinchroninės paradigmos į asinchroninę. Pagrindinis šio darbo tikslas – patobulinti asinchronizavimo metodą, aptinkant ir sprendžiant blokuojančio kodo problemas. Išnagrinėjus mokslinius šaltinius ir atlikus tyrimus, nustatomos spragos. Siekiant įveikti aptiktas spragas, tyrime atsižvelgiama į viso kvietimo grafo, o ne tik kvietimų steko asinchronizavimą. Darbe nagrinėjamos blokuojančio kodo problemos kaip asinchronizavimo proceso priežastis. Kartu su metodu yra sukurtas automatizuotas įrankis, skirtas atlikti kodo pertvarkymą. Metodo validacijai buvo sukurtas ir išanalizuotas bandomųjų projektų rinkinys. Be to, buvo atlikta keletas eksperimentų, kuriuose programuotojai sprendė programinio kodo asinchronizavimo užduotis savarankiškai bei pritaikant įrankį. Eksperimentų rezultatai parodo pasiūlyto metodo efektyvumą.

Struktūra: įvadas, 3 skyriai, išvados, literatūros sąrašas ir 2 priedai. Darbo apimtis – 41 p. teksto be priedų, 25 iliustr., 18 lent., 41 bibliografinis šaltinis. Pridedami priedai.

Prasminiai žodžiai: Kodo pertvarkymas, asinchroninis kodas, sinchroninis kodas, blokuojantis kodas

Table of Contents

List of Images

List of Tables

Abbreviations

IDE – integrated development environment

CE – code editor

PL – programming language

JS – JavaScript

AST – abstract syntax tree

VSC – version control system

TPL – Task Parallel Library

1. Introduction

Asynchronous programming paradigm is the de facto standard for developing modern applications. Almost all popular programming languages provide options to write asynchronous code in efficient manner (Evans & Flanagan, 2018; Parker, 2015; Sarcar, 2020; Van Rossum & The Python development team, 2017). However, the complexity of asynchronous code is higher than that of synchronous code, so there are still a lot of applications written in the synchronous programming paradigm. It's worth mentioning that this issue is especially relevant in applications under migration, i.e. written in older versions of programming language, when no async execution was supported by the language design.

Refactoring, as a process of changing code to more efficient and correct side (M. Fowler & Beck, 2019), is widely used to make code work in asynchronous way (Zhang et al., 2020). However, finding out asynchronous code issues and antipatterns is not a trivial task. Therefore, there are still a lot of issues in the majority of applications.

Finding out issues raised by asynchronous code misuse is a complex and time consuming task, which requires appropriate experience, deep understanding of code structure, behavior, and impact on related dependencies. Moreover, there is a lack of methods and techniques that can help to detect and fix asynchronous code issues and usage antipatterns. Thus, research on such methods and techniques would enable development of corresponding refactoring tools and would make development more efficient.

1.1. Investigation Object

The object of this research is refactoring methods and techniques intended for detecting and fixing asynchronous code issues and usage antipatterns.

1.2. The Aim and Tasks of the Thesis

The main aim of this thesis is to increase the productivity and efficiency of asynchronous paradigm-based development in .NET environment by proposing a refactoring method which would facilitate detection and resolving of asynchronous programming antipatterns.

To achieve the main aim, the following tasks are set.

1. To perform analysis of related literature to identify common asynchronous code issues and find out state-of-the-art methods and techniques for handling them.
2. To propose a solution for detecting and refactoring of asynchronous code issues.
3. To implement the proposed solution in .NET environment.
4. To conduct an experiment to validate the implemented solution.

1.3. Novelty of the Topic

Refactoring has been part of software development from the very beginning, but the constant development of languages and programming methods requires a corresponding development of refactoring techniques as well. Therefore, with each new update of development methods, it is necessary to research new possibilities for refactoring. And even though methods and technologies for developing asynchronous code have been present for quite a long time and various studies have been conducted on the topic, there is still a lack of research on methods for identifying incorrect usage of asynchronous code and ways to fix them.

Current refactoring methods almost do not cover the transformation of the code from the synchronous to the asynchronous paradigm, especially when the transformation is required to fix incorrect use of the asynchronous code constructions. This research focuses on researching the mentioned area and providing a novel method for asynchronizing code.

1.4. Relevance of the Topic

The yearly growth of software development is accompanied by an increasing demand for faster and more qualitative development processes, as well as enhanced standards for code quality. Consequently, developers are required to promptly and accurately detect and fix errors. Hence, the investigation of techniques for identifying and refactoring erroneous utilization of asynchronous code is a critically important task, as demonstrated by the huge number of articles, books, and papers presented on the topic.

1.5. Research Methodology

The analytical part of this work uses a narrative literature review and comparative analysis to examine relevant scientific literature related to the refactoring of asynchronous methods. The analysis focuses on existing tools that are designed to detect and refactor erroneous asynchronous constructions. The analysis results in the proposal of a method for addressing issues, with the aim of enhancing the efficiency of developers. The proposed methodology is currently being implemented, and a controlled experiment is being undertaken to verify its efficiency.

1.6. Scientific Value of the Thesis

This thesis presents a novel and systematic approach to refactor asynchronous code in order to ensure its appropriate usage, thereby enhancing the overall reliability and maintainability of programs developed using asynchronous programming paradigm. The development of an IDE extension to automatically detect and fix those issues further enhances the applicability and significance of the work, making it a valuable asset for both software developers and researchers. By providing a comprehensive solution to the challenges associated with asynchronous code, thereby addressing a common problem in modern software development and making a significant contribution to the field of software engineering.

1.7. Main Results of the Thesis

1. A comprehensive related literature analysis shows that detection of asynchronous code anti-patterns and misuse are almost not studied.
2. A novel method along with its tool support are proposed to address the problem.
3. The results of the experiment confirmed the efficiency of using the method.

1.8. Structure of the Work

The introductory section of this thesis presents the problem at hand, outlines the objectives and tasks of the research, and discusses the significance and originality of the chosen topic. The second section of this thesis undertakes an in-depth analysis of academic literature related to the problem domain and identifies areas for further improvement. The third section presents a method for detecting and refactoring erroneous usage of asynchronous code constructions. The implementation of the proposed method as an extension for the IDE is presented in the fourth section, along with a set of experiments to validate its feasibility and measure its effectiveness.

2. Analysis of Refactoring Methods for Ensuring Asynchronous Code

This section defines the concept of refactoring and explains the idea of asynchronous programming. Existing methods for refactoring synchronous code are listed and compared. Also, tools to automate the refactoring process are studied.

2.1. Introduction to the Domain of the Problem

2.1.1. Refactoring

Refactoring is a process of changing internal structure of code, without changing its external behavior (M. Fowler & Beck, 2019). The aim of refactoring is to improve code quality attributes (Al Dallal & Abdin, 2018). Originally, Fowler and Beck introduced 68 methods and techniques to refactor code. All those methods were divided into 7 groups based on the usage principles. However, according to the recent research, developers disagree with completeness of the list suggested by Fowler, so they extend one with their own refactoring techniques (Almogahed & Omar, 2021).

Code quality attributes are categorized into two kinds: internal and external (Morasca, 2009). There are number of internal attributes like complexity, coupling, size, etc., but the common quality of ones is that that those attributes can be measured, whereas external ones like understandability, readability, extensibility, maintainability, etc. are not measurable (Fenton & Bieman, 2015).

Even though refactoring aims to impact only on code quality attributes, but software metrics like performance, security and even energy consumptions are also affected (Almogahed et al., 2022; Şanlıalp et al., 2022; Traini et al., 2022). Moreover, impact level depends on refactoring technique selected and may result in degradation of the metric (Al Dallal & Abdin, 2018; Traini et al., 2022). It is important to note that refactoring process not only improves code quality, but also, sometimes introduces new bugs (Bavota et al., 2012).

2.1.2. Asynchronous Code

Parallel programming is a technique when multiple code blocks can be run simultaneously without blocking each other (Pacheco & Malensek, 2022). To achieve parallelism, developers need to create more than one thread per application. Having threads allows developers to fully control the behavior of the application, but on the other hand, there are vast number of difficulties related to data synchronization between threads (Lu et al., 2008).

Asynchronous programming is a subset of parallel programing techniques, when a potentially long-running task is run separately from the main application thread, thus not blocking one (Jhala & Majumdar, 2007). In contrast to parallel programming, asynchronous ones do not require to deal with threads, so code is simpler and more maintainable. Using specific code constructions like callbacks or async/await helps to achieve asynchronous execution. On the other hand, this method is not always lead to performance improvements, actually applying asynchronous code may result in longer execution time, however, non-blocked main thread allows user to continue interaction with an application, so user`s performance may increase.

2.2. Literature Review

To better understand the problem domain, the next questions were raised:

- RQ1: What kind of problems are being tried to solve? What are the main obstacles?
- RQ2: What methods and techniques are being used?
- RQ3: What is the level of automation of proposed methods? Is there a tool support?

In the following sections, these questions will be addressed by reviewing the related literature.

2.2.1. Problems (RQ1)

Blocked thread

The first step to make code run in asynchronous way is to extract synchronous code block to a separate function and run it as a Task/AsyncTask/Promise/etc. (Dig, 2015; Lin et al., 2014) or even run the function on a separate thread. This approach is extremely effective in case when main thread is needed to be available for other work, otherwise the main thread is blocked until long-running work is completed and users of the applications must deal with frozen UI or longer responses.

However, running concurrent code may result in race conditions when multiple Tasks or Threads tries to access the same resource (Lu et al., 2008). To overcome this issue, developers need to ensure access policy enforcement. This is typically done with locking mechanism or with usage of thread-safe objects and collections (Dig et al., 2009).

Performance issues

Despite the fact that running asynchronous code is not making response of the program faster for single user scenario, things completely differ when workload is increased greatly (Gokhale et al., 2021). Refactoring code from synchronous to asynchronous allows web-applications to handle more requests per period of time without scaling the application.

Also, receiving data from remote sources is known as a time-consuming task. Splitting one synchronous task to many small ones and running simultaneously may result in significant performance improvements (Sodian et al., 2022).

Callbacks Hell

However, not only issues may lead to necessity of refactoring. Using development techniques with a lot of nested callbacks can cause “Callbacks Hell” (Burchard, 2017; Ogden, 2023) to developers, this increase cyclomatic and cognitive complexities, so code becomes hard to understand, test and maintain. Increased complexity results in performance degradation of developers both in time required to implement new features and introduced bugs amount (Antinyan et al., 2017).

Thus, migrating to code which uses promises and tasks makes it more readable, understandable, and maintainable (Gallaba et al., 2017; Okur, 2015). Converting callbacks to promises and awaiting them in correct places, makes code look more sequential and improves error handling.

Evolution of programming language

One more case that developers need to deal with is the evolution of programming languages. Introduction of new methods, paradigms and abstractions results in obsolescence of other constructions (Okur, 2015).

Asynchrony based on callbacks was deprecated and usage of async/await statement was introduced as a new standard in C# 5 (*Microsoft Asynchronous Programming Patterns*, 2023). Also, some libraries are evolving providing async methods and marking synchronous ones as deprecated or even removing them.

With the ECMAScript6 revision of JavaScript, Promises were introduced. Later, with the ECMAScript2017 revision, async/await keywords were added. Even though the use of callbacks has not been marked deprecated or undesirable, but callbacks-based asynchrony became less popular, with more developer opting for asynchrony with more sequential-looking code (Gokhale et al., 2021).

Misuses and anti-patterns

Nevertheless, introducing clearer programming concepts does not preserve developers from incorrect usage of those techniques. Async/await tends to be easily understandable pattern than callbacks or threads, but there are still a lot of misuses (Turcotte et al., 2022).

Anti-patterns and misuses are challenges to developers as they are hard to be detected without deep knowledge of programming language and execution flow (Orton & Mycroft, 2021). Thus, there are huge number of metrics that can be impacted with incorrect usage of asynchronous code.

2.2.2. Methods and Techniques (RQ2)

Code block extraction

The most straightforward technique to detect a code block to be asynchronized is manually find one (Dig, 2015; Lin et al., 2014). This requires both deep knowledge of a code base, and an enormous amount of attention. Then, the developer needs to extract code block to a separate function and call this function in asynchronous manner. However, manual refactoring results in introducing additional bugs and incorrect usage of tasks/promises. Therefore, automation takes place.

Synchronous system methods replacement with asynchronous ones

The most obvious option to start automatic asynchronization is to replace calls of synchronous system functions with asynchronous ones (Beillahi et al., 2022; Gokhale et al., 2021). The majority of long running functions have asynchronous versions with the same parameters list, so this change can be completed quite smoothly.

Surely, the refactoring process does not end with only one function replacement. Once the initial function is defined, the whole call stack needs to be transformed to an asynchronous version. Unfortunately, not all functions can be automatically converted to asynchronous versions due to limitations of the call stack.

Callbacks flattening

The next step towards more understandable asynchrony is done with refactoring asynchronous callbacks to promises (Gallaba et al., 2017). Finding asynchronous methods is kind of trivial task, as one of method`s parameters is known asynchronous callback. Then the callback is transformed to return a promise and an initial call is flattened. This approach requires complex code transformations; however, initial behavior is preserved as only code struct is changed.

Table 1. Callbacks Hell in JavaScript (Gallaba, 2015).

```
getUser('jackson', function (error, user) {
   try {
      if (error) {
         handleError(error);
      } else {
         getNewTweets(user, function (error, tweets) {
            try {
               if (error) {
                  handleError(error);
               } else {
                  updateTimeline(tweets, function (error) {
                     try {
                        if (error) handleError(error);
                     } catch (e) {
                        globalErrorHandler(e);
                     }
                  });
               }
            } catch (e) {
               globalErrorHandler(e);
            }
         });
      }
   } catch (e) {
      globalErrorHandler(e);
   }
});
```

Table 2. Callbacks Hell flattened (Gallaba, 2015).

```
getUser('jackson')
   .then(getNewTweets, handleError)
   .then(updateTimeline, handleError)
   .catch(globalErrorHandler);
```

Table 1 shows how the Callbacks hell looks like. Even though there is only 3 levels of callbacks, it is hard to understand the execution flow.

Table 2 shows the code after applied transformations. Transformed code is more readable and understandable, and what is more important, adding additional functionality won`t increase complexity.

Upgrade to modern abstractions

Modern async/await constructions provided by programming languages are quite easy to use, however transforming code from callbacks-based asynchrony requires a significant amount of effort. Finding callbacks to convert is the easiest part of the process, then major transformations needed to be done (Okur, 2015; Okur, Erdogan, et al., 2014). Both complete methods rebuilding and the entire call graph transformations may need to be processed to complete the upgrade process. However, as it was mentioned before, due to limitations of the call stack, automated upgrade process is hardly possible.

Anti-patterns and misuses detection

The most challenging part of ensuring correct asynchronous code behavior is to detect and fix anti-patterns and misuses (Lin & Dig, 2015; Okur, 2015; Turcotte et al., 2022). This process involves not only anti-patterns detection, but also code analysis to ensure that the found code negatively impacts the quality attributes of the tested application. Moreover, incorrect usage of asynchronous code structures may lead to deadlocks or race conditions, so such cases also must be analyzed (de Boer et al., 2018; Santhiar & Kanade, 2017).

Each programming language has its own asynchronous code anti-patterns and misuses, thus detection process is almost unique in each case. In order to identify anti-patterns and misuse, the whole call graph should be analyzed both statically and dynamically, otherwise, the major part of issues is unable to be detected automatically.

2.2.3. Automation Level (RQ3)

Manual refactoring is the most common and the most used process of changing code, however most steps of this process can be automated with different tools.

This section covers automation tools invented to automate methods described in section 2.2.3. Not every method is fully automated, but even partial automation helps reduce the time required to perform refactoring operations.

Asynchronizer (Java) was introduced to automate extraction of code blocks to separate functions and make it async (Dig, 2015; Lin et al., 2014). This tool does not provide a wide variety of functionality, but doing even a small thing decreases the number of newly introduced bugs.

Desynchronizer (JS) can perform fully automatic analysis (Gokhale et al., 2021). This tool is limited to a predefined list of functions that can be converted, but after finding the method to transform, Desynchronizer provides the preview of the transformation of the whole call stack. Though, the tool is created to refactor code written with JavaScript, and due to characteristics of the target language, behavior of the application after migration cannot be preserved, so still a lot of manual work is needed.

PromisesLand (JS) performs automatic callback extraction for asynchronous functions and converting them to promises (Gallaba et al., 2017). This tool relies on multiple assumptions and standard code-style convention. However, according to authors, refactoring made under defined conditions is quite accurate and reliable.

Asyncifier (C#) converts legacy callback-based asynchronous methods to ones that use Tasks (Okur, 2015; Okur, Hartveld, et al., 2014). The tool does not perform analysis of the codebase, so methods to covert must be selected by the developer.

AsyncFixer (C#) finds incorrect usage of async/await and fixes them (Okur, 2015; Okur, Hartveld, et al., 2014). The tool covers 5 different misuses but suggests only local fixes without analyzing the whole call-stack or even analyzing related code.

Taski-fier (C#) converts old-fashioned Thread and ThreadPool constructions to modern Task abstraction (Okur, 2015; Okur, Erdogan, et al., 2014).

DrAsync (JS) performs static and dynamic analysis to find 8 misuses of async/await construction and promises (Turcotte et al., 2022). This tool provides a preview of available changes and possible performance improvements.

Dead-Wait (C#) performs static analysis to find 2 blocking code issues while awaiting for Task completion (Santhiar & Kanade, 2017). This tool provides information about issues detected only.

Table 3. Automation tools comparison.

Author (Year)	Name	PL	Problem	Call graph analysis	Automatic issue detection	Refactoring automation
Dig, Lin (2014)	Asynchronizer	Java	Code block extraction and asynchronization	No	No	Only local change
Gokhale (2021)	Desynchronizer	JS	Convert synchronous functions to asynchronous	Call stack only	Predefined list of functions	Preview
Gallaba (2017)	PromisesLand	JS	Convert callback to Promise	No	Yes, with limitations	Only local change
Hartveld, Okur (2014)	Asyncifier	C#	Convert callbacks to Tasks	No	No	Only local change
Hartveld, Okur (2014)	AsyncFixer	C#	Async misuses fix	No	Limited to 5 misuses	Only local change
Hartveld, Okur (2014)	Taski-fier	C#	Threads to Tasks and Parallel	No	Yes	Only local change
Santhiar, Kanade (2017)	Dead-Wait	C#	Blocking code while awaiting for Task completion	Call stack only	Limited to 2 cases	No
Turcotte, Shah (2022)	DrAsync	JS	Async misuses fix	No	Yes	Preview

2.2.4. Tools Support

Beside the tools that are described in previous section, there are plenty of ones provided by commercial companies, but not based on scientific research. Mostly those tools are offered as a part of IDEs or as extensions to them.

Visual Studio – is the most known IDE in .Net development. Invented and maintained by Microsoft, this IDE supports majority of refactoring methods described by Fowler (M. Fowler & Beck, 2019). On the other hand, this IDE does not support anti-patterns detection, even the most known as “God`s class” or “Spaghetti code”. Moreover, synchronous code refactoring to asynchronous version is also not supported.

JetBrains ReSharper – extension to Visual Studio, that aims to extend refactoring and code editing features of Visual Studio. It was released in 2004, when Visual Studio lacked refactoring support. It is still very popular but has the same limitations on asynchrony refactoring as Visual Studio.

JetBrains Rider – the next widely used IDE in .Net development. Invented as a competitor to Visual Studio. Having all features of ReSharper, it still has all limitations as Visual Studio has.

Visual Studio Code – out of the box is a little bit more than just code editor, but have tons of extensions, and as a result supports probably any programming language. It is impossible to check all extensions for refactoring support, but the most popular ones cover the same functionality as tools described before.

Eclipse IDE – was originally designed to support Java language, however, over the years it was expanded to support a variety of programming languages. Eclipse provides refactoring support, however, out of the box there is no support for asynchronization. Nevertheless, there is the highly maintainable plugins marketplace, where extensions can be found, which helps developers to simplify asynchronization process.

Atom – lightweight code editor that supports great number of programming languages. There is no built-in support for refactoring in the way traditional IDEs have. However, due to being highly extensible, there are a number of packages that add support for refactoring. Also, there are few packages that help with asynchronization process.

WebStorm – IDE created to support full-stack web development, so all main programming languages and technologies are supported. WebStorm has strong refactoring support, and what is important, that few refactoring options to asynchronization are provided.

Table 4. IDEs` comparison.

Name	Type	Supported languages	Refactoring support	Asynchronization support	Async misuses detection
Visual Studio	IDE	C#, F#, C++, VB.Net	Full	No	No
JetBrains ReSharper	Extension	C#	Full	No	No
JetBrains Rider	IDE	C#	Full	No	No
Visual Studio Code	CE	Any	With extensions	With extensions	No
Eclipse IDE	IDE	Any	Full	No	No
Atom	CE	Any	With extensions	With extensions	No
WebStorm	IDE	Web	Full	Convert callbacks to async/await	No

Table 4 summarizes information about the described tools. It is clearly visible that the refactoring responsible for asynchronization has almost no support provided. Moreover, there is no async misuse detection, so many issues might go unnoticed until unexpected behavior of applications is obtained after release.

2.3. Main Results of the Second Section

Through a comprehensive literature review, it was defined that there are 5 main problems that developers attempt to address through refactoring. Each solution for these problems is a complex task that entails significant code transformations. Despite their different origins, all solutions ultimately lead to the entire call stack being refactored for asynchrony. It is noteworthy that during the review, a common issue was observed across all studied methods: none of them consider the analysis and modification of the entire call graph. Analyzing the entire graph is a crucial step, as some other call branches may contain code that is incompatible with asynchrony. This highlights the importance of considering the entire call graph when refactoring code for asynchrony to ensure compatibility and maintain the integrity of the codebase.

3. Proposed Method

This section provides a description of proposed method on asynchronous code refactoring. There are described flow of the method and steps to automation.

3.1. Asynchronous Code Refactoring Method

This section describes the idea and main steps of the method to refactor asynchronous code. The one consists of 4 main steps:

- Analyze code and find issues
- Fix issues (convert blocking calls to async one)
- Asynchronize all call stacks for each fixed issue
- On each method converted to asynchronous one, run whole process again

The actual schema of the process is provided on Figure 1.

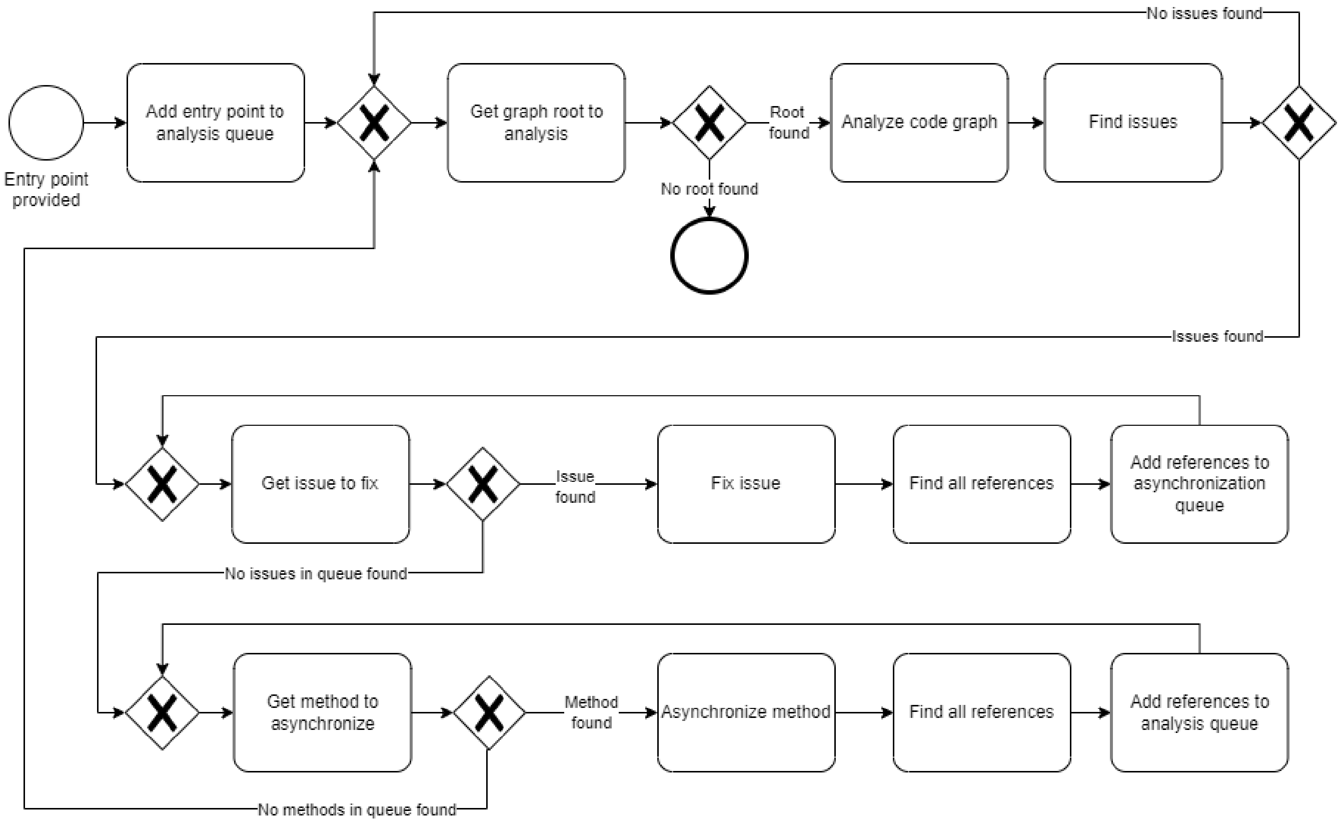


Figure 1. Asynchronous code refactoring process.

3.1.1. Code Analysis and Issue Detection

Code analysis begins with getting an entry point to start with. The most convenient way is to analyze the entire source code of a solution, however, in case of huge code base, this approach could be ineffective and time consuming, which could lead to time loss for developer instead of work performance improvements. Thus, entry point (method or class) should be provided by programmer based on problems context.

The next step of the analysis is to build code representation. The most suitable type of representation is an AST. One has complete information about used methods, expressions, etc. so allowing to analyze calls and dependencies in an efficient way.

Finally, the issue detection step is performed. To detect the issues, the AST is completely examined, what is done by usage of Visitor pattern. Using this technique, each AST node is processed, all expressions are analyzed and compared with a predefined list of error patterns. As a result, a set of possible issues is created.

The AST generation process, as well as the list of error patterns, are specific to the programming language and technology, so ones will be described in the section with implementation of the algorithm.

3.1.2. Issue Fixes

As it is stated in Section 2.2.1 there are a lot of different types of issues to be detected. So, each one has its own fixing logic. Moreover, the issue fixing process is highly dependent on problem context and technologies stack selected. Therefore, the process will be explained in detail in the method implementation section.

3.1.3. Call Stack Asynchronization

Converting synchronous call stack to asynchronous one plays a crucial role in the proposed solution. However, the whole async process comes down to the following steps:

- Change method signature – update return type by wrapping original return type to Promise/Task/etc. and update method name (ex. Add “Async” postfix) in case the last one meets code style of the programming language
- Find all expressions where current method is called and rewrite call to asynchronous manner
- For each method containing the expression from the previous step, follow the steps starting from “change method signature”

Following the presented algorithm, the entire call stack can be converted into an asynchronous version, and the execution of the algorithm will be completed at the moment when no callers are found.

3.1.4. Process Repeat

Every synchronous method converted to an asynchronous one means that previously correct synchronous expressions might become invalid ones. Thus, every converted method should be added to the processing queue and should be processed as soon as previous ones are examined and fixed. This implies that the entire processing algorithm should operate iteratively and complete processing once the queue has been processed entirely.

It is important to note that in case of highly coupled code base, there is high possibility of call stacks being intersected, so the processing algorithm should take care of already processed methods and avoid duplicate processing to provide better performance and faster response.

As soon as there are no nodes left to process, the algorithm ends its work, and it can be stated that the entire call graph is asynchronized.

3.2. Method Automation

This section describes how the method provided above should be automated. Despite the fact that the method itself is complete, automation requires few additional steps. The whole automated approach provided on .

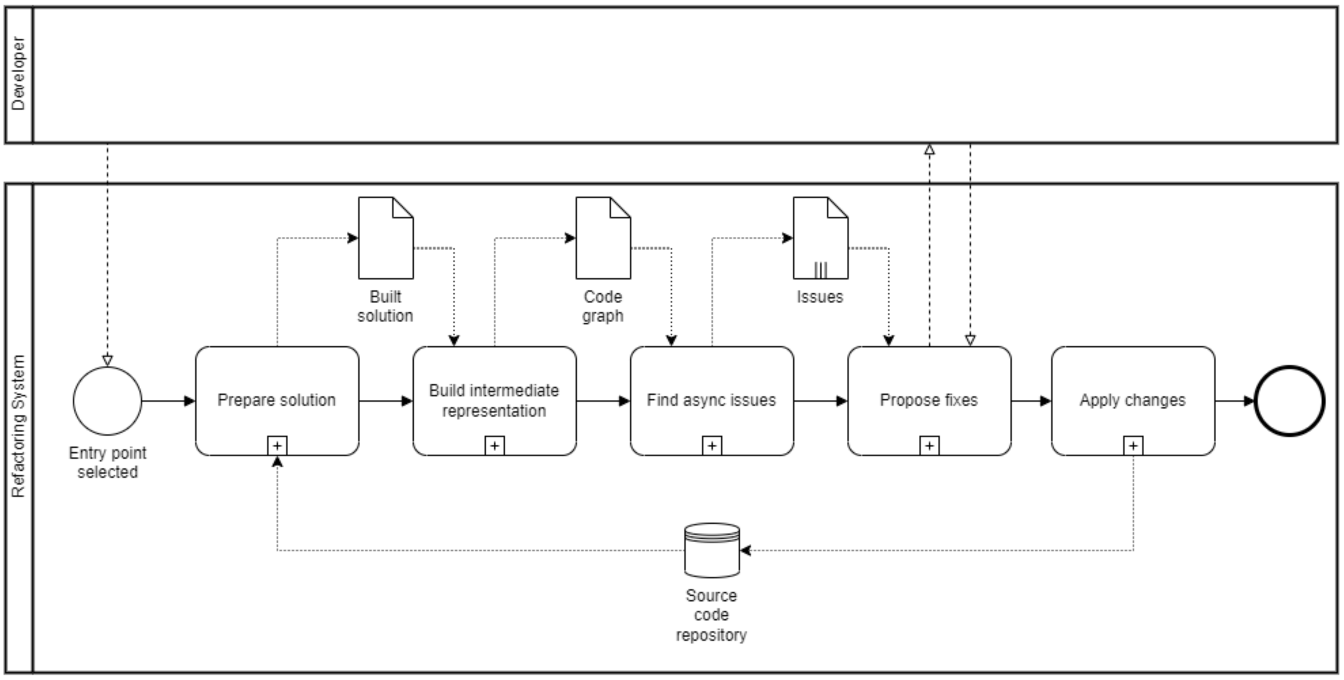


Figure 2. Automated tool working process.

As it is seen from the provided schema, automated method consists of 5 steps:

- Prepare solution
- Build intermediate representation

- Find async issues
- Propose issue fixes to developer
- Apply changes

Each step in detail is described in the next sections.

Even though the provided algorithm is universal and does not depend on the programming language, for more detailed and accurate description of its steps and logic, the .Net (C#) stack is used.

3.2.1. Code Parsing Library

The first step to process with automation is to choose which parsing/analysis/editing library to use. In moder .Net world there are two main technologies to work with source code: Antlr(Parr, 2013) and Roslyn(Mukherjee, 2016).

Both of those technologies provide a wide range of supported functions, however, considering the requirements and goals of this work, Roslyn has a number of fundamental advantages:

- It provides a complete and accurate representation of C# code, including syntax trees, syntax tables, and semantic information;
- It is designed to support compiler-related tasks, such as code analysis, refactoring, and code generation;
- It allows you to perform deep semantic analysis of code, including type checking, and symbol resolution;
- It is the official platform provided by Microsoft, having deep integration with all technologies provided by one, and is kept up with latest language version.

3.2.2. Solution Preparation

Solution preparation is processed in the next order:

- Platform dependencies are loaded: The required platform dependencies are loaded in order to provide basic functionality for the solution. These dependencies are crucial libraries and tools that the solution needs.
- External dependencies (NuGet packages) are installed (restored): The solution might additionally require external dependencies in the form of NuGet packages in addition to the platform dependencies. These external dependencies add pre-built libraries or utilities, which improve the functionality of the solution.
- Solution is built: Once the dependencies are loaded, the build process can be started. The process of building a solution entails linking the required libraries, compiling the source code, and creating build artifacts. Also, the built process ensures consistency of the source code and finds

any compile-time errors. The built solution is not a prerequisite for the generation of AST, however it is necessary for its more accurate construction and the potential for a thorough analysis.

- AST is generated: After the solution is built, an AST is generated. AST shows the relationships between all source code elements and their hierarchical structure in a tree-like representation. This representation allows to perform deep analysis and make various modifications to the source code.

All the steps described above are the starting point for processing the solution. Due to the tight integration of Roslyn and the .Net environment, the developer only needs to write a small amount of code to complete these steps. The required one is shown on Table 5.

Table 5. Solution preparation code using Roslyn.

```
using Microsoft.Build.Locator;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis. MSBuild;

class Program
{
        static async Task Main()
        {
                // Locate and register the default instance of MSBuild installed on this machine.
                MSBuildLocator. RegisterDefaults();
                // Path to the solution file
                string solutionPath = @"C:\Path\to\YourSolution.sln";
                // Load the solution
                MSBuildWorkspace? workspace = MSBuildWorkspace.Create();
                Solution? solution = await workspace.OpenSolutionAsync(solutionPath);
                // Iterate over the projects in the solution
                foreach (Project project in solution. Projects)
                {
                // Iterate over the documents in the project
                        foreach (var document in project. Documents)
                        {
                                // Get the syntax root of the document
                                SyntaxNode? root = await document.GetSyntaxRootAsync();
                                if(root is null)
                                {
                                        continue;
                                }
                                // Generate the Abstract Syntax Tree (AST)
                                CompilationUnitSyntax? compilationUnit (CompilationUnitSyntax)root;
                                // You can now work with the AST and perform various analyses or modifications
                        }
                }
        }
}
```

3.2.3. Intermediate Representation

Considering the fact that the development environment has ready-made models for representing the source code, however converting one to models and structures designed for the current case will greatly speed up both development of a tool and its` execution. Therefore, the developed method uses the structures provided on the class diagram (Figure 3).

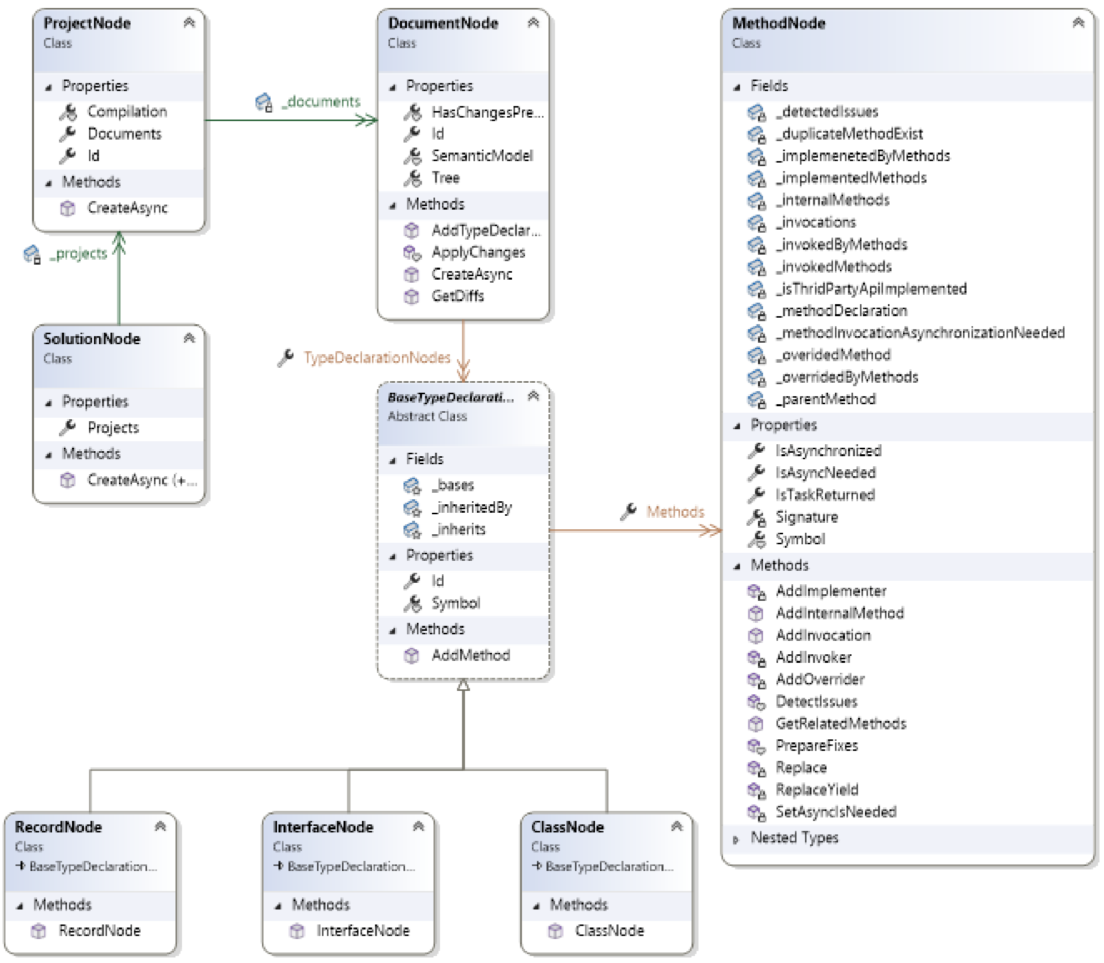


Figure 3. Class diagram of proposed solution.

According to Figure 3, *MethodNode* class takes the core role in the entire process, being responsible both for representing and processing changes, while the remaining classes serve as support functions that are required to project the solution.

3.2.4. Asynchronous Issue Detection

As it is described in Section 2.2.2 (Problems (RQ1)) there are bunch of different issues that developers need to address while dealing with asynchronous code issues. However, all of them requires completely different approach for detection. Thus, the primary objective of this work is to identify and fix blocking code that could lead to deadlocks.

Blocking code detection

According to (D. Fowler, 2023) there are plenty of incorrect asynchronous code usage patterns that lead to blocking code. Main ones, that this work aims to detect and fix are shown on the next subsections.

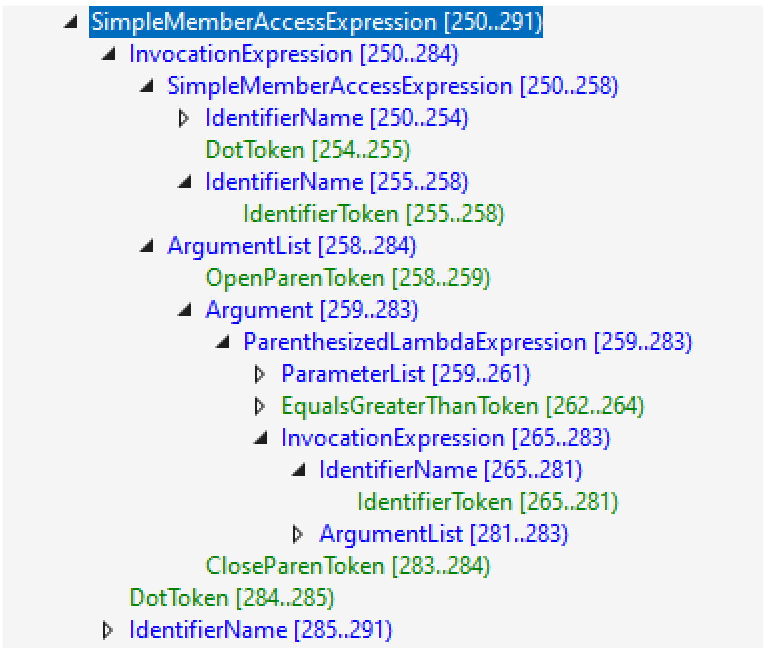
Pattern 1

Table 6. Blocking code example. Pattern 1.

```
public string DoOperationBlocking()
{
        return Task.Run(() => DoAsyncOperation()).Result;
}
```

The code shown on Table 6 blocks entire thread waiting for DoAsyncOperation execution is completed. Despite the fact that the call is wrapped under Task.Run expression, it is still not executed in asynchronous way, so accessing Result property leads to waiting for execution is ended.

In order to detect described issue during AST analysis, the expression shown on Figure 4 should be found.



```
◢ SimpleMemberAccessExpression [250..291)
    ◢ InvocationExpression [250..284)
        ◢ SimpleMemberAccessExpression [250..258)
            ▷ IdentifierName [250..254)
              DotToken [254..255)
            ◢ IdentifierName [255..258)
                  IdentifierToken [255..258)
        ◢ ArgumentList [258..284)
              OpenParenToken [258..259)
            ◢ Argument [259..283)
                ◢ ParenthesizedLambdaExpression [259..283)
                    ▷ ParameterList [259..261)
                    ▷ EqualsGreaterThanToken [262..264)
                    ◢ InvocationExpression [265..283)
                        ◢ IdentifierName [265..281)
                              IdentifierToken [265..281)
                        ▷ ArgumentList [281..283)
              CloseParenToken [283..284)
      DotToken [284..285)
    ▷ IdentifierName [285..291)
```

Figure 4. AST example for Pattern 1.

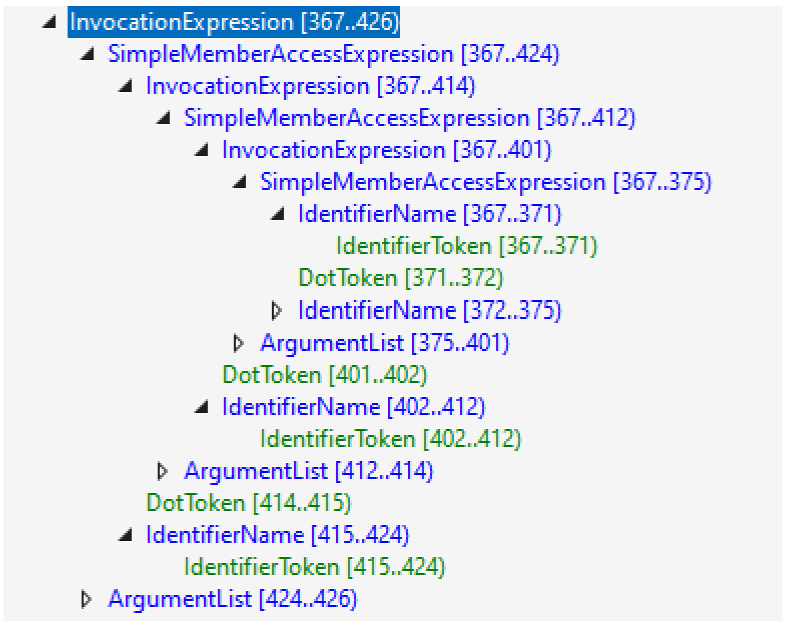
Pattern 2

Table 7. Blocking code example. Pattern 2.

```
public string DoOperationBlocking2()
{
    return Task.Run(() => DoAsyncOperation()).GetAwaiter().GetResult();
}
```

The code shown on Table 7 blocks entire thread waiting for DoAsyncOperation execution is completed. Despite the fact that the call is wrapped under Task.Run expression and then GetResult is called on TaskAwaiter, it is still not executed in asynchronous way, as getting result by calling method GetResult still leads to waiting for execution is ended.

To detect described issue during AST analysis, the expression shown on Figure 5 should be found.



```
◢ InvocationExpression [367..426)
  ◢ SimpleMemberAccessExpression [367..424)
    ◢ InvocationExpression [367..414)
      ◢ SimpleMemberAccessExpression [367..412)
        ◢ InvocationExpression [367..401)
          ◢ SimpleMemberAccessExpression [367..375)
            ◢ IdentifierName [367..371)
                IdentifierToken [367..371)
              DotToken [371..372)
            ▷ IdentifierName [372..375)
          ▷ ArgumentList [375..401)
          DotToken [401..402)
        ◢ IdentifierName [402..412)
            IdentifierToken [402..412)
      ▷ ArgumentList [412..414)
      DotToken [414..415)
    ◢ IdentifierName [415..424)
        IdentifierToken [415..424)
  ▷ ArgumentList [424..426)
```

Figure 5. AST example for Pattern 2.

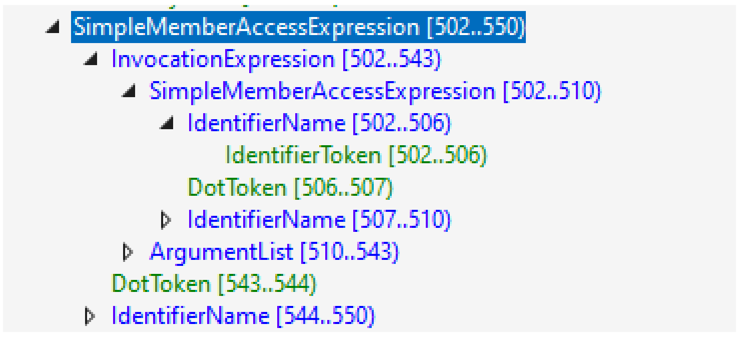
Pattern 3

Table 8. Blocking code example. Pattern 3.

```
public string DoOperationBlocking3()
{
   return Task.Run(() => DoAsyncOperation().Result).Result;
}
```

The code shown on Table 8 blocks entire thread by accessing Result of the task created by Task.Run. Just because external task is blocked, there is no direct impact of blocking operation made by call to Result of DoAsyncOperation.

In order to detect described issue during AST analysis, the expression shown on Figure 6 should be found.



```
◢ SimpleMemberAccessExpression [502..550)
  ◢ InvocationExpression [502..543)
    ◢ SimpleMemberAccessExpression [502..510)
      ◢ IdentifierName [502..506)
          IdentifierToken [502..506)
        DotToken [506..507)
      ▷ IdentifierName [507..510)
    ▷ ArgumentList [510..543)
    DotToken [543..544)
  ▷ IdentifierName [544..550)
```

Figure 6. AST example for Pattern 3.

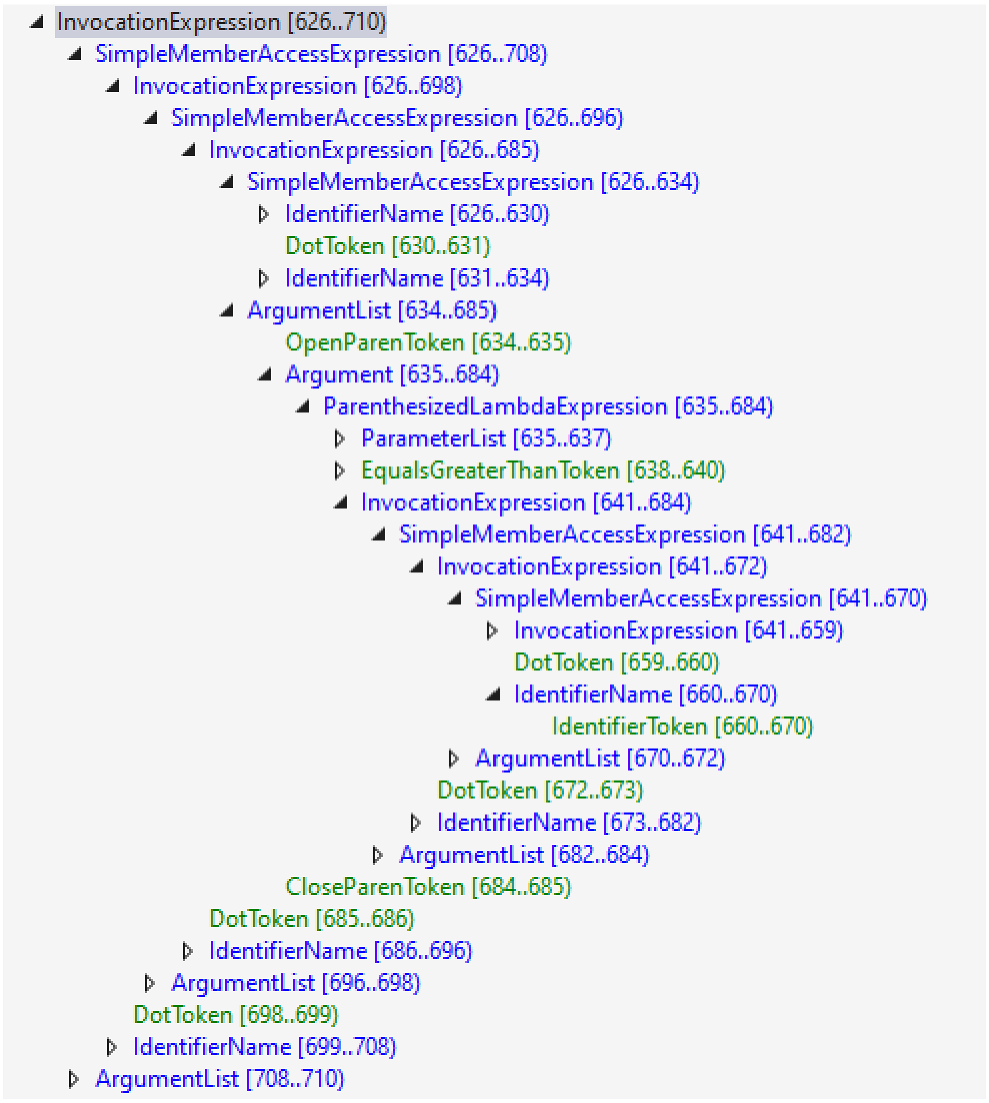
Pattern 4

Table 9. Blocking code example. Pattern 4.

```
public string DoOperationBlocking4()
{
   return Task.Run(() =>
                    DoAsyncOperation().GetAwaiter().GetResult()).GetAwaiter().GetResult();
}
```

The code shown on Table 9 blocks entire thread in the same way as one on Table 8 does, but the only difference is accessing Result by calling GetResult.

To detect described issue during AST analysis, the expression shown on Figure 7 should be found.



```
◢ InvocationExpression [626..710)
   ◢ SimpleMemberAccessExpression [626..708)
      ◢ InvocationExpression [626..698)
         ◢ SimpleMemberAccessExpression [626..696)
            ◢ InvocationExpression [626..685)
               ◢ SimpleMemberAccessExpression [626..634)
                  ▷ IdentifierName [626..630)
                    DotToken [630..631)
                  ▷ IdentifierName [631..634)
               ◢ ArgumentList [634..685)
                    OpenParenToken [634..635)
                  ◢ Argument [635..684)
                     ◢ ParenthesizedLambdaExpression [635..684)
                        ▷ ParameterList [635..637)
                        ▷ EqualsGreaterThanToken [638..640)
                        ◢ InvocationExpression [641..684)
                           ◢ SimpleMemberAccessExpression [641..682)
                              ◢ InvocationExpression [641..672)
                                 ◢ SimpleMemberAccessExpression [641..670)
                                    ▷ InvocationExpression [641..659)
                                      DotToken [659..660)
                                    ◢ IdentifierName [660..670)
                                         IdentifierToken [660..670)
                                 ▷ ArgumentList [670..672)
                                DotToken [672..673)
                              ▷ IdentifierName [673..682)
                           ▷ ArgumentList [682..684)
                    CloseParenToken [684..685)
              DotToken [685..686)
            ▷ IdentifierName [686..696)
         ▷ ArgumentList [696..698)
        DotToken [698..699)
      ▷ IdentifierName [699..708)
   ▷ ArgumentList [708..710)
```

Figure 7. AST example for Pattern 4.

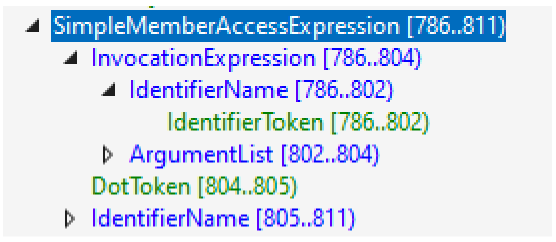
Pattern 5

Table 10. Blocking code example. Pattern 5.

```
public string DoOperationBlocking5()
{
   return DoAsyncOperation().Result;
}
```

The code shown on Table 10 is the most common and obvious way to block entire thread. There is no additional wraps for async function and execution result is accessed directly, so until result is calculated the entire thread is blocked.

In order to detect described issue during AST analysis, the expression shown on Figure 8 should be found.



```
◢ SimpleMemberAccessExpression [786..811)
   ◢ InvocationExpression [786..804)
      ◢ IdentifierName [786..802)
           IdentifierToken [786..802)
      ▷ ArgumentList [802..804)
     DotToken [804..805)
   ▷ IdentifierName [805..811)
```

Figure 8. AST example for Pattern 5.

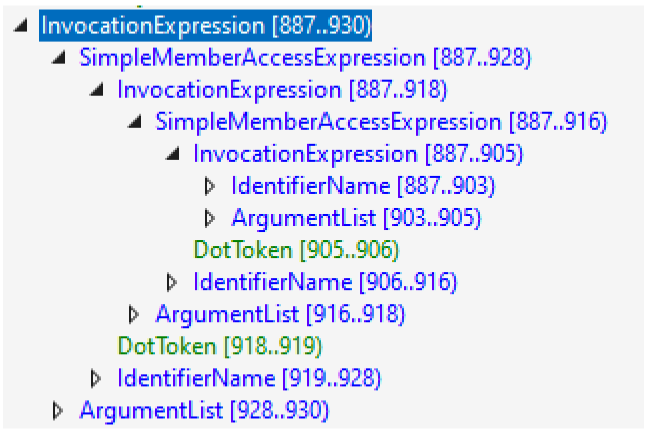
Pattern 6

Table 11. Blocking code example. Pattern 6.

```
public string DoOperationBlocking6()
{
   return DoAsyncOperation().GetAwaiter().GetResult();
}
```

The code shown on Table 11 blocks entire thread in the same way as one on Table 10 does, but the only difference is accessing Result by calling GetResult.

To detect described issue during AST analysis, the expression shown on Figure 9 should be found.



```
◢ InvocationExpression [887..930)
   ◢ SimpleMemberAccessExpression [887..928)
      ◢ InvocationExpression [887..918)
         ◢ SimpleMemberAccessExpression [887..916)
            ◢ InvocationExpression [887..905)
               ▷ IdentifierName [887..903)
               ▷ ArgumentList [903..905)
              DotToken [905..906)
            ▷ IdentifierName [906..916)
         ▷ ArgumentList [916..918)
        DotToken [918..919)
      ▷ IdentifierName [919..928)
   ▷ ArgumentList [928..930)
```

Figure 9. AST example for Pattern 6.

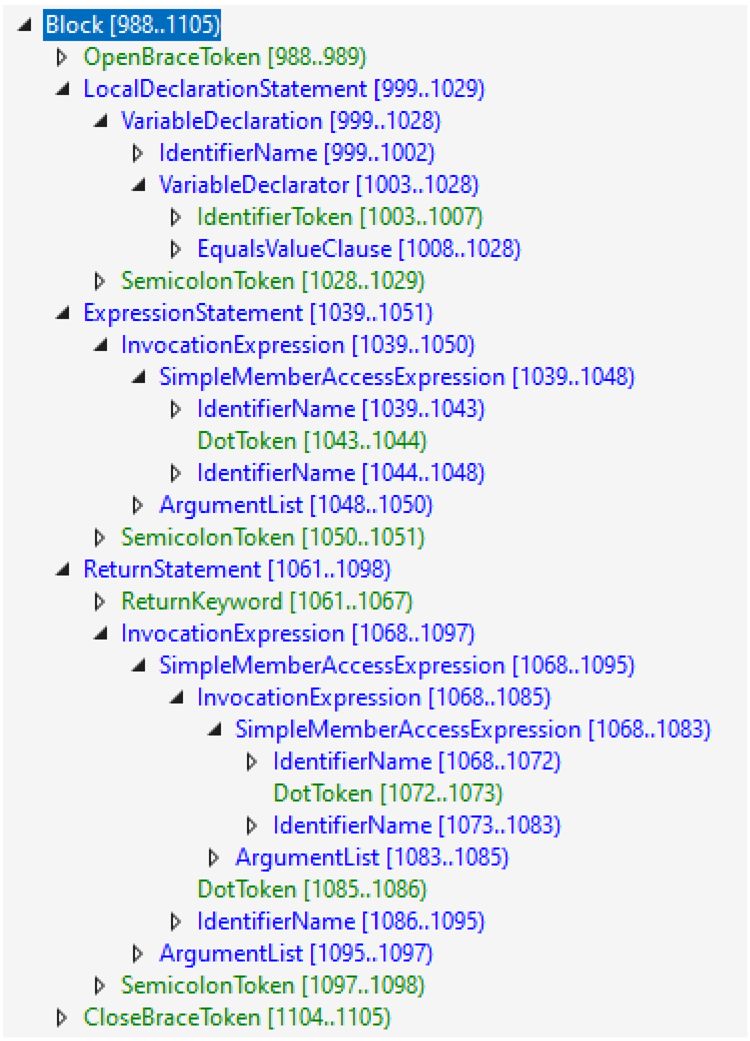
Pattern 7

Table 12. Blocking code example. Pattern 7.

```
public string DoOperationBlocking7()
{
    var task = DoAsyncOperation();
    task.Wait();
    return task.GetAwaiter().GetResult();
}
```

The code shown on Table 12 blocks the entire thread as the all examples above do. However, instead of accessing Result of the execution directly on the first place, the Wait() function is called. One is still blocking as it is executed in synchronous way.

In order to find such issue during AST analysis, the expression shown on Figure 10 should be detected.



```
◢ Block [988..1105)
   ▷ OpenBraceToken [988..989)
   ◢ LocalDeclarationStatement [999..1029)
      ◢ VariableDeclaration [999..1028)
         ▷ IdentifierName [999..1002)
         ◢ VariableDeclarator [1003..1028)
            ▷ IdentifierToken [1003..1007)
            ▷ EqualsValueClause [1008..1028)
      ▷ SemicolonToken [1028..1029)
   ◢ ExpressionStatement [1039..1051)
      ◢ InvocationExpression [1039..1050)
         ◢ SimpleMemberAccessExpression [1039..1048)
            ▷ IdentifierName [1039..1043)
              DotToken [1043..1044)
            ▷ IdentifierName [1044..1048)
         ▷ ArgumentList [1048..1050)
      ▷ SemicolonToken [1050..1051)
   ◢ ReturnStatement [1061..1098)
      ▷ ReturnKeyword [1061..1067)
      ◢ InvocationExpression [1068..1097)
         ◢ SimpleMemberAccessExpression [1068..1095)
            ◢ InvocationExpression [1068..1085)
               ◢ SimpleMemberAccessExpression [1068..1083)
                  ▷ IdentifierName [1068..1072)
                    DotToken [1072..1073)
                  ▷ IdentifierName [1073..1083)
               ▷ ArgumentList [1083..1085)
              DotToken [1085..1086)
            ▷ IdentifierName [1086..1095)
         ▷ ArgumentList [1095..1097)
      ▷ SemicolonToken [1097..1098)
   ▷ CloseBraceToken [1104..1105)
```

Figure 10. AST example for Pattern 7.

Deadlocks detection

Blocking constructions described above could lead to deadlocks when are used inside an asynchronous context (de Boer et al., 2018; Santhiar & Kanade, 2017), so the aim is to detect such cases. In order to achieve that the construction shown on Table 13 should be detected.

Table 13. Example of blocking code inside asynchronous context.

```
public async Task DoAsyncOperation()
{
   DoAsyncOperation2().Wait()
}
```

However, the construction shown on Table 13 is the most straightforward case, while in more general case there can be number of synchronous function calls between asynchronous function and blocking code. Thus, in this case the construction shown on Table 14 should be found.

Table 14. Example of blocking code inside asynchronous context (general case).

```
public async Task DoAsyncOperation ()
{
   DoSyncOperationl();
}
public void DoSyncOperationl ()
{
   DoSyncOperation2();
}
public void DoSyncOperation2 ()
{
   // ..
}
public void DoSyncOperationN_1()
{
   DoSyncOperationN();
}
public void DoSyncOperationN()
{
   DoAsyncOperation2().Wait(); //blocking code
}
```

Crucial part about fixing such kind of issues is that call stack asynchronization may result in converting anti-patterns described in previous section to the deadlock possible code, so all called methods should be checked again at each asynchronization level of the call stack.

Limitations of automatic asynchronization

Even though blocking code described above is considered misuse in any part of an application, there are certain cases where automatic asynchronization is almost impossible due to the need to make architectural decisions to change the structure of code. Those cases are listed below:

- Constructors - were intentionally designed to be lightweight and free of time-consuming calculations, so they could not operate asynchronously. The asynchronous method should be moved to the factory method or a separate initialization method so the blocking code can be eliminated here.

- Properties – should be used to access or set up value of the backing field with little or no additional calculations, however, it is possible to call asynchronous method in blocking manner here. In order to fix this problem, the property should be replaced with corresponding methods, or calculated before first access.
- Methods that implement/override ones defined in external libraries – could be fixed by using different APIs or by completely redesigning usage strategy.

On the other hand, there is a number of constructions that can be easily enough fixed without requiring redesigning the structure of the code. Those constructions are listed below and may be fixed by wrapping return type to Task and in some cases adding *async* modifier.

- Functions and methods
- Local functions
- Lambda functions
- Delegates

3.2.5. Fix Proposal

As soon as analysis of source code is ended, all fixes are found, and refactoring is planned, there is need to user interaction. Such complex changes as entire call graph asynchronization require validation in order to reduce possibility of newly introduced issues count.

Human validation is a crucial step as there is a huge amount of use cases, which could be skipped or processed in incorrect way by an automation tool. Such cases as class constructors require different logic of asynchronization, and do not fall under the definition of method asynchronization, as this require usage of Builder or Factory design patterns. Also, methods used as commands could be asynchronous, but require void return type, but it is tough to detect where method is a command, so validation is also required.

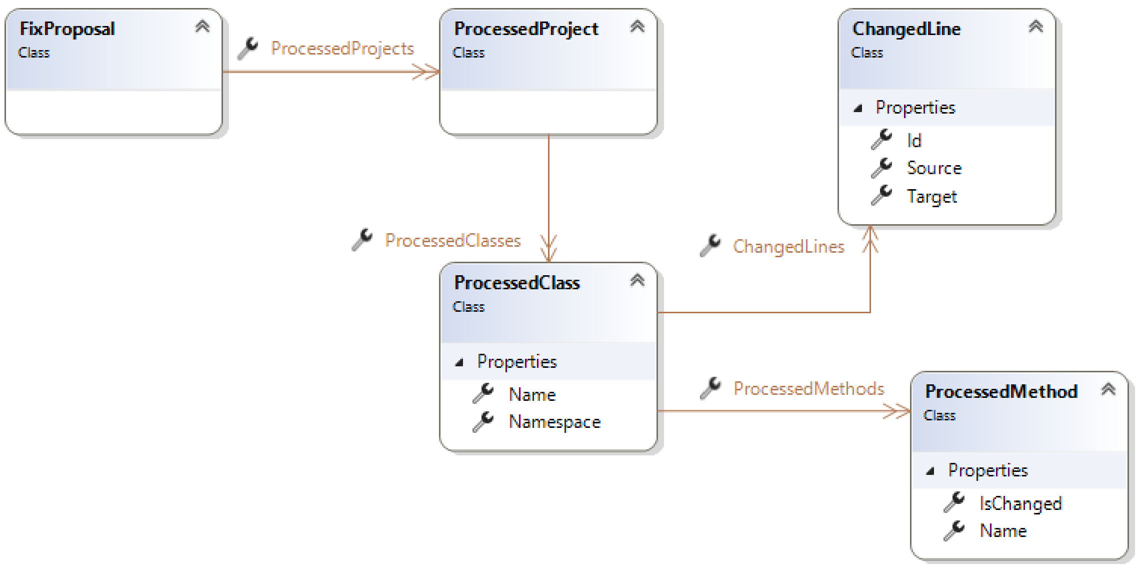


Figure 11. Fix proposal class diagram.

To propose a refactoring solution for reviewing, the structure provided on Figure 11 is used. This structure is intended to be used both to show affected methods on code graph, and show line by line comparison, so developer could efficiently examine all expected changes and approve or discard ones.

3.2.6. Changes Application

In order to complete refactoring process 3 steps are required:

- Creating a new branch on VCS (ex. GIT checkout) – making changes on a separate branch helps to reduce undesired conflicts and allows safe testing, also working on main/master branch is considered to be a bad practice (Ferdinando Santacroce, 2016).
- Apply changes – until “apply changes” command is not invoked explicitly, there is no one symbol changed under original source code location. So, in order changes take effect and become available/visible for OS/GIT/etc. the “Save” function must be executed. Also, current directory should be specified, so Roslyn could apply changes, and not to create copy of the entire solution with applied changes.
- Commit message – the last thing refactoring tool need to complete is to create meaningful commit message and stage changes under VCS. Meaningful commit message does not affect code quality or development process directly, however, during feature changes review, it will be easier to understand context of the changes.

Automation of the refactoring process could be complete by pushing changes to a remote VCS server, but since no one automated system can guarantee 100% error-free operation, all changes must be reviewed and one more time confirmed by the responsible developer.

3.3. Supporting Tool

Clean Architecture(Martin, 2018) is a standard approach to developing the architecture of modern applications, therefore, Figure 12 demonstrates the architectural design of the suggested tool in accordance with its principles and practices.

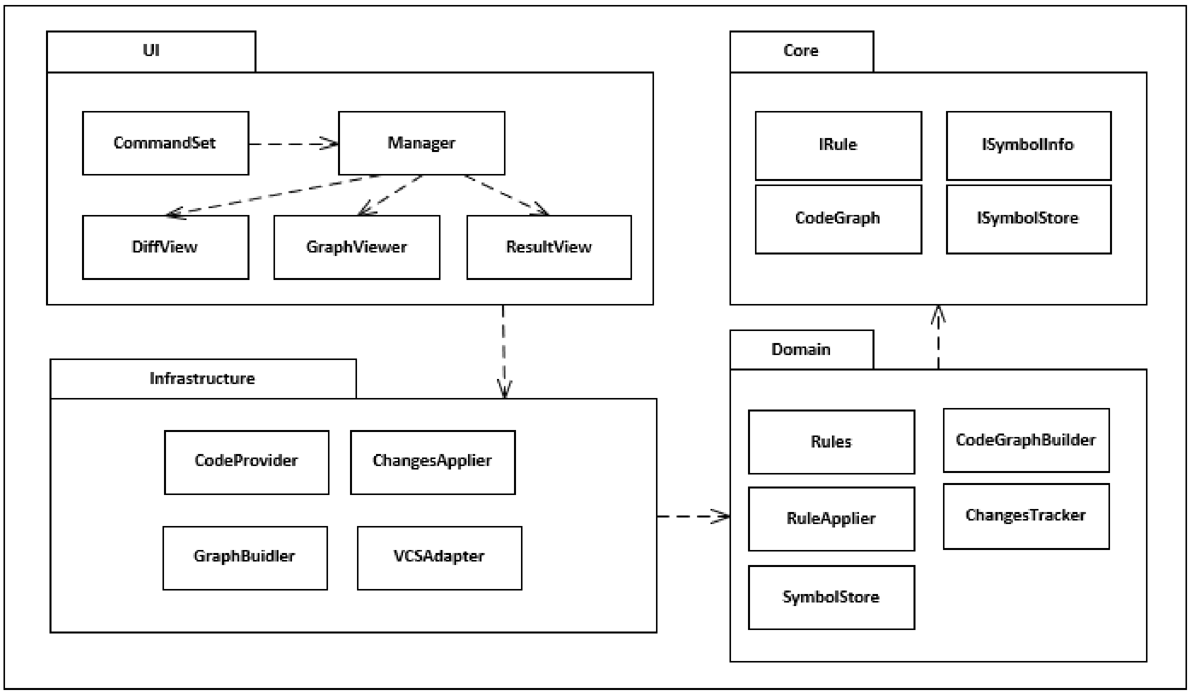


Figure 12. Architecture of logical components for supported tool.

The schema provided shows relations between components and the direction they are dependent on. This schema lacks information about actual components required but demonstrates how responsibility is divided between layers.

- Core layer – contains only base definitions (IRule, ISymbolInfo,..) and core concepts (CodeGraph).
- Domain layer – has everything to process code, detect issues and make changes inside internal code representation.
- Infrastructure layer – relates to code processing on files level.
- UI layer – integration into Visual Studio point, so developer could interact with the tool.

3.4. Main Results of the Third Section

Throughout this section, next goals were achieved:

1. A novel method to fix asynchronous code issues with the main focus on code graph asynchronization was proposed.

2. Technologies and libraries were selected to automate proposed method and conduct the future experiment.
3. A list of issues to detect and fix was provided and described.
4. An interaction plan between the automated solution and the user has been created.

4. Method Validation

4.1. Implementation of the Tool

During the implementation process of the tool, test projects were created to validate the process of identifying and resolving issues. While these projects are not vital for the experiment, they provide precise insights into the tool`s approach to deals with certain type of issues.

The source code of the test projects is available under the next link: https://studgit.vilniustech.lt/20221561/VTech-AsyncRefactoring-TestProjects.

4.1.1. Test Project 1

The first thing at any project is to make it work in a most basic case. So, the initial test project is with only one issue and one method. The structure is shown on the Figure 13.

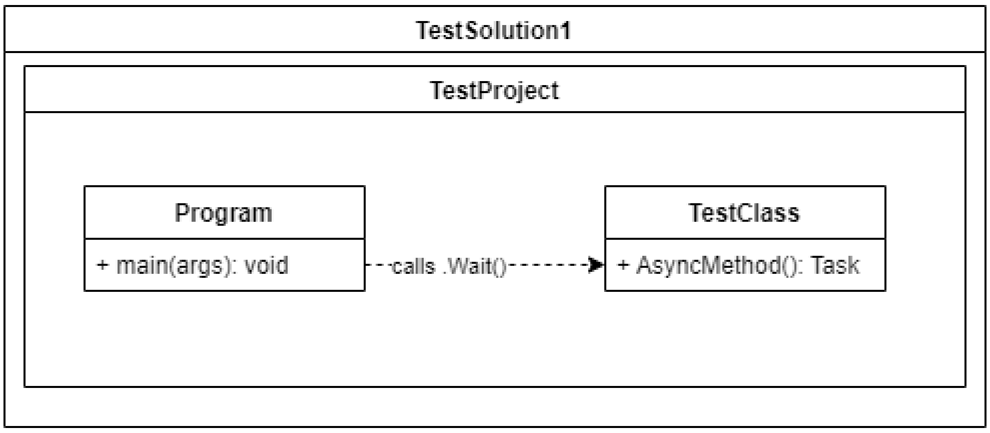
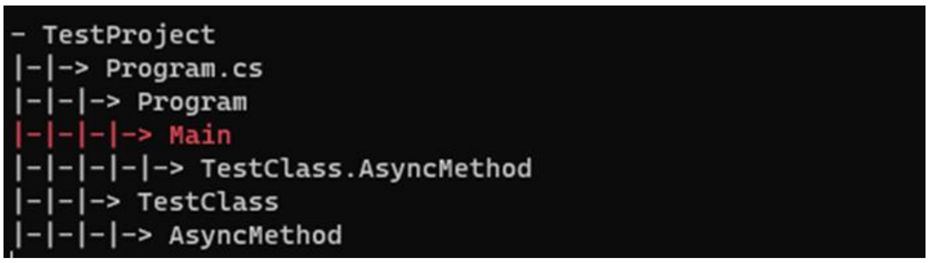


Figure 13. Initial scheme of Test project 1.

According to the structure provided, one issue exists, so the only one method is expected to be suggested for changes. As it is shown on Figure 14 changes should be applied to *main* method only and include method asynchronization plus update of the way how AsyncMethod is called.



```
- TestProject
|-|-> Program.cs
|-|-|-> Program
|-|-|-|-> Main
|-|-|-|-|-> TestClass.AsyncMethod
|-|-|-> TestClass
|-|-|-|-> AsyncMethod
```

Figure 14. Changes suggestion for Test Project 1.

Figure 15 shows the structure of the project after the changes were applied. Main method were updated to return *Task* instead of *void* and AsyncMethod is called in asynchronous manner.

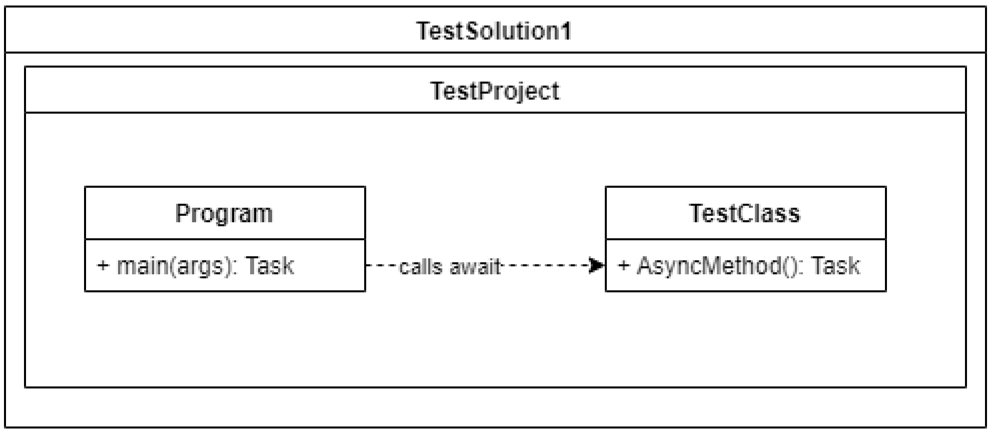


Figure 15. Scheme of Test project 1 after changes applied.

4.1.2. Test Project 2

The second test project provides more complex structure where multiple issues should be detected. One more important difference that the structure of the solution contains more than one project. The scheme is provided on Figure 16.

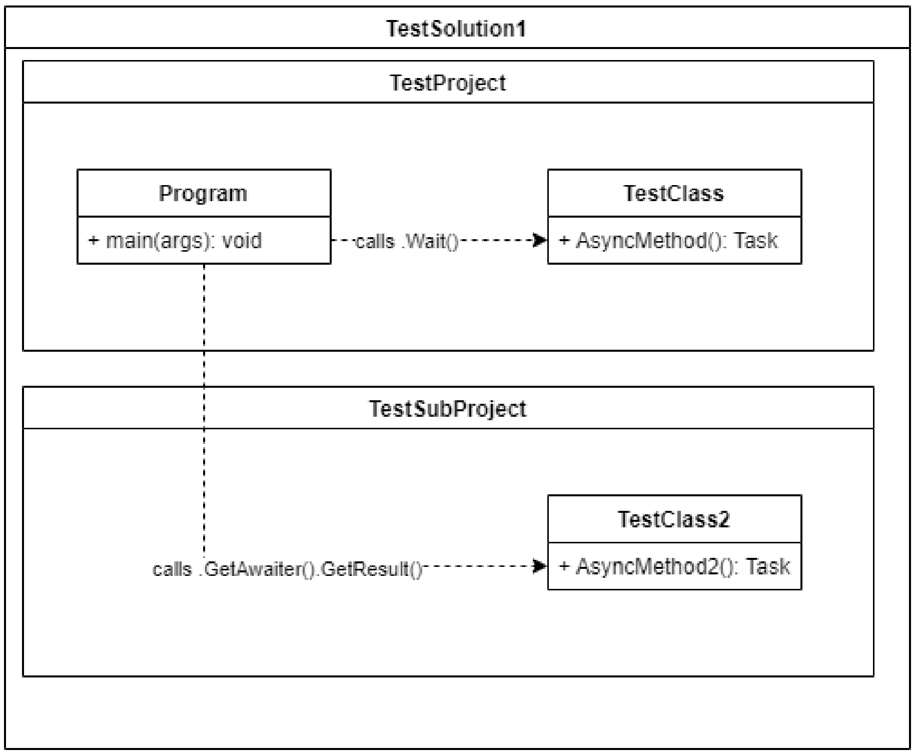
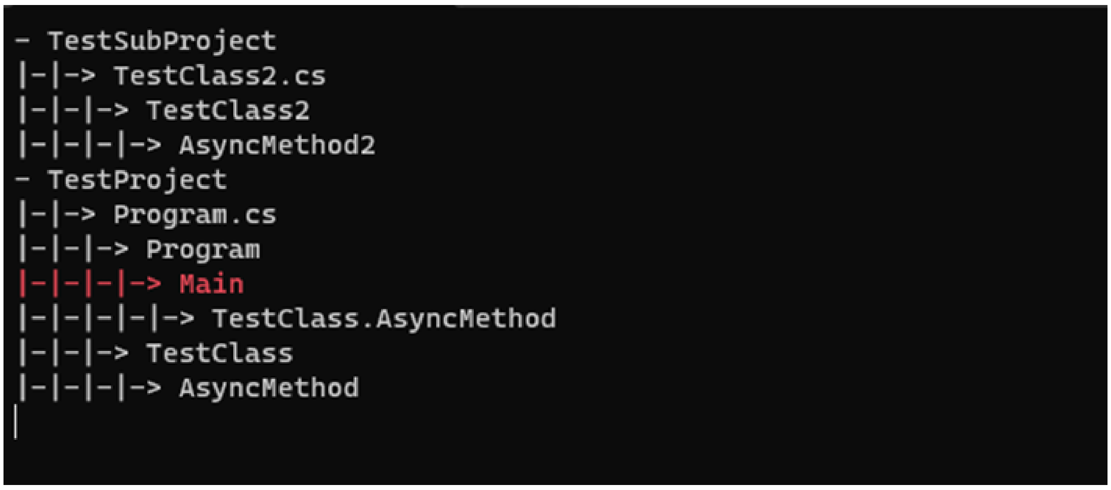


Figure 16. Initial scheme of Test project 2.

According to the scheme there are 2 blocking call that should be fixed and as a result *main* method should be asynchronized. Those changes are proposed by the tool and provided on Figure 17.



```
- TestSubProject
|-|-> TestClass2.cs
|-|-|-> TestClass2
|-|-|-|-> AsyncMethod2
- TestProject
|-|-> Program.cs
|-|-|-> Program
|-|-|-|-> Main
|-|-|-|-|-> TestClass.AsyncMethod
|-|-|-> TestClass
|-|-|-|-> AsyncMethod
|
```

Figure 17. Changes suggestion for Test Project 2.

After the proposed changes are applied, the structure of the project is changed as expected. One is shown on Figure 18.

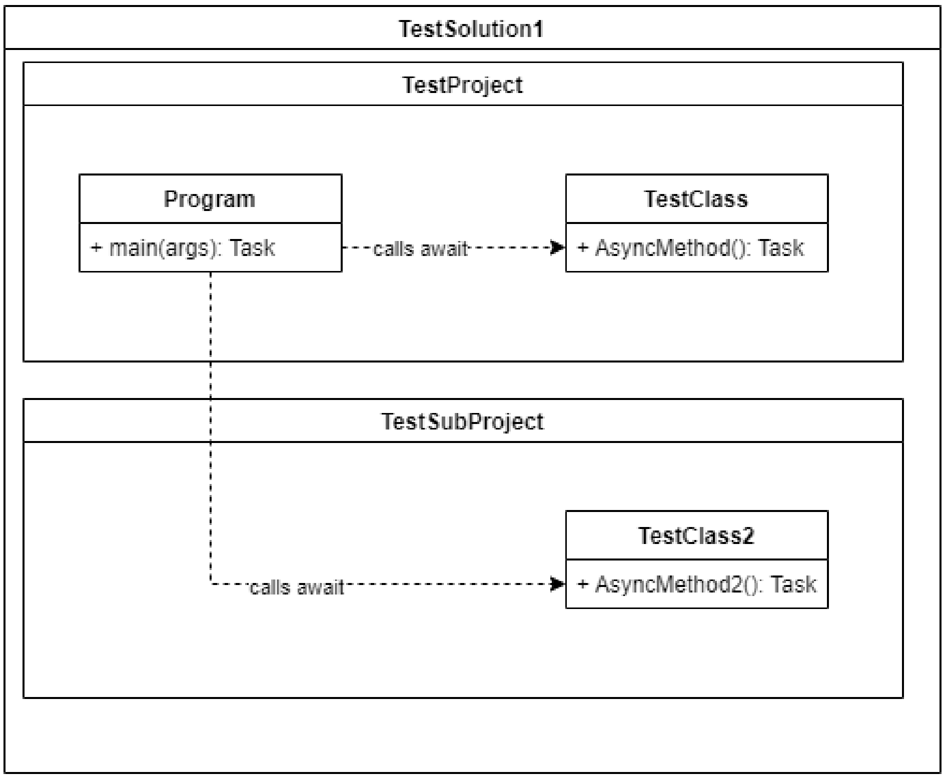


Figure 18. Scheme of Test project 2 after changes applied.

4.1.3. Test Project 3

The third project evaluates the capability of the tool to handle errors that require multiple levels of asynchronization in order to be fixed. Additionally, the project contains a synchronous call path that must remain unchanged. The scheme is provided on Figure 19.

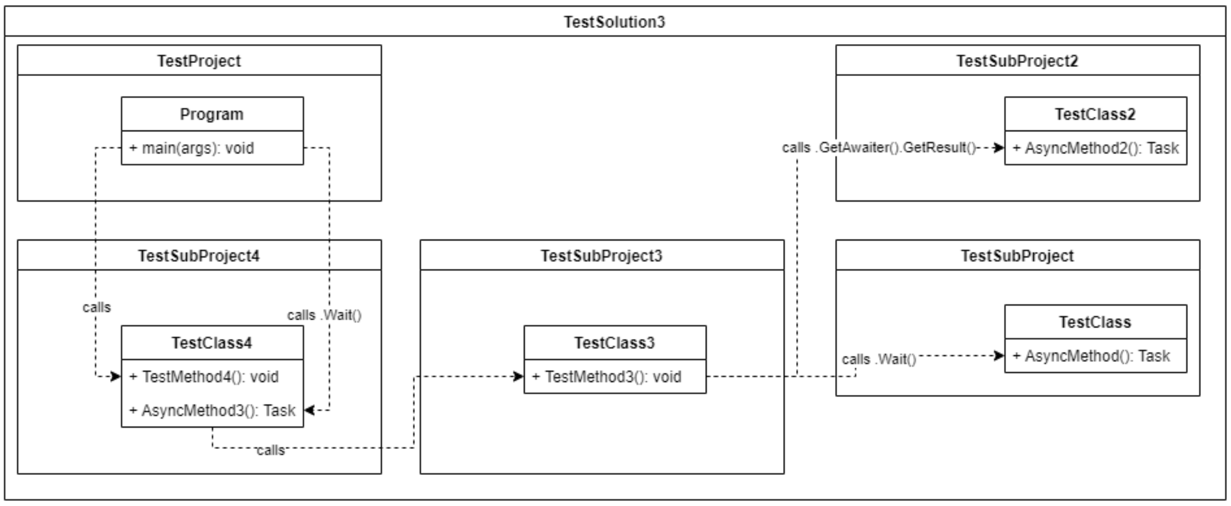
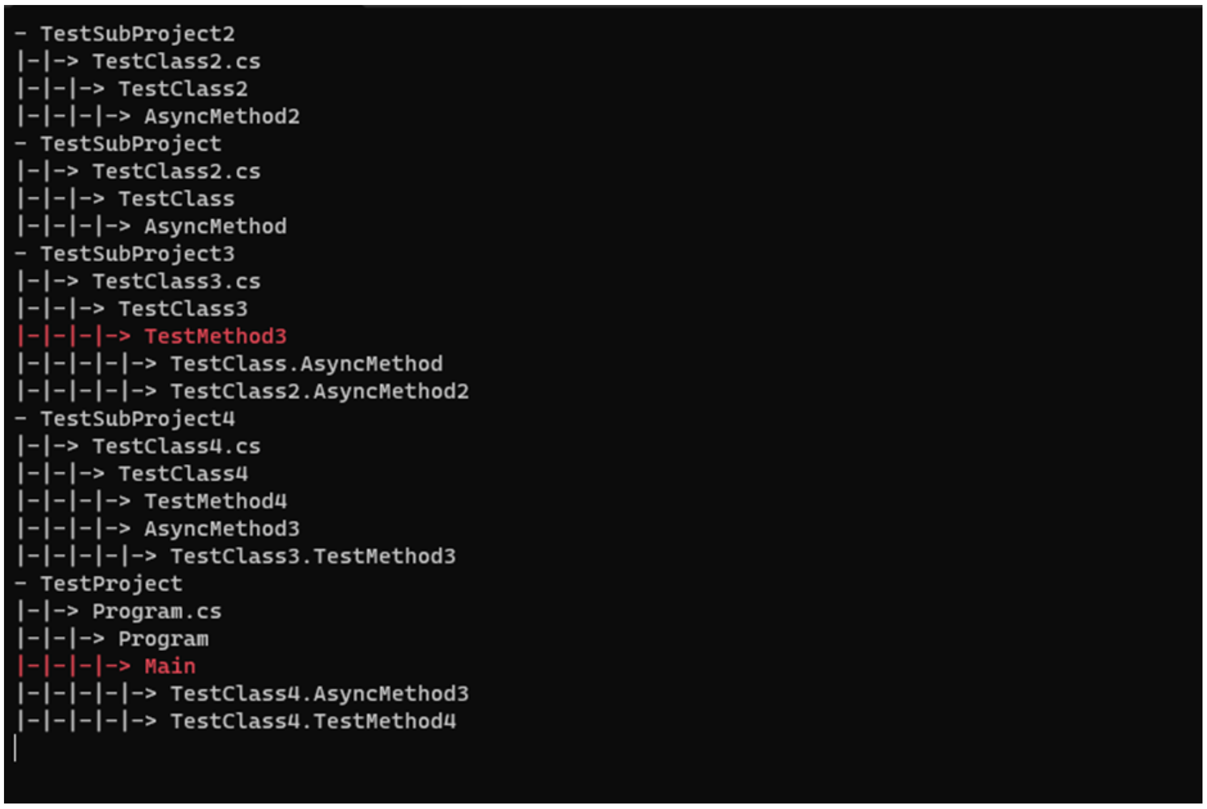


Figure 19. Initial scheme of Test project 3.

As per the scheme provided, three blocking methods calls should be translated to asynchronous ones. TestMethod3 should be asynchronized and renamed, and as a result should be called in async way either. The same as in the previous examples *main* method should be asynchronized in the same manner. TestMethod4 is fully synchronous and should not be updated. Those changes are suggested on Figure 20.



```
- TestSubProject2
|-|-> TestClass2.cs
|-|-|-> TestClass2
|-|-|-|-> AsyncMethod2
- TestSubProject
|-|-> TestClass2.cs
|-|-|-> TestClass
|-|-|-|-> AsyncMethod
- TestSubProject3
|-|-> TestClass3.cs
|-|-|-> TestClass3
|-|-|-|-> TestMethod3
|-|-|-|-|-> TestClass.AsyncMethod
|-|-|-|-|-> TestClass2.AsyncMethod2
- TestSubProject4
|-|-> TestClass4.cs
|-|-|-> TestClass4
|-|-|-|-> TestMethod4
|-|-|-|-> AsyncMethod3
|-|-|-|-|-> TestClass3.TestMethod3
- TestProject
|-|-> Program.cs
|-|-|-> Program
|-|-|-|-> Main
|-|-|-|-|-> TestClass4.AsyncMethod3
|-|-|-|-|-> TestClass4.TestMethod4
|
```

Figure 20. Changes suggestion for Test Project 3.

The project's structure is altered as anticipated following the implementation of the suggested modifications. The resulted scheme can be found in Figure 21.

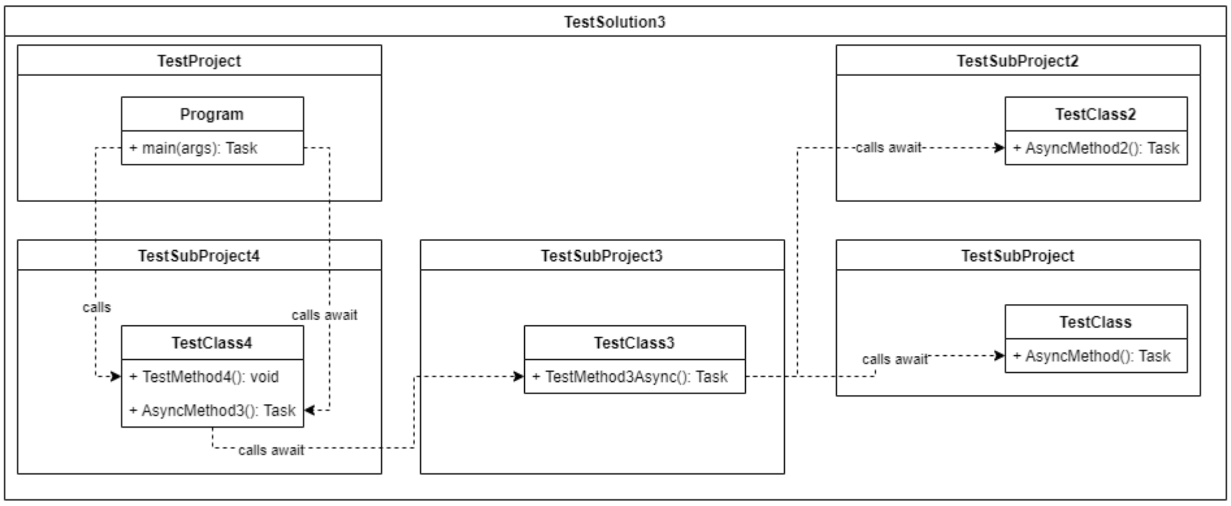


Figure 21. Scheme of Test project 3 after changes applied.

4.1.4. Extension for Visual Studio IDE

Upon successfully testing the logic for handling issues using the test projects, the development of an extension for Visual Studio IDE was successfully completed. The development of the extension was executed according to the requirements formulated in section 3.2.

According to the start point of the workflow is a selected method. In order to select method or set of methods, a developer needs to call the context menu on the desired method or class and select “Find and Fix async issues” command (Figure 22).

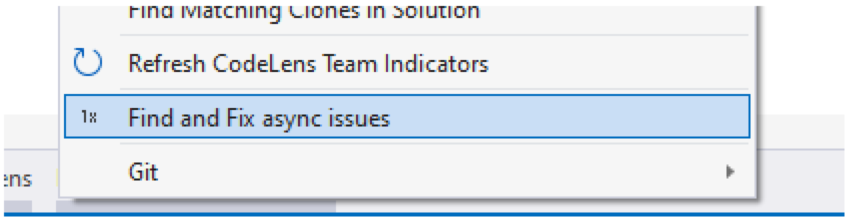
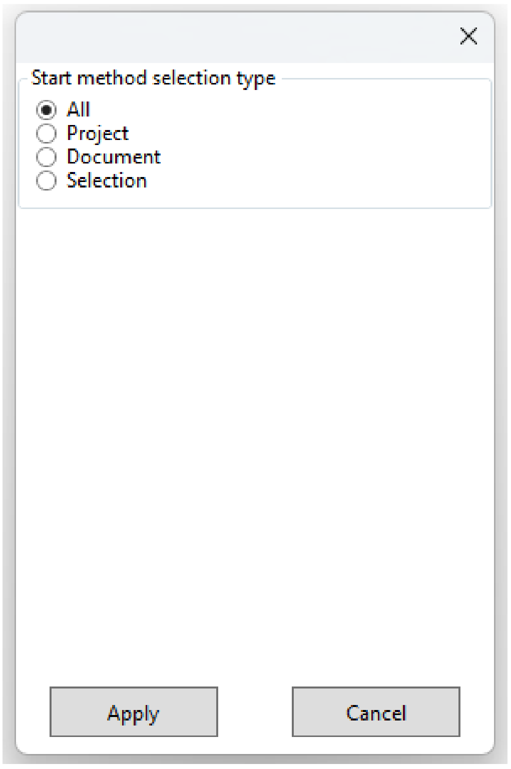

Find Matching Clones in Solution
Refresh CodeLens Team Indicators
Find and Fix async issues
Git

Figure 22. Context menu with command.

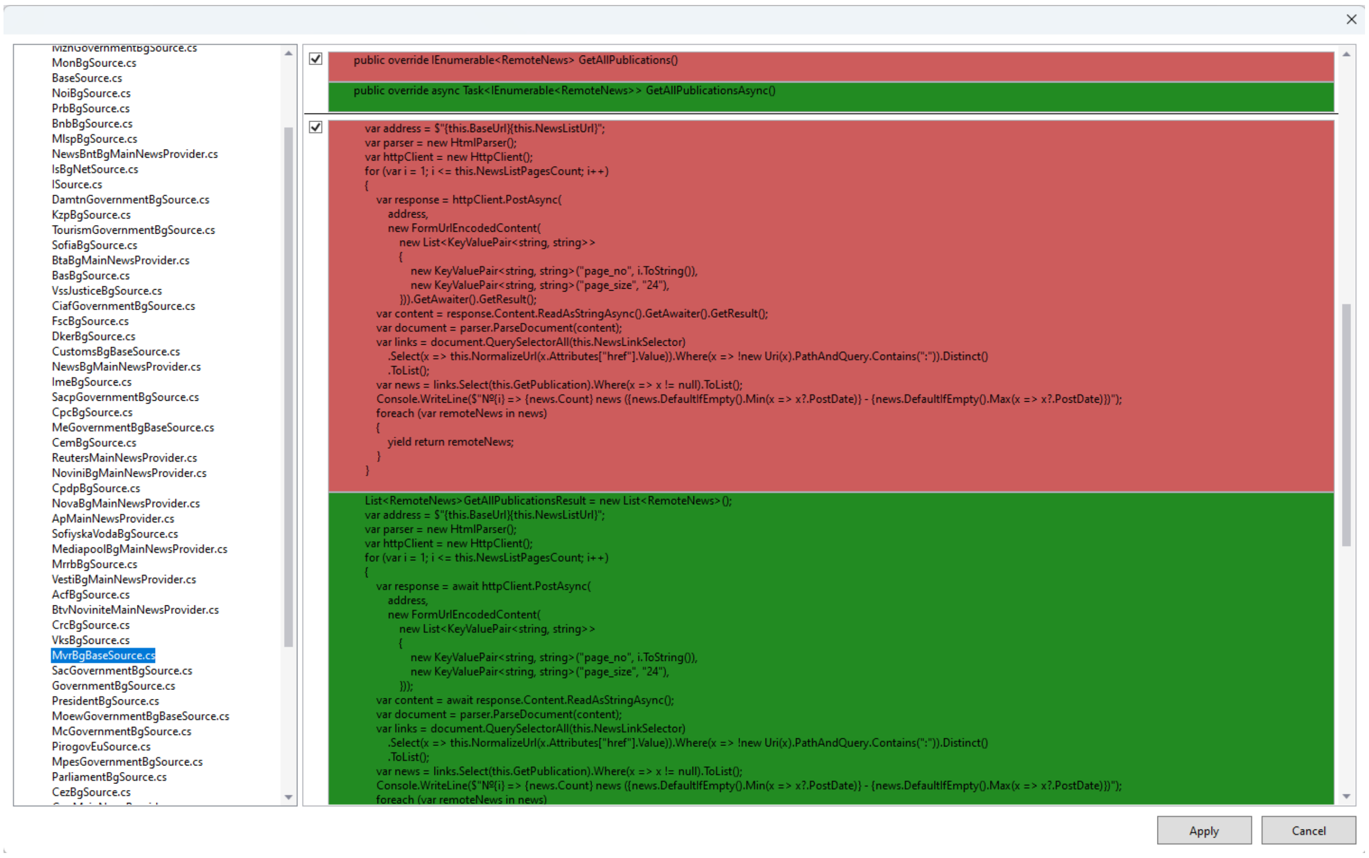
After the execution of the command started the developer is able to choose whether the call graph should be created for current method only, or for all methods within class/project/solution (Figure 23).


Start method selection type

[✓ All]
Project
Document
Selection

Figure 23. Call graph creation scope.

On the next step, the suggested changes are presented to the developer (Figure 24). Within the current form, the developer has the capability to review the suggested changes. Once the change has been reviewed, the developer has the option to reject, approve or modify the change. The final confirmation leads to changes being applied.



```
MznGovernmentBgSource.cs
MonBgSource.cs
BaseSource.cs
NoiBgSource.cs
PrbBgSource.cs
BnbBgSource.cs
MlspBgSource.cs
NewsBntBgMainNewsProvider.cs
IsBgNetSource.cs
ISource.cs
DamtnGovernmentBgSource.cs
KzpBgSource.cs
TourismGovernmentBgSource.cs
SofiaBgSource.cs
BtaBgMainNewsProvider.cs
BasBgSource.cs
VssJusticeBgSource.cs
CiafGovernmentBgSource.cs
FscBgSource.cs
DkerBgSource.cs
CustomsBgBaseSource.cs
NewsBgMainNewsProvider.cs
ImeBgSource.cs
SacpGovernmentBgSource.cs
CpcBgSource.cs
MeGovernmentBgBaseSource.cs
CemBgSource.cs
ReutersMainNewsProvider.cs
NoviniBgMainNewsProvider.cs
CpdpBgSource.cs
NovaBgMainNewsProvider.cs
ApMainNewsProvider.cs
SofiyskaVodaBgSource.cs
MediapoolBgMainNewsProvider.cs
MrrbBgSource.cs
VestiBgMainNewsProvider.cs
AcfBgSource.cs
BtvNoviniteMainNewsProvider.cs
CrcBgSource.cs
VksBgSource.cs
MvrBgBaseSource.cs
SacGovernmentBgSource.cs
GovernmentBgSource.cs
PresidentBgSource.cs
MoewGovernmentBgBaseSource.cs
McGovernmentBgSource.cs
PirogovEuSource.cs
MpesGovernmentBgSource.cs
ParliamentBgSource.cs
CezBgSource.cs
```

```
public override IEnumerable<RemoteNews> GetAllPublications()
```

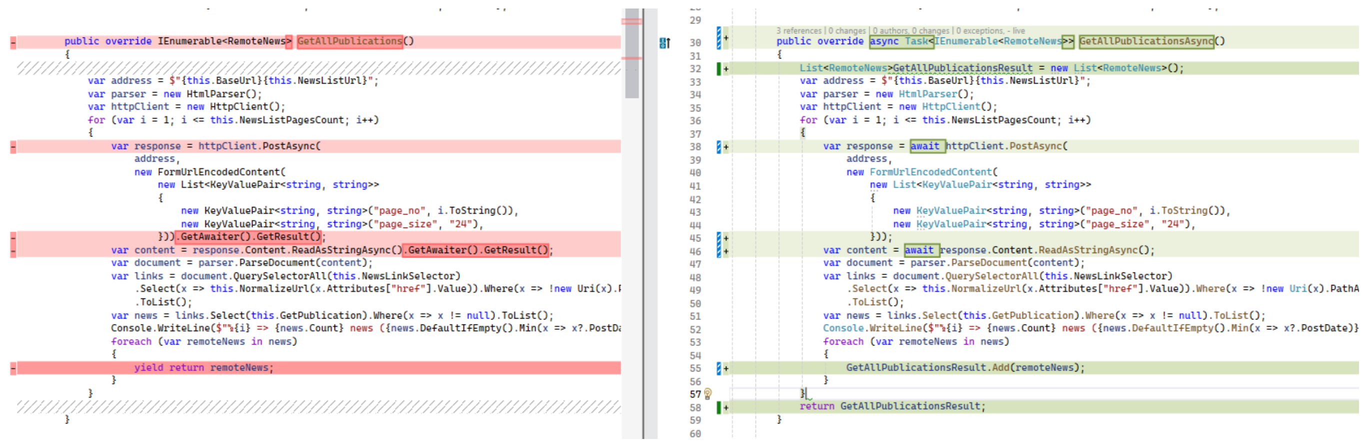
```
public override async Task<IEnumerable<RemoteNews>> GetAllPublicationsAsync()
```

```
var address = $"{this.BaseUrl}{this.NewsListUrl}";
var parser = new HtmlParser();
var httpClient = new HttpClient();
for (var i = 1; i <= this.NewsListPagesCount; i++)
{
    var response = httpClient.PostAsync(
        address,
        new FormUrlEncodedContent(
            new List<KeyValuePair<string, string>>
            {
                new KeyValuePair<string, string>("page_no", i.ToString()),
                new KeyValuePair<string, string>("page_size", "24"),
            })).GetAwaiter().GetResult();
    var content = response.Content.ReadAsStringAsync().GetAwaiter().GetResult();
    var document = parser.ParseDocument(content);
    var links = document.QuerySelectorAll(this.NewsLinkSelector)
        .Select(x => this.NormalizeUrl(x.Attributes["href"].Value)).Where(x => !new Uri(x).PathAndQuery.Contains(":")).Distinct()
        .ToList();
    var news = links.Select(this.GetPublication).Where(x => x != null).ToList();
    Console.WriteLine($"№{i} => {news.Count} news ({news.DefaultIfEmpty().Min(x => x?.PostDate)} - {news.DefaultIfEmpty().Max(x => x?.PostDate)})");
    foreach (var remoteNews in news)
    {
        yield return remoteNews;
    }
}
```

```
List<RemoteNews> GetAllPublicationsResult = new List<RemoteNews>();
var address = $"{this.BaseUrl}{this.NewsListUrl}";
var parser = new HtmlParser();
var httpClient = new HttpClient();
for (var i = 1; i <= this.NewsListPagesCount; i++)
{
    var response = await httpClient.PostAsync(
        address,
        new FormUrlEncodedContent(
            new List<KeyValuePair<string, string>>
            {
                new KeyValuePair<string, string>("page_no", i.ToString()),
                new KeyValuePair<string, string>("page_size", "24"),
            }));
    var content = await response.Content.ReadAsStringAsync();
    var document = parser.ParseDocument(content);
    var links = document.QuerySelectorAll(this.NewsLinkSelector)
        .Select(x => this.NormalizeUrl(x.Attributes["href"].Value)).Where(x => !new Uri(x).PathAndQuery.Contains(":")).Distinct()
        .ToList();
    var news = links.Select(this.GetPublication).Where(x => x != null).ToList();
    Console.WriteLine($"№{i} => {news.Count} news ({news.DefaultIfEmpty().Min(x => x?.PostDate)} - {news.DefaultIfEmpty().Max(x => x?.PostDate)})");
    foreach (var remoteNews in news)
```

Figure 24. Changes proposal.

After successfully applying the changes, Visual Studio IDE has the ability to demonstrate a comparison of unmodified and modified source code using a built-in tool (Figure 25) highlighting exact code blocks that have been changed.



```
-       public override IEnumerable<RemoteNews> GetAllPublications()
        {
            var address = $"{this.BaseUrl}{this.NewsListUrl}";
            var parser = new HtmlParser();
            var httpClient = new HttpClient();
            for (var i = 1; i <= this.NewsListPagesCount; i++)
            {
-               var response = httpClient.PostAsync(
                    address,
                    new FormUrlEncodedContent(
                        new List<KeyValuePair<string, string>>
                        {
                            new KeyValuePair<string, string>("page_no", i.ToString()),
                            new KeyValuePair<string, string>("page_size", "24"),
-                       })).GetAwaiter().GetResult();
-               var content = response.Content.ReadAsStringAsync().GetAwaiter().GetResult();
                var document = parser.ParseDocument(content);
                var links = document.QuerySelectorAll(this.NewsLinkSelector)
                    .Select(x => this.NormalizeUrl(x.Attributes["href"].Value)).Where(x => !new Uri(x).F
                    .ToList();
                var news = links.Select(this.GetPublication).Where(x => x != null).ToList();
                Console.WriteLine($"№{i} => {news.Count} news ({news.DefaultIfEmpty().Min(x => x?.PostDa
                foreach (var remoteNews in news)
                {
-                   yield return remoteNews;
                }
            }
        }
```

```
        3 references | 0 changes | 0 authors, 0 changes | 0 exceptions, - live
+       public override async Task<IEnumerable<RemoteNews>> GetAllPublicationsAsync()
        {
+           List<RemoteNews>GetAllPublicationsResult = new List<RemoteNews>();
            var address = $"{this.BaseUrl}{this.NewsListUrl}";
            var parser = new HtmlParser();
            var httpClient = new HttpClient();
            for (var i = 1; i <= this.NewsListPagesCount; i++)
            {
+               var response = await httpClient.PostAsync(
                    address,
                    new FormUrlEncodedContent(
                        new List<KeyValuePair<string, string>>
                        {
                            new KeyValuePair<string, string>("page_no", i.ToString()),
                            new KeyValuePair<string, string>("page_size", "24"),
+                       }));
+               var content = await response.Content.ReadAsStringAsync();
                var document = parser.ParseDocument(content);
                var links = document.QuerySelectorAll(this.NewsLinkSelector)
                    .Select(x => this.NormalizeUrl(x.Attributes["href"].Value)).Where(x => !new Uri(x).PathA
                    .ToList();
                var news = links.Select(this.GetPublication).Where(x => x != null).ToList();
                Console.WriteLine($"№{i} => {news.Count} news ({news.DefaultIfEmpty().Min(x => x?.PostDate)}
                foreach (var remoteNews in news)
                {
+                   GetAllPublicationsResult.Add(remoteNews);
                }
            }
+           return GetAllPublicationsResult;
        }
```

Figure 25. Applied changes preview of Visual Studio IDE.

The source code of the extension is available under the next link: https://studgit.vilniustech.lt/20221561/VTech-AsyncRefactoring-Extension.

Also, the extension was published to the Visual Studio Marketplace and is available under the next link: https://marketplace.visualstudio.com/items?itemName=BorisK.AsyncRefactoring.

4.2. Experiment

4.2.1. Planning the Experiment

In order to conduct an experiment, the next plan is designed:

1. Select projects to evaluate the performance of the method designed
2. Create a survey for participants to validate the method
3. Collect the results
4. Compare the effort required for manual and automatic issue processing

Projects selection

When selecting projects, Github is used, being the most common repository of open-source projects. Project filtering is based on two main criteria:

1. C# language should be used as the main one
2. .Wait(), .Result, .GetAwaiter(), .GetResult() constructions should exist within full-text search.

Despite the fact that the constructions listed above are not necessarily used within Task type, hence such search helps to filter out projects that do not use TPL.

The third search criteria is the path inside the project where source code files are located. According to conventional practice, the "*src*" folder is typically used to store those files. Excluding directories such as "*tests*" and "*build*" helps to eliminate issue detection within secondary code.

An additional requirement is for a project to be active, meaning that if it has not been updated within the past year, it is considered abandoned. The activity criteria are determined by the intention of implementing modifications that would potentially assist other developers in fixing issues described in Section 2.2.1.

Taking into account that participants of the experiment will be developers whose time is extremely valuable and hardly available, specific requirements are raised:

- The size of the project should be as small as possible, so the experiment would not take more than one hour. The possible number of source files should be less than 100.
- The project should be possible to build without the use of additional scripts and tools. In order to achieve that, the project should use the current tech stack (at least .net 6) and there should not be a folder or file named "*build*".
- Changes made by developers while performing the experiment should be easily validated by running the project. Whereas console or web applications are started easily, frameworks and libraries cannot be checked in such a straightforward manner. So, projects with "*Program.cs*" or "*Startup.cs*" files should be selected.

The next step is to perform an analysis of the projects that have been filtered previously, exclusively choosing those that contain blocking code and meet previous requirements.

Development of the survey

Developers who participate in the experiment should be provided with projects to process and given explicit instructions on specific types of issues that need to be identified and corresponding rules for resolving them. In order to achieve that, a clear survey should be conducted.

At the first step, the survey should clearly state the reasons and aims of the experiment. Also, general information about blocking code and its consequences should be provided. Besides this, system requirements should be clearly stated.

The second step should involve collecting information both about system configuration and professional experience. Each of these may have an impact on the results of the experiment, so the analysis of the results should consider them.

The third section of the survey should provide clear instructions on how to get the source code of the test project and how to build it. Also, in the case of a project with multiple solutions available, the exact one should be outlined to be used.

The fourth part is about manual issue detection and fixing. This part describes exactly how manual processing should be done. A precise check list should be provided with both instructions on what to measure and how to measure it, along with information on where to save the results of the measurements.

The final part of the survey should contain the same instructions as the previous one, but for automatic issue detection and fixing. A precise checklist should also be created. However, after the main part is finished, some kind of feedback about the tool should be gathered, such as about performance, usability, and overall satisfaction.

Collecting results

Once the developers have completed their tasks, all changes should be committed to VCS, and the times taken to accomplish each task should be entered into the survey of the experiment.

In order for the process of collecting results to be more accurate, the survey should be available in the form of an e-document with predefined fields to be filled.

Results processing

After all surveys are obtained, the final validation should be done. A comparison should be done based on the next values:

- Difference between the number of issues detected manually and by tool
- Difference between time required for manual and automatic processing

Based on the numbers calculated above, the final decision should be made if the tool developed is more efficient than manual processing and the proposed method is efficient.

However, the time required to detect issues manually is related to the expertise of the developer, so calculations should respect the level of expertise of the participants. Also, the time to automatically process the project is directly proportional to the calculation power of the workstation where the experiment is conducted.

Additionally, the feedback gathered should be examined to determine the pros and cons of the proposed approach and tool.

4.2.2. Evaluation

The initial filtering of the projects available on Github resulted in 900 projects for the ".GetAwaiter()" keyword and 800 for the ".Wait()" keyword. While combining those keywords resulted in 600 projects. Due to the huge number of projects containing ".Result" text, the rational decision to skip this keyword was made.

Then the descriptions of the projects were reviewed. According to the previously developed plan, all projects whose description indicated that they were frameworks or libraries were skipped. At this point, there are 320 repositories left.

The next stage was performed manually under the strict guidance of a supervisor. The review of each project included an analysis of:

- The description of the project
- The size of the project
- The type of the project
- The number and location of the blocking code inside the project

After the analysis was completed, "presscenters.com" project was selected as a perfect candidate to perform the experiment on. This project has 10 blocking code constructions, and the process of asynchronization requires dealing with implementations and overrides.

During scanning Github for projects to perform the experiment on, it was noticed that mostly all of them require a deep understanding of project aims and structure. As a result, it may take too long to conduct the experiment on them. Thus, it was decided to use a previously developed test project. This project has a simple, mostly linear structure, so developers could focus on finding issues and fixing them without diving into architectural questions.

Right after projects were selected, surveys for the experiment were developed. Although the main procedure is the same for both projects, there are certain differences. Both surveys are available in the Annexes of this paper.

After conducting the experiment on the test project, the results presented in Table 15 were obtained.

Table 15. Test project experiment results.

	Dev. 1	**Dev. 2**	**Dev. 3**	**Dev. 4**	**Dev. 5**
CPU	Intel core i5	Intel core i5 13600KF	AMD Ryzen 5 3600X	Intel core i7 11800H	Intel core i7 11800H
RAM	16 GB	48 GB	48 GB	32 GB	64 GB
Expertise	Middle	Senior	Senior	Senior	Lead
Time for manual processing	20 min	3 min	3:30 min	5 min	3 min
Time for automatic processing	6 min	30 sec	1 min	10 sec	15 sec
Manually issues processed	100 %	100 %	100 %	100 %	100 %
Automatically issues processed	100 %	100 %	100 %	100 %	100 %

According to the obtained results, there is a strong correlation between the level of the developer and the time required to detect and fix issues manually. While developers at the senior level managed to complete tasks in 4 minutes on average, for middle level developer 20 minutes were required.

Automatic processing was at least 3 times faster than manual processing. Again, senior level developers managed to complete tasks at more than 5 times faster than middle-level one.

Table 16. Test project experiment feedback.

	Dev. 1	Dev. 2	Dev. 3	Dev. 4	Dev. 5
Easy to use	5	5	5	5	5
Performance	5	5	5	5	5
Recommend	5	5	5	5	5
Overall satisfaction	5	5	5	5	5

The feedback presented in Table 16 shows that developers were extremely satisfied with the method and the tool proposed.

Table 17. Presscenters.com project experiment results.

	Dev. 1	Dev. 2	Dev. 3	Dev. 4	Dev. 5
CPU	Intel core i5	Intel core i5 13600KF	AMD Ryzen 5 3600X	Intel core i7 11800H	Intel core i7 11800H
RAM	16 GB	48 GB	48 GB	32 GB	64 GB
Expertise	Middle	Senior	Senior	Senior	Lead
Time for manual processing	40 min	60 min	90 min	60 min	50 min
Time for automatic processing	3 min	5 min	3 min	2 min	2 min
Manually issues processed	10 %	100 %	100 %	100 %	100 %
Automatically issues processed	95 %	95 %	95 %	95 %	95 %

Table 17 presents the results obtained during the experiment processed on “Presscenters.com” project. All senior-level developers managed to detect and fix all issues manually, with an average time required of more than an hour. On the other side, the middle developer was not able to detect and fix all issues within the 40-minute time limit.

For all levels of developers, automatic processing took no more than 5 minutes. However, the tool proposed did not manage to fix all issues, so the required time includes manual processing.

Table 18. Presscenters.com project experiment feedback.

	Dev. 1	Dev. 2	Dev. 3	Dev. 4	Dev. 5
Easy to use	5	5	5	4	5
Performance	5	5	5	5	5
Recommend	5	5	5	5	5
Overall satisfaction	5	5	5	4	4

According to Table 18, the satisfaction rating of using the extension is lower for more complex projects. Based on free-text comments provided by the participant, the tool lacks support for fixing lambdas and should provide a dependency tree for proposed changes. However, it was mentioned that even with the current implementation, it is an incredible improvement in performance when dealing with asynchronization tasks.

Summarizing the research results, the proposed method, and the developed extension results in significant performance improvements, even when dealing with small projects.

4.3. Threats to Validity

This section addresses potential threats to the validity of the research results on the proposed method.

4.3.1. Internal Validity

Several factors could affect the internal validity of the thesis:

Sample size and diversity: The experiment was conducted on 2 projects only. The small scope of projects, especially those been selected as simple as possible, do not sufficiently capture the variety of .Net applications, which could potentially introduce bias in the results.

Developer expertise: The limited number of developers who participated in the experiment might not capture the full range of potential issues or benefits due to the limited differences in individual skills when working with asynchronous code.

4.3.2. External Validity

The external validity of the research is limited by the scale of the experiment:

Generalizability: Testing on 2 projects within a single development environment limits the generalizability of the results. Testing the proposed method on different types of projects may result in different asynchronization requirements.

Tool applicability: The tool was developed to be used with the latest .Net version under the latest release of Visual Studio IDE. Using the tool with the older versions may result in lower performance and/or being completely unsuitable.

4.3.3. Construct Validity

Several factors could impact the construct validity of the research:

Practical application of blocking code: The tool was developed based on widely known examples of blocking code. Nevertheless, there is a high possibility of uncommon or exceptional cases of blocking code usage that were not investigated during the research.

Effectiveness metrics: The research evaluates only quantitative metrics, while qualitative metrics like maintainability and readability are not studied.

4.3.4. Conclusion Validity

The validity of conclusions could be impacted by:

Statistical power: With only 2 project and 5 participants, the sample size is small, limiting the statistical power of the experiment. Large samples, both for project type and participants, are necessary to provide more reliable conclusions.

Results interpretation: The obtained results might be influenced not only by the developed tool and proposed method but also by other tools used by developers and/or their experience with asynchronous programming principles.

4.4. Main Results of the Fourth Section

Throughout this section, several primary goals were successfully achieved. Every stage of accomplishing them was carefully executed and accurately recorded.

The tool to validate the efficiency of the proposed method was developed using the previously described architecture. In order to validate the correct behavior of the tool, test projects were created. The tool was implemented as an extension for the Visual Studio IDE, allowing developers to utilize it in a familiar manner.

The experiment was meticulously designed. The logic for selecting a project for the experiment, as well as additional requirements for the project itself, were formulated. A set of requirements for the survey of the experiment were created. Appropriate metrics to analyze the obtained results were selected.

The experiment was successfully executed in accordance with the previously developed plan. Consequently, reliable and accurate results and feedback were acquired.

The data collected from the experiment was meticulously analyzed. The interpretation of the results confirmed the efficiency of the proposed method and the value of the tool developed.

5. Conclusions

1. After conducting the literature analysis, it was observed that there are 5 main problems that developers address while refactoring synchronous code to asynchronous version, including the evolution of programming languages, callbacks hell, blocked treads, performance issues, and anti-patterns.
2. Research on methods and techniques for dealing with synchronous code issues stated that each issue has its own way of fixing the core problem; however, the entire fixing process includes asynchronization of the entire call graph. Although the research pointed out that no previous research discussed the entire call graph asynchronization tasks. Moreover, investigation of the available refactoring tools has shown that there is a lack of tools for almost every type of asynchronization process.
3. Based on the previously detected problem, an improved method of asynchronization was proposed. The main idea of the method is to check all child call graphs on each step of graph asynchronization, thus processing all possible issues.
4. In order to validate the proposed method, a tool was created. The tool was implemented as an extension to Visual Studio IDE. A number of tests were performed on common cases to verify correct behavior of the developed tool. The experiment was conducted and resulted in confirming that the proposed method is effective and may improve the efficiency of developers work at least twice, and in some cases up to 10 times.

5.1. Further Works

Although the proposed method has demonstrated a significant improvement in developers` performance, additional research is required to broaden the study. Subsequent work should focus on:

- Investigation of more complex and advanced blocking code constructions, that have not been investigated in this work;
- Investigation of transformations applied by developers in order to fix all studied issues;
- Development of methods to select the most appropriate transformation in each situation.

The new knowledge obtained through further research might additionally improve the efficiency of developers` work while performing the refactoring task.

6. References

Al Dallal, J., & Abdin, A. (2018). Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review. *IEEE Transactions on Software Engineering*, *44*(1), 44–69. https://doi.org/10.1109/TSE.2017.2658573

Almogahed, A., & Omar, M. (2021). Refactoring Techniques for Improving Software Quality: Practitioners’ Perspectives. *Journal of Information and Communication Technology*, *20*(No.4), 511–539. https://doi.org/10.32890/jict2021.20.4.3

Almogahed, A., Omar, M., & Zakaria, N. H. (2022). Refactoring Codes to Improve Software Security Requirements [Article]. *Procedia Computer Science, 204,* 108–115. https://doi.org/10.1016/j.procs.2022.08.013

Antinyan, V., Staron, M., & Sandberg, A. (2017). Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time [Article]. *Empirical Software Engineering : An International Journal*, *22*(6), 3057–3087. https://doi.org/10.1007/s10664-017-9508-2

Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., & Strollo, O. (2012). When does a refactoring induce bugs? An empirical study. *Proceedings - 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, SCAM 2012*, 104–113. https://doi.org/10.1109/SCAM.2012.20

Beillahi, S. M., Bouajjani, A., Enea, C., & Lahiri, S. (2022). *Automated Synthesis of Asynchronizations* [Article]. https://doi.org/10.48550/arxiv.2209.06648

Burchard, E. (2017). *Refactoring JavaScript* [Book]. O’Reilly Media, Incorporated.

de Boer, F. S., Bravetti, M., Lee, M. D., & Zavattaro, G. (2018). A Petri Net Based Modeling of Active Objects and Futures [Article]. *Fundamenta Informaticae*, *159*(3), 197–256. https://doi.org/10.3233/FI-2018-1663

Dig, D. (2015). *Refactoring for Asynchronous Execution on Mobile*. http://learnasync.net,

Dig, D., Marrero, J., & Ernst, M. D. (2009). Refactoring sequential Java code for concurrency via concurrent libraries. *2009 IEEE 31st International Conference on Software Engineering*, 397–407. https://doi.org/10.1109/ICSE.2009.5070539

Evans, B. J., & Flanagan, D. (2018). *Java in a nutshell*.

Fenton, N. E., & Bieman, J. (2015). *Software metrics: a rigorous and practical approach* (3rd ed.) [Book]. CRC Press/Taylor & Francis Group.

Ferdinando Santacroce, A. O. (2016). *Git: Mastering Version Control* [Book]. Packt Publishing.

Fowler, D. (Retrieved 2023, June 15). *Async Guidance*. https://github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AsyncGuidance.md

Fowler, M., & Beck, K. (2019). *Refactoring Improving the Design of Existing Code Second Edition*.

Gallaba, K., Hanam, Q., Mesbah, A., & Beschastnikh, I. (2017). Refactoring Asynchrony in JavaScript. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 353–363. https://doi.org/10.1109/ICSME.2017.83

Gokhale, S., Turcotte, A., & Tip, F. (2021). Automatic migration from synchronous to asynchronous JavaScript APIs. *Proceedings of the ACM on Programming Languages*, *5*(OOPSLA), 1–27. https://doi.org/10.1145/3485537

Jhala, R., & Majumdar, R. (2007). Interprocedural analysis of asynchronous programs [Article]. *SIGPLAN Notices*, *42*(1), 339–350. https://doi.org/10.1145/1190215.1190266

Lin, Y., & Dig, D. (2015). A study and toolkit of CHECK-THEN-ACT idioms of Java concurrent collections [Article]. *Software Testing, Verification & Reliability*, *25*(4), 397–425. https://doi.org/10.1002/stvr.1567

Lin, Y., Radoi, C., & Dig, D. (2014). Retrofitting concurrency for Android applications through refactoring. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 341–352. https://doi.org/10.1145/2635868.2635903

Lu, S., Park, S., Seo, E., & Zhou, Y. (2008). Learning from mistakes [Proceeding]. *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 329–339. https://doi.org/10.1145/1346281.1346323

Martin, R. C. (2018). *Clean architecture: a craftsman's guide to software structure and design* [Book]. Prentice Hall.

Microsoft Asynchronous Programming Patterns. (Retrieved 2023, January). https://learn.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/

Morasca, S. (2009). A probability-based approach for measuring external attributes of software artifacts. *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 44–55. https://doi.org/10.1109/ESEM.2009.5316048

Mukherjee, S. (2016). Source Code Analytics With Roslyn and JavaScript Data Visualization. In *Source Code Analytics With Roslyn and JavaScript Data Visualization*. Apress. https://doi.org/10.1007/978-1-4842-1925-6

Ogden, M. (2023, January 8). *Callback Hell. A guide to writing asynchronous JavaScript programs*. http://callbackhell.com/

Okur, S. (2015). Understanding, Refactoring, and Fixing Concurrency in C#. *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 898–901. https://doi.org/10.1109/ASE.2015.82

Okur, S., Erdogan, C., & Dig, D. (2014). *Converting Parallel Code from Low-Level Abstractions to Higher-Level Abstractions* (pp. 515–540). https://doi.org/10.1007/978-3-662-44202-9_21

Okur, S., Hartveld, D. L., Dig, D., & Deursen, A. Van. (2014). A study and toolkit for asynchronous programming in c#. *Proceedings - International Conference on Software Engineering*, *1*, 1117–1127. https://doi.org/10.1145/2568225.2568309

Orton, I., & Mycroft, A. (2021). Refactoring traces to identify concurrency improvements. *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, 16–23. https://doi.org/10.1145/3464971.3468420

Pacheco, P. S., & Malensek, M. (2022). *An introduction to parallel programming* (Second Edition.) [Book]. Morgan Kaufmann.

Parker, D. (2015). *JavaScript with Promises*.

Parr, T. (2013). *The Definitive ANTLR 4 Reference*.

Şanlıalp, İ., Öztürk, M. M., & Yiğit, T. (2022). Energy Efficiency Analysis of Code Refactoring Techniques for Green and Sustainable Software in Portable Devices [Article]. *Electronics (Basel)*, *11*(3), 442. https://doi.org/10.3390/electronics11030442

Santhiar, A., & Kanade, A. (2017). Static deadlock detection for asynchronous C# programs [Proceeding]. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 292–305. https://doi.org/10.1145/3062341.3062361

Sarcar, V. (2020). *Getting Started with Advanced C#*. Apress. https://doi.org/10.1007/978-1-4842-5934-4

Sodian, L., Wen, J. P., Davidson, L., & Loskot, P. (2022). Concurrency and Parallelism in Speeding Up I/O and CPU-Bound Tasks in Python 3.10. *2022 2nd International Conference on Computer Science, Electronic Information Engineering and Intelligent Control Technology (CEI)*, 560–564. https://doi.org/10.1109/CEI57409.2022.9950068

Traini, L., Di Pompeo, D., Tucci, M., Lin, B., Scalabrino, S., Bavota, G., Lanza, M., Oliveto, R., & Cortellessa, V. (2022). How Software Refactoring Impacts Execution Time [Article]. *ACM Transactions on Software Engineering and Methodology*, *31*(2), 1–23. https://doi.org/10.1145/3485136

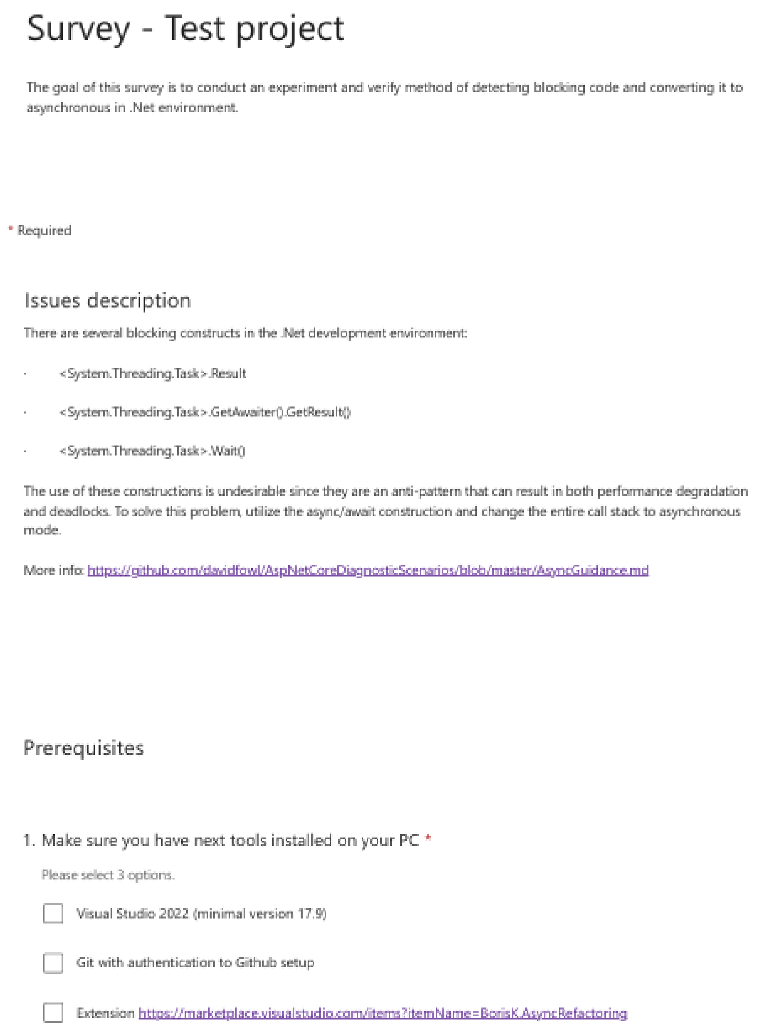
Turcotte, A., Shah, M. D., Aldrich, M. W., & Tip, F. (2022). DrAsync: Identifying and Visualizing Anti-Patterns in Asynchronous JavaScript [Proceeding]. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 774–785. https://doi.org/10.1145/3510003.3510097

Van Rossum, G., & The Python development team. (2017). *The Python Language Reference Release 3.6.0*.

Zhang, Y., Li, L., & Zhang, D. (2020). A survey of concurrency-oriented refactoring. *Concurrent Engineering Research and Applications*, *28*(4), 319–330. https://doi.org/10.1177/1063293X20958932

7. Annexes

Annex 1. Survey of the Experiment for the Test Project



Survey - Test project

The goal of this survey is to conduct an experiment and verify method of detecting blocking code and converting it to asynchronous in .Net environment.

* Required

Issues description

There are several blocking constructs in the .Net development environment:

- <System.Threading.Task>.Result
- <System.Threading.Task>.GetAwaiter().GetResult()
- <System.Threading.Task>.Wait()

The use of these constructions is undesirable since they are an anti-pattern that can result in both performance degradation and deadlocks. To solve this problem, utilize the async/await construction and change the entire call stack to asynchronous mode.

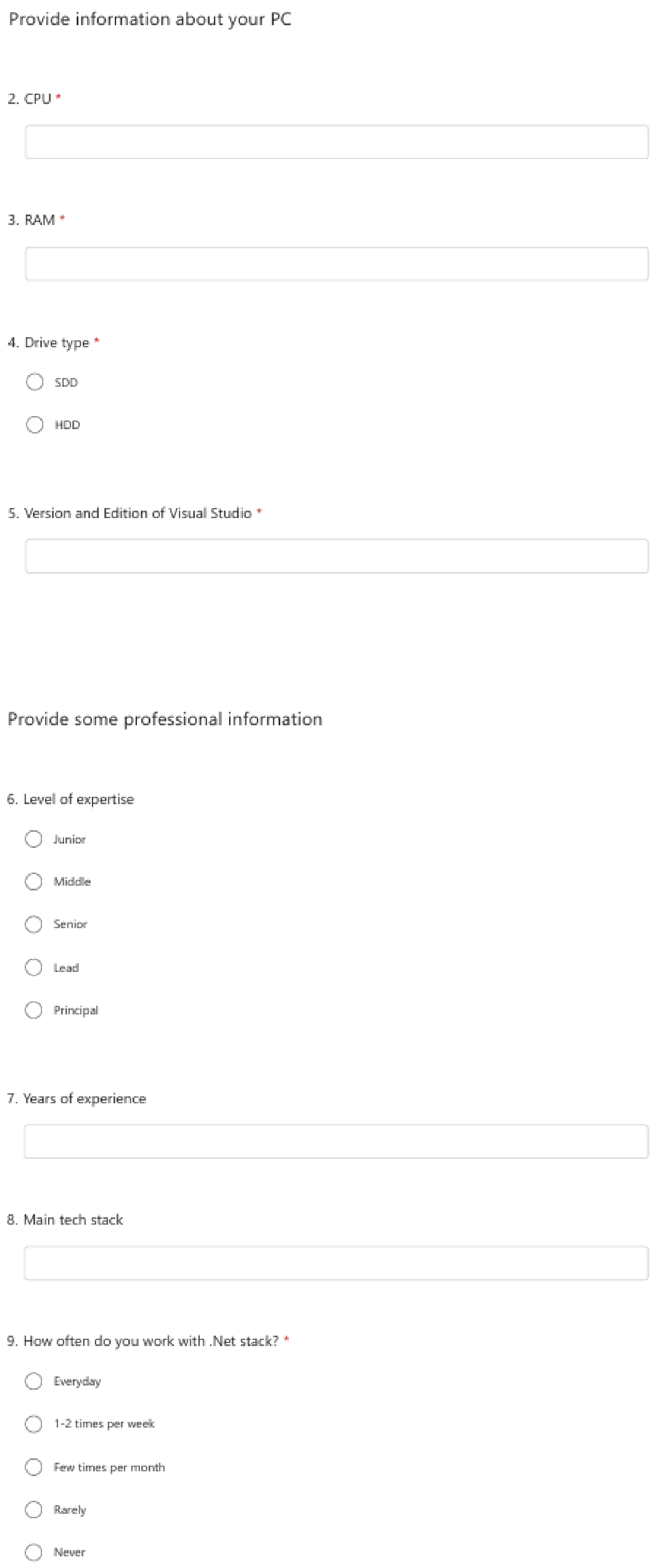
More info: https://github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AsyncGuidance.md

Prerequisites

1. Make sure you have next tools installed on your PC *

Please select 3 options.

- ☐ Visual Studio 2022 (minimal version 17.9)
- ☐ Git with authentication to Github setup
- ☐ Extension https://marketplace.visualstudio.com/items?itemName=BorisK.AsyncRefactoring



Provide information about your PC

2. CPU *

3. RAM *

4. Drive type *

- ○ SDD
- ○ HDD

5. Version and Edition of Visual Studio *

Provide some professional information

6. Level of expertise

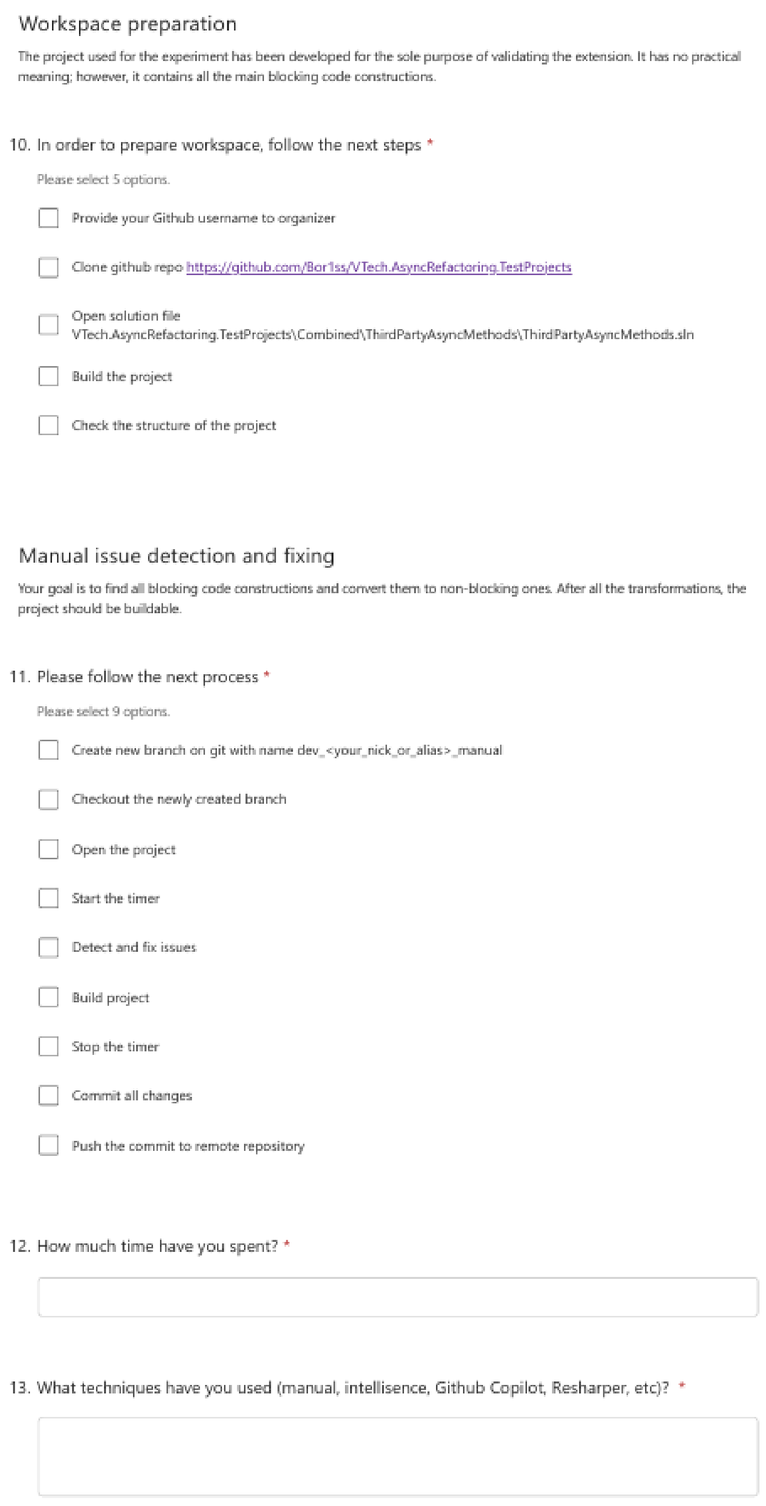
- ○ Junior
- ○ Middle
- ○ Senior
- ○ Lead
- ○ Principal

7. Years of experience

8. Main tech stack

9. How often do you work with .Net stack? *

- ○ Everyday
- ○ 1-2 times per week
- ○ Few times per month
- ○ Rarely
- ○ Never



Workspace preparation

The project used for the experiment has been developed for the sole purpose of validating the extension. It has no practical meaning; however, it contains all the main blocking code constructions.

10. In order to prepare workspace, follow the next steps *

Please select 5 options.

- ☐ Provide your Github username to organizer
- ☐ Clone github repo https://github.com/Bor1ss/VTech.AsyncRefactoring.TestProjects
- ☐ Open solution file VTech.AsyncRefactoring.TestProjects\Combined\ThirdPartyAsyncMethods\ThirdPartyAsyncMethods.sln
- ☐ Build the project
- ☐ Check the structure of the project

Manual issue detection and fixing

Your goal is to find all blocking code constructions and convert them to non-blocking ones. After all the transformations, the project should be buildable.

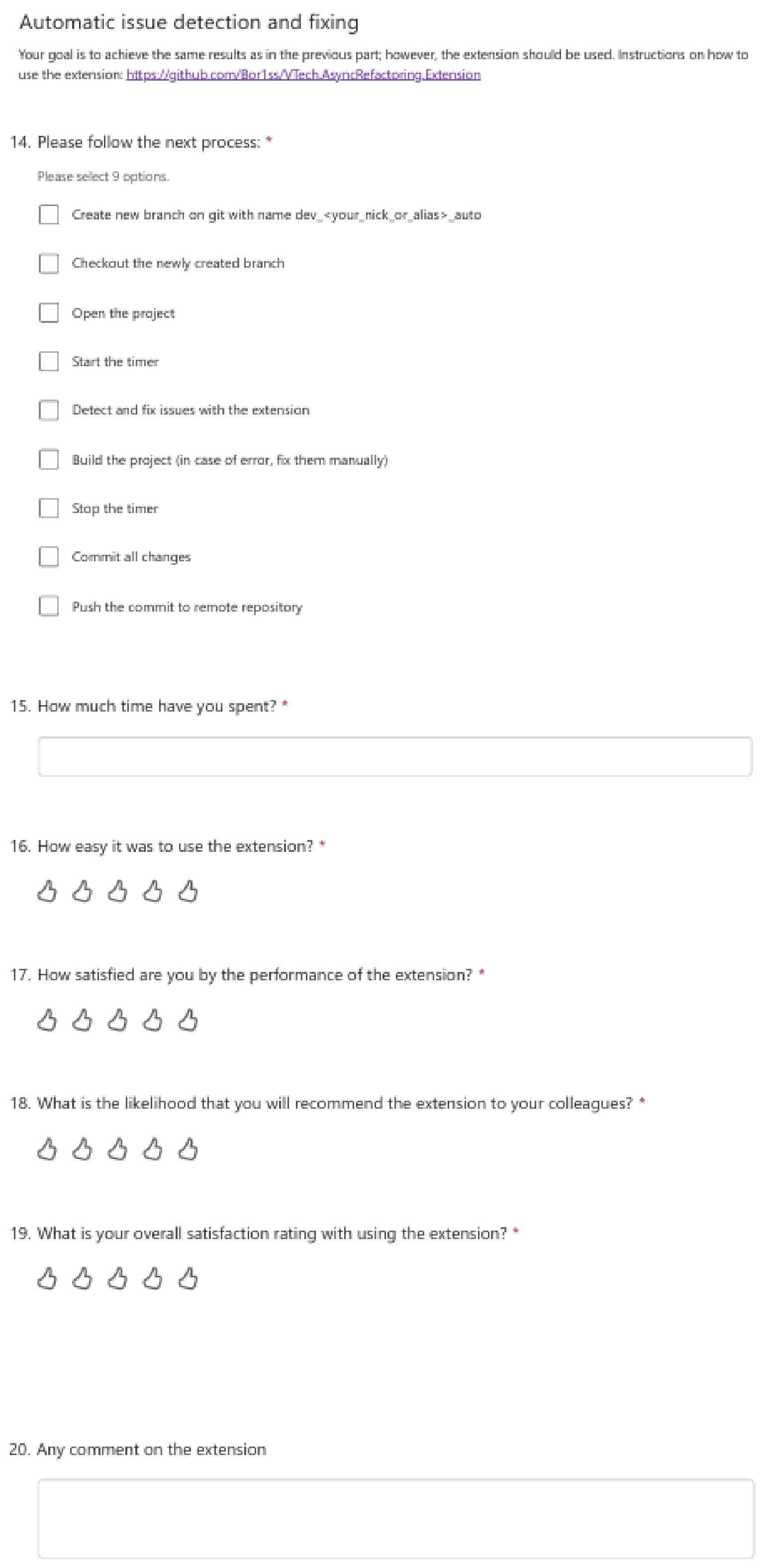
11. Please follow the next process *

Please select 9 options.

- ☐ Create new branch on git with name dev_<your_nick_or_alias>_manual
- ☐ Checkout the newly created branch
- ☐ Open the project
- ☐ Start the timer
- ☐ Detect and fix issues
- ☐ Build project
- ☐ Stop the timer
- ☐ Commit all changes
- ☐ Push the commit to remote repository

12. How much time have you spent? *

13. What techniques have you used (manual, intellisence, Github Copilot, Resharper, etc)? *



Automatic issue detection and fixing

Your goal is to achieve the same results as in the previous part; however, the extension should be used. Instructions on how to use the extension: https://github.com/Bor1ss/VTech.AsyncRefactoring.Extension

14. Please follow the next process: *

Please select 9 options.

- ☐ Create new branch on git with name dev_<your_nick_or_alias>_auto
- ☐ Checkout the newly created branch
- ☐ Open the project
- ☐ Start the timer
- ☐ Detect and fix issues with the extension
- ☐ Build the project (in case of error, fix them manually)
- ☐ Stop the timer
- ☐ Commit all changes
- ☐ Push the commit to remote repository

15. How much time have you spent? *

16. How easy it was to use the extension? *

👍 👍 👍 👍 👍

17. How satisfied are you by the performance of the extension? *

👍 👍 👍 👍 👍

18. What is the likelihood that you will recommend the extension to your colleagues? *

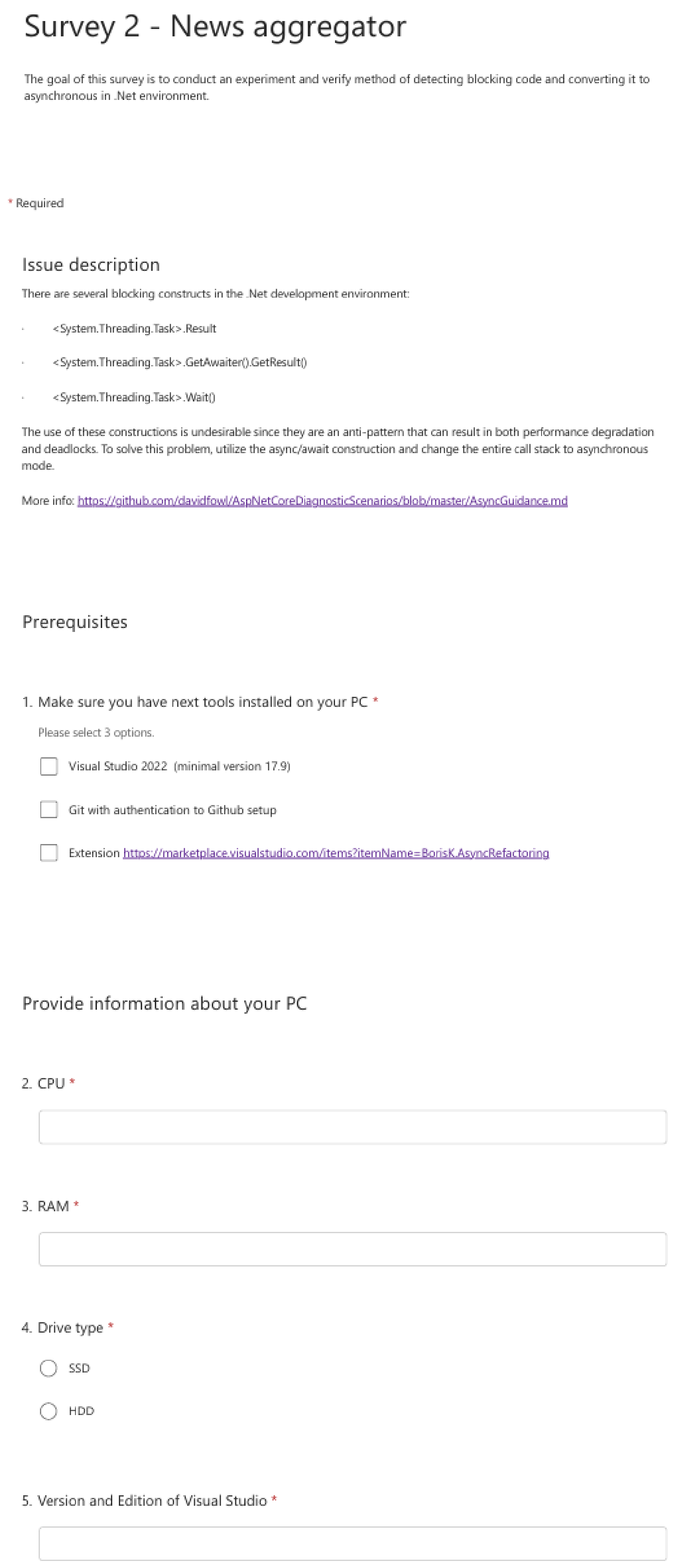
👍 👍 👍 👍 👍

19. What is your overall satisfaction rating with using the extension? *

👍 👍 👍 👍 👍

20. Any comment on the extension

Annex 2. Survey of the Experiment for the *Presscenters.com* Project



Survey 2 - News aggregator

The goal of this survey is to conduct an experiment and verify method of detecting blocking code and converting it to asynchronous in .Net environment.

* Required

Issue description

There are several blocking constructs in the .Net development environment:

- <System.Threading.Task>.Result
- <System.Threading.Task>.GetAwaiter().GetResult()
- <System.Threading.Task>.Wait()

The use of these constructions is undesirable since they are an anti-pattern that can result in both performance degradation and deadlocks. To solve this problem, utilize the async/await construction and change the entire call stack to asynchronous mode.

More info: https://github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AsyncGuidance.md

Prerequisites

1. Make sure you have next tools installed on your PC *

Please select 3 options.

☐ Visual Studio 2022 (minimal version 17.9)

☐ Git with authentication to Github setup

☐ Extension https://marketplace.visualstudio.com/items?itemName=BorisK.AsyncRefactoring

Provide information about your PC

2. CPU *

3. RAM *

4. Drive type *

○ SSD

○ HDD

5. Version and Edition of Visual Studio *

Provide some professional information

6. Level of expertise

◯ Junior

◯ Middle

◯ Senior

◯ Lead

◯ Principal

7. Years of experience

8. Main tech stack

9. How often do you work with .Net stack? *

◯ Everyday

◯ 1-2 times per week

◯ Few times per month

◯ Rarely

◯ Never

Workspace preparation

The project used for the experiment is News aggregator for the press releases of the Bulgarian government sites. This project has great number of constructions that utilize blocking operations

10. In order to prepare workspace, follow the next steps *

Please select 5 options.

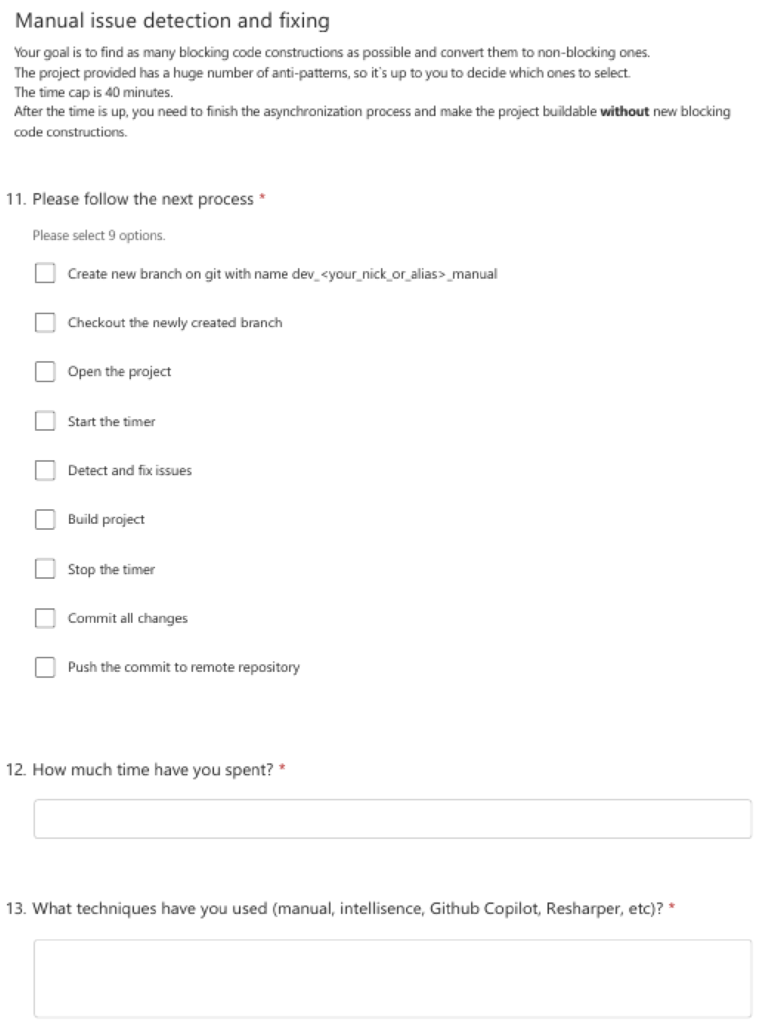
☐ Provide your Github username to organizer

☐ Clone github repo https://github.com/Bor1ss/VTech.AsyncRefactoring.Test.PressCenters.com

☐ Open solution file VTech.AsyncRefactoring.Test.PressCenters.com/src/PressCenters.sln

☐ Build the project

☐ Check the structure of the project



Manual issue detection and fixing

Your goal is to find as many blocking code constructions as possible and convert them to non-blocking ones.
The project provided has a huge number of anti-patterns, so it`s up to you to decide which ones to select.
The time cap is 40 minutes.
After the time is up, you need to finish the asynchronization process and make the project buildable **without** new blocking code constructions.

11. Please follow the next process *

Please select 9 options.

☐ Create new branch on git with name dev_<your_nick_or_alias>_manual

☐ Checkout the newly created branch

☐ Open the project

☐ Start the timer

☐ Detect and fix issues

☐ Build project

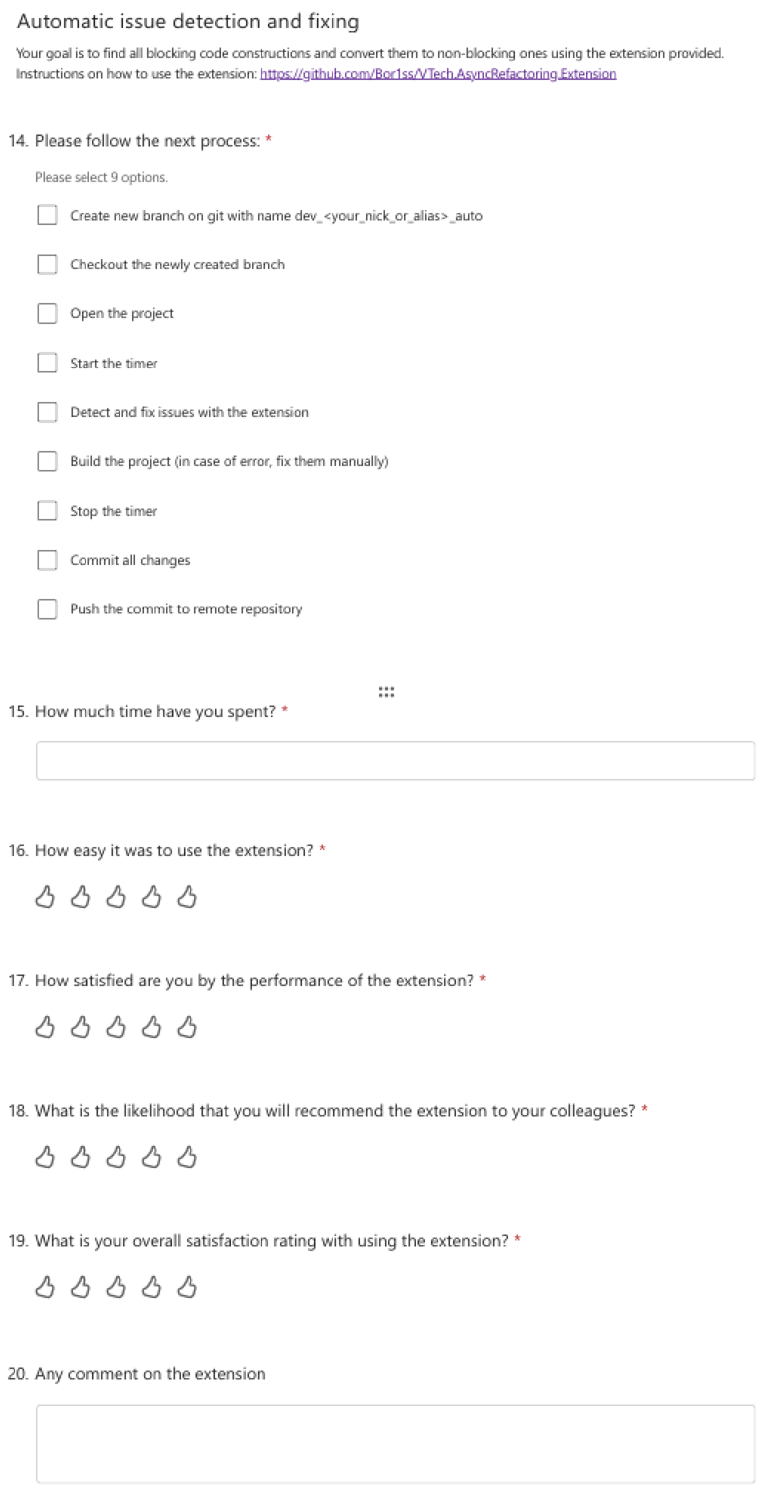
☐ Stop the timer

☐ Commit all changes

☐ Push the commit to remote repository

12. How much time have you spent? *

13. What techniques have you used (manual, intellisence, Github Copilot, Resharper, etc)? *



Automatic issue detection and fixing

Your goal is to find all blocking code constructions and convert them to non-blocking ones using the extension provided.
Instructions on how to use the extension: https://github.com/Bor1ss/VTech.AsyncRefactoring.Extension

14. Please follow the next process: *

Please select 9 options.

- ☐ Create new branch on git with name dev_<your_nick_or_alias>_auto
- ☐ Checkout the newly created branch
- ☐ Open the project
- ☐ Start the timer
- ☐ Detect and fix issues with the extension
- ☐ Build the project (in case of error, fix them manually)
- ☐ Stop the timer
- ☐ Commit all changes
- ☐ Push the commit to remote repository

15. How much time have you spent? *

16. How easy it was to use the extension? *

👍 👍 👍 👍 👍

17. How satisfied are you by the performance of the extension? *

👍 👍 👍 👍 👍

18. What is the likelihood that you will recommend the extension to your colleagues? *

👍 👍 👍 👍 👍

19. What is your overall satisfaction rating with using the extension? *

👍 👍 👍 👍 👍

20. Any comment on the extension