FACULTY OF ENGINEERING OF THE UNIVERSITY OF PORTO

Towards Live Refactoring to Patterns

Carlos Eduardo da Nova Duarte

U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Master in Informatics and Computing Engineering

Supervisor: Prof. Ademar Manuel Teixeira de Aguiar

Company Supervisor: José António Silva

July 25, 2022

Towards Live Refactoring to Patterns

Carlos Eduardo da Nova Duarte

Master in Informatics and Computing Engineering

July 25, 2022

Abstract

Refactoring enhances the design of existing code while maintaining its original functionality, improving its quality while decreasing maintenance costs. Practitioners do not always refactor despite the advantages, especially if unaware of refactoring opportunities. Moreover, if the developer is not comfortable with the process, it can cause considerable slowdowns and dirty code, ultimately decreasing productivity.

Live Software Development is an idea that aims to improve the liveness of SDLC[1] activities, for example, by making the development environment more informative, responsive, and immediate. Live Refactoring, concretely, incentivizes practitioners to refactor more often by automating the environment's refactoring capabilities, providing feedback on quality attribute enhancements, code previews, and automatic suggestions.

Currently, IDEs[2] only support the most popular refactorings. Specific organizations or frameworks might have their own refactoring needs and processes, which can also benefit from the liveness. While IDEs usually provide refactoring extensibility APIs[3], creating a new refactoring plugin has a very steep learning curve, potentially translating into significant time investments. Additionally, multiple refactorings might have patterns in common, and replicating the same behaviors over different refactorings is counterproductive and may introduce bugs.

This dissertation aims to improve development environments by making the refactoring plugin creation processes more automatic and reusable. Refactoring plugins are built by selecting the desired, auto-generated behavior modules featuring Live Refactoring capabilities. The user can focus on orchestrating modules instead of writing low-value boilerplate code. An internal DSL[4] was created to allow the development of said tools, which was validated by performing a set of controlled experiments with programmers of varying maturity.

Ultimately, this work intends to improve the refactoring plugin creation experience and assess if the proposed solution can help novice and experienced practitioners enhance their development experience and productivity, leading to fewer bugs, time savings, and better code quality.

[1]Software Development Life-Cycle
[2]Integrated Development Environment
[3]Application Programming Interface
[4]Domain-Specific Language

Resumo

O processo de Refactoring aperfeiçoa o design de código já existente mantendo a sua funcionalidade original, resultando numa melhoria da sua qualidade e, ao mesmo tempo, numa diminuição dos custos de manutenção. Apesar das vantagens, os programadores nem sempre aplicam refactorings, especialmente se não tiverem noção das oportunidades existentes para o fazerem. Para além disso, se o programador não estiver confortável com o processo, este processo pode causar atrasos significativos e código "sujo", que, ultimamente, resulta em quedas de produtividade.

O Live Software Development é uma ideia que pretende melhorar a vivacidade das atividades do Ciclo de Vida de Desenvolvimento de Software, contribuindo para tornar os ambientes de desenvolvimento mais informativos, responsivos e imediatos. O Live Refactoring, concretamente, incentiva os programadores a aplicarem refactorings mais frequentemente através da automatização das capacidades de refactoring do ambiente, providenciando feedback na melhoria de atributos de qualidade, pré-visualização de código, e sugestões automáticas.

Atualmente, os IDEs [5] apenas suportam os refactorings mais populares. Organizações específicas ou frameworks podem ter as suas próprias necessidades e processos de refactoring, que também poderiam beneficiar de vivacidade acrescida. Apesar destes IDEs normalmente oferecerem APIs [6] de extensibilidade de refactorings, o processo de criação de um plugin de refactoring possui uma curva de aprendizagem muito acentuada, o que pode traduzir-se em investimentos de tempo significativos. Adicionalmente, vários refactorings podem ter padrões em comum, sendo que replicar o mesmo comportamento ao longo de vários refactorings é contraproducente e pode introduzir erros.

Esta dissertação tem em vista a melhoria dos ambientes de desenvolvimento de forma que o processo de criação de plugins de refactoring se torne mais automático e reutilizável. Os plugins são construídas através da seleção dos módulos comportamentais desejados, que são automaticamente gerados, possuindo características associadas ao Live Refactoring. Desta forma, o utilizador pode-se focar na orquestração de módulos ao invés de escrever código de base com baixo valor. Uma DSL [7] interna foi criada para permitir o desenvolvimento desses plugins, a qual foi validada através da realização de várias experiências controladas com programadores de variados níveis de maturidade.

Por fim, este trabalho pretende melhorar a experiência de criação de plugins de refactoring e avaliar se a solução proposta é capaz de melhorar a experiência de desenvolvimento e produtividade tanto de programadores novatos quanto experientes, conduzindo a menos erros, poupanças de tempo, e melhor qualidade de código.

[5] Ambiente de Desenvolvimento Integrado
[6] Interface de Programação de Aplicação
[7] Linguagem de Domínio Específico

Acknowledgements

First, I would like to thank my supervisor, professor Ademar Aguiar. It really was a pleasure working with you to create this dissertation, which actually began half a year before the start of this academic year. Since our initial brainstorming meetings you were always supportive, enlightening, and reachable. Moreover, your cheerfulness made it much easier to finish this project. Thank you for trusting in me, for welcoming me as your supervisee, and for all you have taught me.

My acknowledgements also go to DevScope, the company that welcomed and supported this idea. More concretely, I would like to thank José António Silva for trusting in me and giving me the chance to work at such a great company. Additionally, I want to thank David Mota for all the support, knowledge, and discussions on life, the universe and everything.

I am also tremendously thankful to my parents, Carlos and Margarida, and my two closest aunts, Helena and Isabel. You have raised me and supported me since the day I was born, during good and bad times. Without you, this would not have been possible. I am truly privileged to have you by my side. From the bottom of my heart, thank you for your love. And Lena, never lose your faith and hope. As St. John of the Cross once said: "The endurance of darkness is the preparation for great light".

To my friends Zé and Bárbara: thank you for never letting our friendship fade away. You have been my closest friends since we met at University of Minho in 2016, and I am forever grateful to have you in my life. You are truly special.

I must also thank my dear friend "Broas". You are not here, but the truth is that you have never left. Your will and character have helped shape me into the person I am today. I plan to keep our promise. Thank you for your friendship. May your light keep shining over us all, and may God bless you.

To all others who have helped me along the way, thank you.

Carlos Eduardo da Nova Duarte

To my parents Carlos and Margarida,
and to my aunts Helena and Isabel.

In memory of my dear friend
André "Broas" Reis.

Contents

List of Figures

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CI/CD	Continuous Integration/Continuous Deployment
DSL	Domain-Specific Language
IDE	Integrated Development Environment
LiveSD	Live Software Development
MDSD	Model-Driven Software Development
MDSE	Model-Driven Software Engineering
MOF	Meta-Object Facility
SDK	Software Development Kit
SDLC	Software Development Life-Cycle
UML	Unified Modelling Language
RPCL	Refactoring Plugin Creation Language

Chapter 1

Introduction

There is an old saying that the only constant in life is change. In the software engineering world, this is ever more true: market needs vary over time, competitors create new products that put other companies' competitive advantages at stake, organizations restructure themselves, time and cost budgets are surpassed, and teams' communication abilities and morale differ considerably [40]. These events affect the entire software development life-cycle activities: changes in requirements might lead to flawed designs, implementations that solve the wrong problem, and tests that evaluate scenarios that have drifted from the project's reality.

Most software projects fail to meet time and budget constraints [39]. While projects get increasingly larger and more complex, developers may be tempted to rush the implementation of certain features or implement small hacks in the project's architecture to meet deadlines. As code quality worsens, the team's productivity falls, making it even harder to meet the constraints mentioned above [52]. This snowball effect can make it very hard to maintain a software project.

1.1 Context

In order to minimize this issue, refactoring is often used as a means to improve the design of existing code while maintaining its original functionality [33]. Therefore, it is in developers' best interests to refactor as often as needed. Design patterns - battle-tested software designs that solve common and recurring problems - are also a way to improve software maintainability and facilitate its evolution [35]. However, they are not a silver bullet, and implementing them from early design stages might not be as beneficial as one may think. As requirements change with time, the added complexity necessary to increase flexibility might become unnecessary [19].

1.2 Motivation

Refactorings vary in complexity. While some are simple, others require multiple steps to be applied. These steps can repeat themselves along multiple refactorings. In this case, it can be said that they represent a pattern. For instance, a refactoring that operates over the iterable variable of a for-each loop and converts it to a list is comparable to another refactoring that does the same thing but converts the variable to a string.

One example of more complex refactorings is those that aim to implement a design pattern. Refactoring to a design pattern is not as simple as refactoring a simple code smell, requiring multiple steps to be completed until the pattern is implemented. However, due to the increased complexity of these refactorings, it may not always be possible to apply them thoroughly. Nevertheless, their partial implementation may still improve software design. Kerievsky [45] argues that refactoring to, towards, and away from patterns is a better approach than implementing said patterns straight away, which often leads to over-engineering. This further highlights the complexity of refactoring code to follow a design pattern and, ultimately, the variety and intricacies that the process involves.

Research shows that automatic refactoring suggestions is one of the most requested features by developers for development environments [62]. This is a characteristic of Live Environments [14], which provide, among other features, more immediate and automatic feedback to developers without needing to compile the source code. Modern development environments such as Visual Studio and IntelliJ are powered by automatic refactoring capabilities, live feedback, and relevant suggestions. Development environments provide users with a robust catalog of the most common refactorings, such as those described by Fowler [33].

These tools make it easier for developers to improve their software while minimizing the required effort. Having access to these features should motivate them to refactor often. It is advantageous for practitioners to use the Live Refactoring capabilities of the IDEs they use for applying both simple and complex refactorings.

1.3 Problem

Unfortunately, IDEs provide users with a limited set of refactorings, which are usually the most popular ones. These are also often simple ones, not supporting more complex transformations such as applying a design pattern. This is not good enough.

Additionally, organizations may have identified patterns in their frameworks or internal tooling that could benefit from the Live Refactoring features offered by IDEs. Suppose these environments do not offer the refactorings they need. To benefit from such code transformations, practitioners may either refactor code by hand or build new refactoring tools that cater to their needs.

It is plausible to assume that if the refactoring tool creation process is not accessible, then practitioners are more likely to refactor manually or not refactor at all. The decision will depend on how much change affects the project, as mentioned at the beginning of this chapter.

While environments provide practitioners with APIs to create refactoring tools of their own, the creation process is often not trivial, requiring knowledge of the platform's compiler SDK, and syntactic and semantic analysis topics. Consequently, smaller organizations might not have the resources to build their custom refactoring tools, requiring more experienced practitioners knowledgeable of the process's intricacies.

Ultimately, the main problem is that it is hard to create refactoring tools in a fast and accessible manner while providing users with enough power to create complex transformations, potentially detracting practitioners from refactoring code.

1.4 Objectives and Expected Results

By easing and adding flexibility to the refactoring tool creation process, the goal is that developers are more incentivized to create their tools, thus leading them to refactor more often. By doing so, code quality should increase, making the project more robust and increasing its maintainability over time. Moreover, using the liveness features of modern development environments, the created refactoring tools would also contribute to making the refactoring experience more accessible.

This dissertation will begin by presenting a literature review aiming to understand current practices and theoretical models of refactoring tool creation.

Then, it will explore implementing a tool for creating them in an accessible and straightforward manner, using the Live Refactoring features offered by modern IDEs.

Finally, an expert assessment and survey will be made to assess the implemented tool's viability, quality, and defects. A survey will be presented to assess how effective this new refactoring tool creation process is and how much developers benefit from using it.

1.5 Structure

Chapter 1 describes the context and motivation of this research and the main research problem, objectives, structure, and work plan. Chapter 2 addresses the background information required to understand the dissertation better, and Chapter 3 presents the state of the art of refactoring tool creation. Chapter 4 addresses the problem statement in-depth, as well as a brief description of the proposed solution. Chapter 5 goes over the implementation details, and Chapter 6 presents and discusses the validation process and results. Finally, Chapter 7 presents the overall conclusions and future work.

Chapter 2

Background

In order to understand the state of the art of refactoring creation tools and techniques, it is necessary to understand the building blocks that support these technologies. From refactoring and design patterns to code generation and meta-programming, the creation of refactoring tools is certainly not trivial.

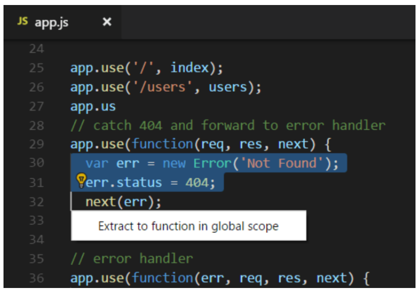
2.1 Refactoring

Refactoring is a beneficial technique to ensure that software can be maintained throughout its lifetime by iteratively applying structural changes that improve its core architecture without modifying its behavior. In fact, as developers keep applying cumulative refactorings, the number of software defects decreases proportionately [66], which consequently makes software easier to maintain [65] [53] [57] [45].

Refactoring: Improving the design of existing code while maintaining its original functionality.

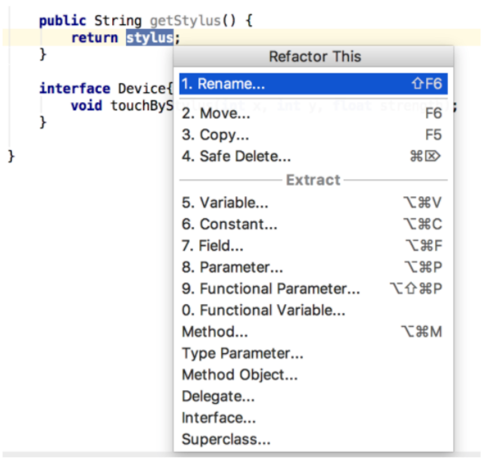
Code may be refactored, among other reasons, in response to changing requirements that make it necessary to reorganize the software's internal architecture or eliminate code smells [72]. The latter is especially important, as code inevitably starts to rot. The more code smells accumulate, the harder it is to make changes to the existing code base, which inevitably leads to steep decreases in productivity and an unhappy team [52]. While it is no silver bullet, the process makes software easier to understand, helps identify bugs in source code, and speeds up time spent programming [33]. It also contributes to developers getting a better grasp on how the code works, which in turn minimizes the need to consult with other developers, increasing the autonomy of the practitioner [45].

Modern development environments such as Visual Studio and IntelliJ provide tools that aid the refactoring process, as shown in Figures 2.1 and 2.2. These IDEs can analyze source code while the user is programming, automatically generating the refactored source code, alerting the user, previewing transformations, and applying the refactoring with a single click [8] [1].



```
app.js

app.use('/', index);
app.use('/users', users);
app.us
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);

// error handler
app.use(function(err, req, res, next) {
```

Figure 2.1: A Live Refactoring suggestion in Visual Studio Code. [8]



```
    public String getStylus() {
        return stylus;
    }

    interface Device{
        void touchByS
    }

}
```

Figure 2.2: IntelliJ's refactoring GUI. [1]

This activity contributes very positively to the reduction of software defects [66], and may also make teams more productive [56]. In a world where Agile methodologies are cornerstones of modern software development, refactoring becomes even more essential to ensure the success of software projects.

2.2 Design Patterns

Design patterns help solve the challenge of designing good, reusable software. Since the dawn of object-oriented design, developers have identified design solutions to recurring problems. Like cooking recipes, design patterns provide tested designs proven to work for certain use cases, which can be reused and shared among practitioners [35]. The classic set of patterns developed by the Gang of Four [35] is shown in Figure 2.3.

Design Pattern: A general, well-tested solution to a common problem.

Useful to both experienced and novice practitioners [16], implementing design patterns can bring various benefits to software projects. One is improving communication between software designers, as patterns allow for a common understanding when discussing implementation details. Other advantages are they can better preserve software adaptability during maintenance changes, ease new team developers get into the project's architecture, contribute to clearer software architecture documentation, and encourage best development practices [20] [16] [63].

For all benefits that design patterns provide, there are also some drawbacks. Not all patterns are easy to grasp, which might lead to developers implementing half-baked solutions that do more harm than good [80]. Moreover, enough experience is needed for a practitioner to write compelling and beneficial patterns, requiring them to get a hold of enough examples to recognize similarities

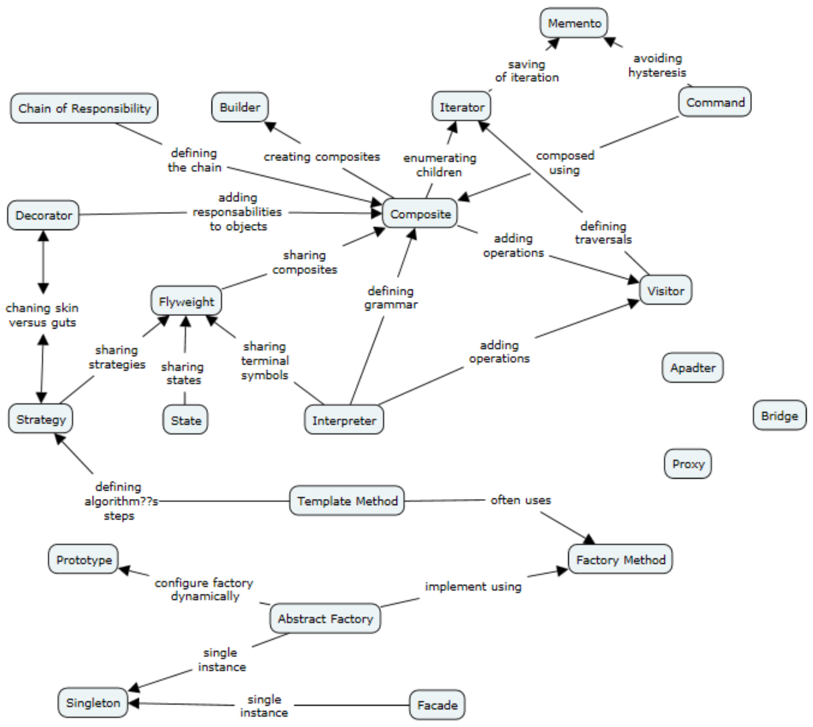


Figure 2.3: Design patterns proposed by Gamma et al.. [35] [4]

in code [16]. Finally, even if patterns often help improve program maintenance globally, that is not always the case. Sometimes the solution can be implemented much simpler rather than jumping straight to implementing a pattern [45], even if flexibility caused by changing requirements may be lost [63].

Given all this, it is fundamental that programmers have studied design patterns well enough to know how and when to apply them. A practitioner lacking this knowledge might look at a better, more maintainable design built with a pattern in mind and still prefer the old solution [45], with familiarity probably being an essential factor for justifying this behavior.

2.3 Refactoring to Patterns

Refactorings and design patterns ease software maintenance, allowing designs to evolve better as requirements change. While some might associate design patterns more with the overall architecture of the software, design patterns are also about the code. Therefore, these techniques go hand in hand during the software development process [45].

Practitioners might apply patterns in two distinct ways. The first one is to plan and, before development starts, think about which patterns make more sense to implement, both at the moment and for the foreseeable future. This is refactoring: code structures being adapted to improve maintainability in the long term, even if not replacing existing code. The second one is refactoring existing lousy code to clean code, resulting in implementing a pattern that improves overall design without altering functionality [45].

Refactoring Pattern: A behavior that is shared across multiple refactorings. A pattern may be atomic or constituted by a sequence of more complex patterns. For example, finding all variables of a given type is a behavior that makes up many refactoring algorithms.

Refactoring to Patterns: A technique for improving software design via implementing design patterns by sequentially applying refactoring patterns as necessary, instead of implementing them from the initial architecture stages.

Kerievsky [45] points out a few problems with the first alternative. First, it is risky to implement patterns to improve future design requirements: if the product does not need those adaptations, time and money are wasted. Furthermore, as previously discussed, change is always happening, so developers cannot afford to waste time unnecessarily. Additionally, code that gets added and is not used may never be removed, as developers new to the system might not be sure if that piece of code will or will not break other core system functionalities. These novice developers may also get confused about why those structures are there and not understand their role in the overall system, which worsens design since they do not play any role.

The author also argues that developers should focus their time and effort on building the system without applying patterns. If patterns are needed, they should primarily result from a system refactoring concerned with preparing the system for new features or improving code design. It is, then, vital to know how and when to refactor towards and away from them. A particular design pattern may not improve the solution as much as the user hoped, and applying other techniques might be more helpful.

2.4 Meta-Programming

In order to implement a design pattern, it is necessary to assess how existing code will change and which code blocks must be inserted into the program. Similarly, refactoring may require existing code to be modified and new code to be introduced. Both of these activities require detecting code structures and modifying existing files. Refactoring tools must be able to perform these activities automatically.

Meta-programs are programs that can analyze, interact and create other programs [46]. Meta-programming techniques have been long used in formal logic programming, but throughout time have broadened their application spectrum, for instance, to generating compilers, creating refactorings and design patterns, building code analyzers, and scripting [23]. These programs are usually defined in meta-languages, which in turn represent meta-models.

2.4.1 Modelling and Meta-Modelling

Overbeek [61] describes models as representations of things with associated meaning. The author explains that, in order to represent meaning, a structured language must be used. Those languages are, naturally, composed of a syntactic component, which is the concrete notation of the language, and a semantic component, which explains the syntactic components. Meta-languages, as they are usually named, are defined by Czarnecki and Eisenecker as languages that can analyze and describe other languages [22]. Programs that result from these languages are known as meta-programs.

Meta-models are models that represent modeling languages and are one abstraction level above those same models. It is possible to keep increasing abstraction levels: for instance, a model that represents a meta-model is called a meta-meta-model. A representation of the meta-model structure is available in Figure 2.4.

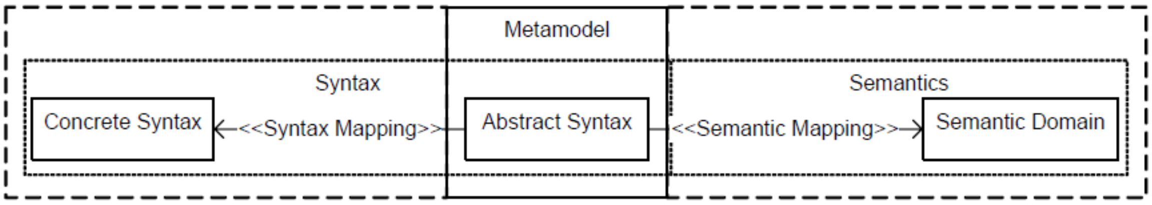


Figure 2.4: The different components that make up a meta-model. [61]

One of the most well-known meta-modeling standards is the Meta-Object Facility. Built by Object Management Group in 1989, MOF was intended to make it easier, more portable, less expensive, and less complex to develop software by basing it on a standard interface [61]. The MOF architecture can be visualized in Figure 2.5.

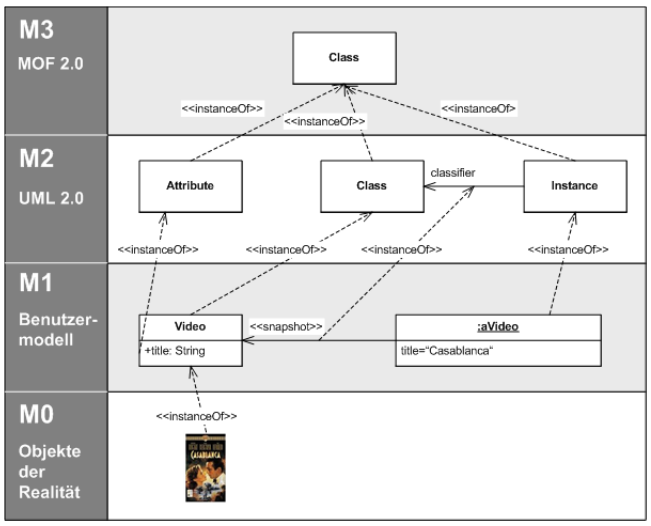


Figure 2.5: The architecture of the Meta-Object Facility standard. [12]

MOF's architecture comprises four hierarchically organized layers, from M0 to M3. M0, the base layer, represents real-world objects. In turn, the M1 layer represents a model of the real object. The M2 layer is a model that describes M1 models: a meta-model. Finally, the M3 layer is a model that allows for manipulating meta-models, naturally, meta-meta-models.

Perhaps the most popular meta-model (M2) built against the MOF specification is the Unified Modelling Language meta-model, which is the model used to describe the widely-used UML models.

2.4.2 Reflection

Reflection is a meta-programming technique for providing practitioners access to the code they are running, allowing them to access and execute its code at run time [37]. This is achieved by allowing developers to interact with the program's meta-model. Some practical applications of reflection are debuggers and system optimization.

On its own, meta-programming is inherently reflective, as it allows for creating software that can manipulate other software [23]. C# is an example of a language that supports reflection [83]. This technology makes it possible, for instance, to build new types at run time, examine and instantiate assembly types, and access attributes from a project's metadata.

2.4.3 Model-Driven Software Development

Models provide abstraction from more complex structures. However, when creating a program, the computer does not directly understand these abstractions: they need to be translated into a language. For instance, a computer does not directly understand the C language: it must be transformed into a lower abstraction language, Assembly, which in turn is translated into an even-lower language, which is binary code. Only then can the computer understand the program's instructions and act accordingly.

No sane developer would create a simple program using binary code, much less a complex one. To do that, programmers use abstractions. For instance, one of the reasons Python became the fastest-growing programming language in the world is its very high levels of abstraction [73]. When developers can focus on the task at hand and not on manually allocating memory or doing garbage collection, chances are they will make much more productive use of their time.

UML models also allow developers to understand the complexity of a system in a much more intuitive way, as they represent functionality at an abstraction level high enough to allow practitioners to reason about its components quickly. Building software by creating abstract models that are then generated in an automatic or semi-automatic fashion is the basis of what is known as Model-Driven Software Development [49].

Models built with UML are often used to document software architectures. In MDSD, according to Stahl and Völter [74] models are considered code and not documentation, given that code will be generated from said models. By following such an approach when developing software, the authors defend that productivity may significantly increase as programming becomes more automated. Moreover, domain experts can understand the models, improving requirements elicitation and specification processes, increasing the chances that software is implemented strictly as the experts requested. To put MDSD into practice, the authors point out some pre-requisites such as compilers, transforming the models into actual code, and a language for specifying the transformations required to pass from a model representation to concrete code. Finally, to formulate said models, a modeling language that is used to pass instructions to the computer and describe programs, having only as many features as necessary to describe the program's domain, which is small and focused, is needed. This is the definition of a Domain-Specific Language [32]. These requirements, among others, make possible the implementation of an MDSD approach.

2.5 Code Generation

With software project requirements changing at an ever-faster rate, developers need to spend more time and resources building new features and refactoring code to keep stakeholders happy and the project in a maintainable state. The more time developers spend developing a particular feature, the more advantage competitors have in building a better, more robust solution. Therefore, it is essential to reduce developers' time on programming, focusing on the rationale behind design solutions and implementing the most critical code structures.

Code generators are a way to automatically synthesize software structures, which is particularly useful for ones that end up getting repeated throughout programs. Duplicate code is a code smell that should be refactored: if there is the need to change a specific aspect of that code block, all instances must be changed, which may introduce errors; also, the duplicated code might do the same but have minimal differences that are hard to catch, and therefore may not be considered duplicate code by the developer when, in fact, they are [3].

Roslyn [82] is the .NET compiler platform SDK, which exposes the tremendous amount of information that the compiler has on program structures. It supports generating source code by writing the code to be generated in C# and then writing annotations that reference said code so that it can be injected on compile time into the program. The code generation process, as well as the structure of a source generator, are described in Figures 2.6 and 2.7.

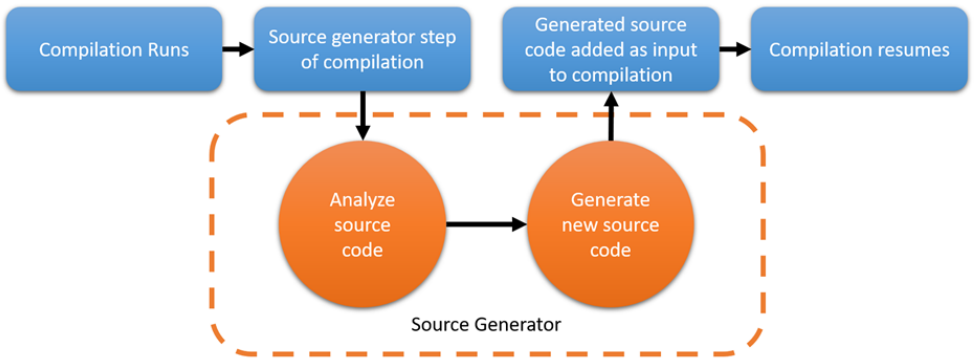
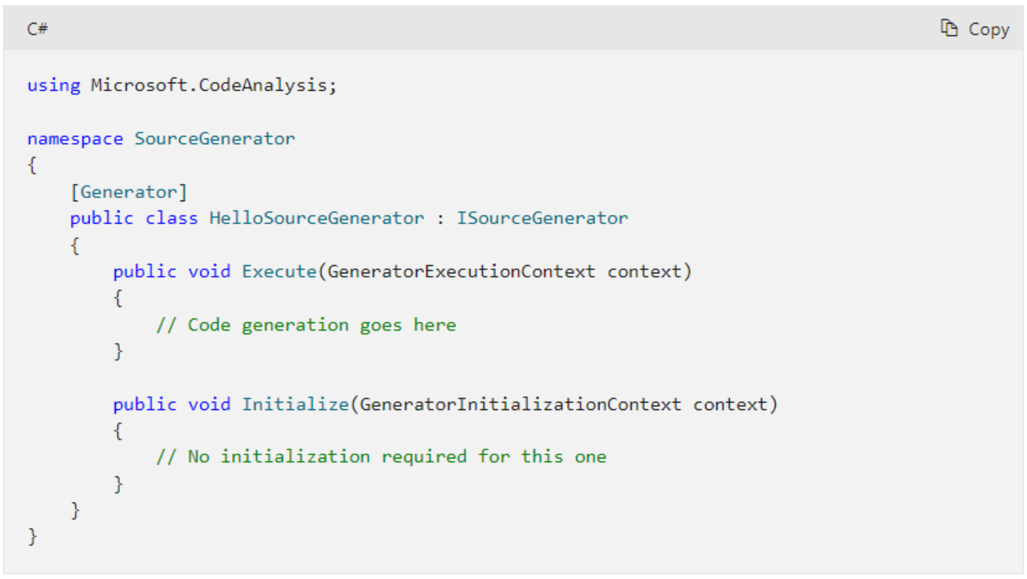


Figure 2.6: The process of injecting source code by Roslyn's compiler. [84]

Roslyn provides several tools that use the code generation feature, such as code analyzers, code fixes, and refactorings. Analyzers and code fixes detect code that should be changed and



```
using Microsoft.CodeAnalysis;

namespace SourceGenerator
{
    [Generator]
    public class HelloSourceGenerator : ISourceGenerator
    {
        public void Execute(GeneratorExecutionContext context)
        {
            // Code generation goes here
        }

        public void Initialize(GeneratorInitializationContext context)
        {
            // No initialization required for this one
        }
    }
}
```

Figure 2.7: Boilerplate code of a .NET source generator. [84]

transform it to improve its design, respectively. On the other hand, refactorings combine both features but provide the information contextually. That means they will only be suggested when the users interact with code that could benefit from improvements. In contrast, analyzers and code fixes parse all project files and suggest transformations to the user in a dedicated panel. These APIs are tremendously powerful and allow practitioners to make the most of the .NET compiler.

One of the most popular applications of code generators has to do with web development. Naik and Shivalingaiah [58] describe the first iteration of the web as a system that provided access to documents that could be linked to one another and visualized in a read-only fashion, with little interactivity and dynamism. Later, subsequent web versions allowed for user interaction, with social media as the primordial example. Nowadays, some websites still present users with static content that does not allow interaction. For situations where there is much static content developers wish to make available on the web, it is not resource-efficient to manually build web pages for those documents.

For this use case, Static Site Generators are one possible solution. Tools such as Jekyll [6] allow users to describe documents in Markdown, a markup language, apply a layout corresponding to the document type, and generate the web page accordingly. This allows users to save time by focusing on writing the content rather than building the web page.

Code generation techniques can improve the software in multiple ways, such as better performance, reduced code volume, and platform compatibility [74].

Code generators are meta-programs. These create new code by transforming the model of existing code received as input [74].

2.6 Live Software Development

The Software Development Life-Cycle comprises many requirements analysis, design, implementation, testing, and maintenance activities. Agile methodologies have long attempted to shorten the feedback loops of the SDLC by providing more and faster feedback to developers [14]. Not only that, but it also focuses on allowing for better comprehension of the system, leading to faster evolution and maintenance processes and aided by techniques such as software visualization

Live Software Development is an idea that aims to bring more liveness to all activities of the SDLC through more immediate and automatic feedback, ultimately intending to ease software evolution and maintenance processes [14]. Practitioners are human and naturally make mistakes and may not identify problems as they arise. Aguiar et al. [14] suggest providing more meaningful information to each SDLC phase through the engineer's environment to lessen these issues.

2.6.1 Live Requirements

Requirements Engineering is one field that may not immediately resonate with live environments. While there are tools for managing and tracking requirements, these are unaware of events in other SDLC phases, such as failing tests and CI/CD pipelines. These tools also require engineers to manually assess the validity and satisfiability of requirements, as they are not connected to the implementation structures, nor is there a way to automatically identify when something has gone wrong [25].

Duarte [25] provides an example of how the requirements activity can be better integrated into development environments through feedback propagation to and from this SDLC phase. This is achieved by defining requirements so that tests can be automatically generated and signal whether the requirements are satisfied or not, communicating said feedback into the development environment.

Such initiatives could significantly improve communication with stakeholders on system status and let developers understand how code implementations affect requirements more quickly and automatically.

2.6.2 Live Programming

Live development environments should integrate a set of tools so that, during development, engineers can get information both automatically and on-demand. This can directly benefit the programming activity of the SDLC in the form of Live Programming, which currently already benefits from tools such as code auto-completion engines, debuggers, always-on syntax highlighting, and syntactic analyzers.

Live Programming, as described by Tanimoto [75], is characterized by having the program currently being worked on constantly running in the environment, which provides valuable feedback and gives suggestions to the programmer. Current IDEs already implement many Live Programming-inspired features, such as the ones mentioned in the previous paragraph.

For instance, some research on LiveSD has explored novel ways for visualizing software structures through a city metaphor. In it, each building represents a component of the overall system that is the city [15] [51], as seen in Figures 2.8 and 2.9. These allow for visualizing and interacting with source code and cloud instances of popular services like Amazon EC2 [9]. Examples are shown in Figure 2.8 and Figure 2.9.

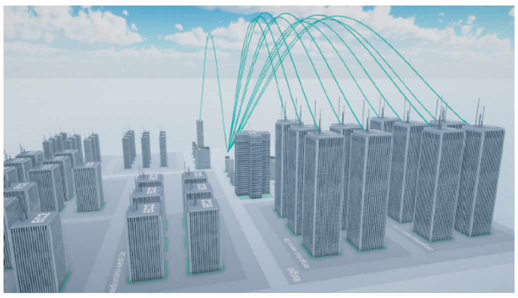


Figure 2.8: City metaphor applied to EC2 deployments in a LiveSD environment

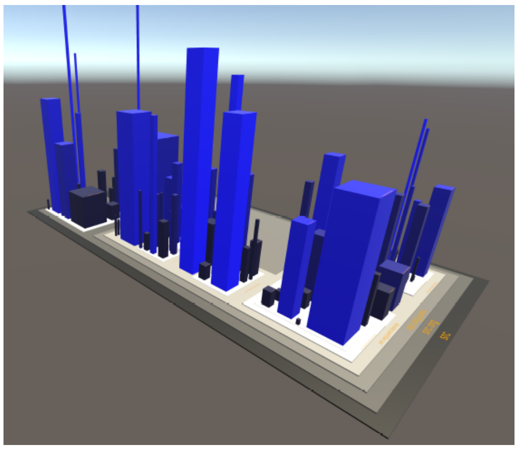


Figure 2.9: Virtual Reality visualization of a Java project

2.6.3 Live Refactoring

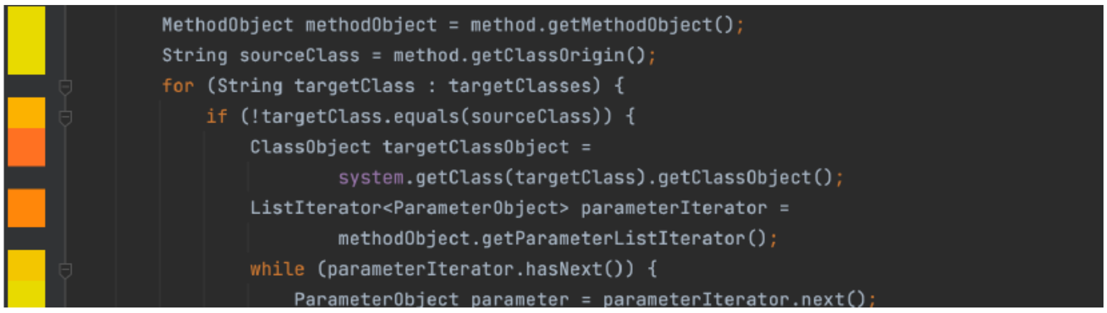
One of the SDLC activities that may benefit from applying a LiveSD approach to it is refactoring. Aguiar et al. [14] suggest that a system supporting Live Refactoring would allow for the automatic detection and application of refactoring opportunities, not through the direct manipulation of code structures but by selecting the quality attributes the programmer intends to improve with the refactoring.

An example of such an environment is one proposed by Fernandes [29]. It computes code quality metrics, uses them to detect code smells, and identifies and proposes refactorings in real-time to users during development to understand which code blocks can benefit from being refactored, as seen in Figure 2.10.

Other tools display Live Refactoring characteristics. For instance, BeneFactor aims to solve the problem of developers not being aware of their IDEs' tools that help refactor code. Therefore, when a practitioner starts applying a refactoring manually, the tool identifies the user attempt and, when invoked, auto-completes the refactoring process [36]. WitchDoctor [31] is a similar tool that also identifies and proposes refactoring opportunities to the user in real-time.

2.6.4 Compiler Platforms

In order to interact and manipulate source code so that patterns can be detected and refactorings applied, the development environment supporting Live Refactoring must have access to the

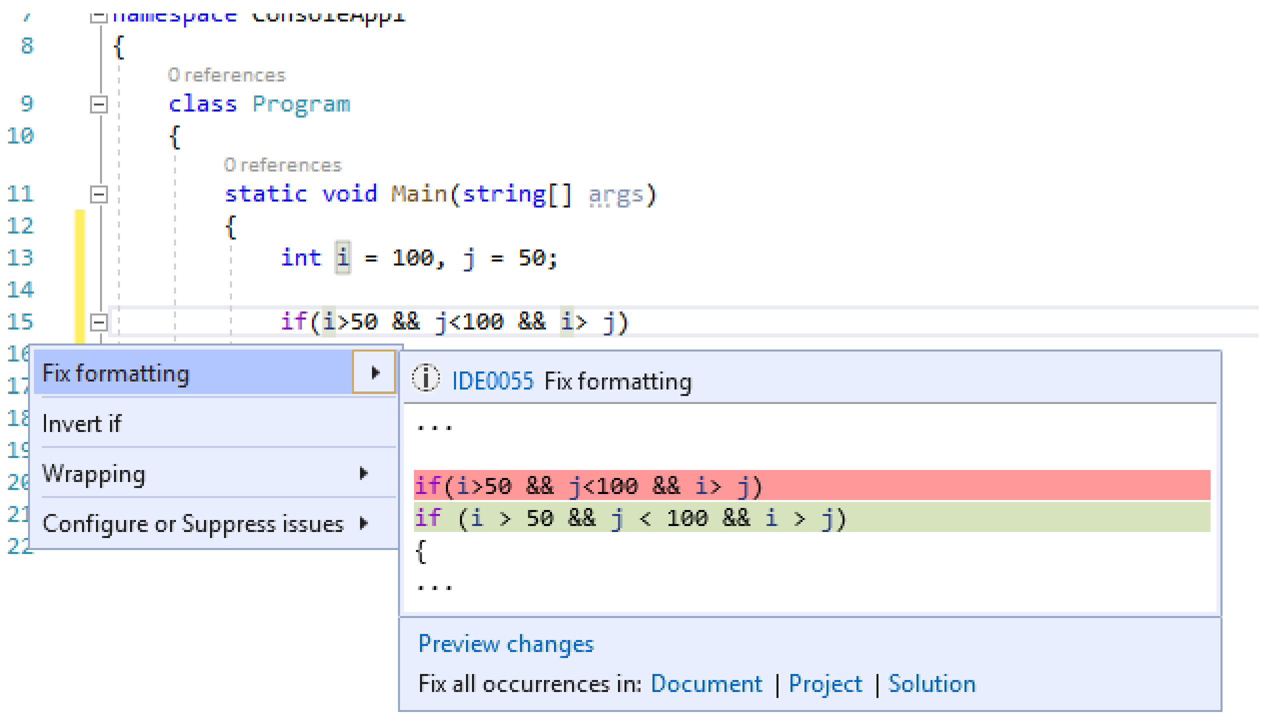


```
MethodObject methodObject = method.getMethodObject();
String sourceClass = method.getClassOrigin();
for (String targetClass : targetClasses) {
    if (!targetClass.equals(sourceClass)) {
        ClassObject targetClassObject =
                system.getClass(targetClass).getClassObject();
        ListIterator<ParameterObject> parameterIterator =
                methodObject.getParameterListIterator();
        while (parameterIterator.hasNext()) {
            ParameterObject parameter = parameterIterator.next();
```

Figure 2.10: A Live Refactoring tool that identifies refactoring opportunities. [29]

platform's compiler tools. These should enable access to syntactic and semantic analysis and automatic code generation, as seen in Figure 2.11.

Roslyn provides helpful features such as code completion, parameter information, smart renaming, and finding references of a particular object that developers can access [82]. It provides access to a large set of code analysis and generation features.



```
namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            int i = 100, j = 50;

            if(i>50 && j<100 && i> j)
```

Fix formatting
Invert if
Wrapping
Configure or Suppress issues

```
IDE0055 Fix formatting
...
if(i>50 && j<100 && i> j)
if (i > 50 && j < 100 && i > j)
{
...
```

Preview changes
Fix all occurrences in: Document | Project | Solution

Figure 2.11: The refactoring experience of Visual Studio 2019. [11]

These features provided by Roslyn enable development environments to support Live Software Development features. In particular, Live Refactoring characteristics such as the previously mentioned refactoring previews, and automatic suggestions and transformation, become possible using Roslyn APIs.

Source generation [84] is a feature that is part of the Roslyn APIs, which are of particular relevance for the process of creating and orchestrating refactoring tools. These work similarly to Roslyn's syntactic and semantic analyzers, which provide information on code quality and style in the shape of warnings or errors, but they also emit source code. By retrieving compilation objects with rich meta-data on syntactic and semantic models, source information can be used and manipulated, and code can be injected directly on compile time. This feature further enables the potential of Live Refactoring tools.

The Java Development Kit also provides access to the platform's compiler through the Java-Compiler interface [5]. It provides similar features to Roslyn, such as syntactic and semantic diagnostics. It also provides an annotation processor and a compiler tree API for traveling the program's AST.

2.7 Summary

Refactoring is a technique that allows for the continued improvement of software during its life cycle. While it does not fix all problems related to software evolution, it dramatically aids developers in building extendable and maintainable systems that are more resilient to change. Support for refactoring tools is vast, with IDEs supplying users with features such as automatic transformations, suggestions, and code previews.

Design patterns are another technique that can increase the resilience of software projects, as they provide tested solutions to recurring problems.

Joining both previous approaches is Refactoring to Patterns. It is an idea that aims to incentivize users to not only apply design patterns at the initial architectural specifications but as change happens and as code is developed by applying refactorings that make the code closer or further from a given pattern.

Meta-programming is the basis of Model-Driven Software Development, which allows for abstracting code into more comprehensible logic structures. With code generation techniques, it is possible to add and transform existing code by manipulating said abstractions, even visually.

Live Refactoring is an idea that aims to bring quick and automatic feedback to developer environments that aid the developer in applying code transformations. It is a subset of Live Software Development, encompassing all SDLC activities to tighten feedback loops.

Existing compiler platforms make it possible to manipulate existing code and generate new one, opening the doors to developing new refactoring tools that use Live Refactoring features.

Chapter 3

Literature Review

After learning about the building blocks of the process of creating refactoring tools, the focus now shifts into understanding existing practices, trends, and techniques of the creation process. The following sections display the state of the art of refactoring creation tools.

3.1 Crossing the New Refactoring Rubicon

Between 49-45 BC, the Roman Republic fought one of its last wars before the surge of the Roman Empire. After his term as governor, Julius Caesar was commanded to demobilize his army and return to Rome. In an act of disobedience, Caesar decided to cross the northern border of Italy recklessly - physically signaled by the Rubicon River, as seen in Figure 3.1 - which was interpreted as an act of treason and declaration of war to the central power of Rome. Since then, the expression "crossing the Rubicon" has been used to signify the passage of a point of no return.

Figure 3.1: The Rubicon river. [68]

When the first Smalltalk refactoring tool was built 25 years ago, Fowler [2] declared that refactoring tools had crossed the refactoring Rubicon. This meant that refactoring support had become mainstream in IDE tools and was something that could not be ignored by such environments going forward [79]. Now, some authors identify batch refactorings as the next Rubicon, resulting from the increased complexity of creating them and orchestrating their internal components [79].

Improving the creation process of refactoring plugins presupposes knowledge of existing refactoring creation techniques and tools. Additionally, it is essential to understand which issues they have and how they can enhance the overall plugin creation experience.

3.2 Batch Refactoring

As software grows in complexity, applying a single refactoring may not remove all code smells that plague programs [28]. This may be a common issue when creating refactorings to design patterns. Thus, developers have to apply additional refactorings to fix these issues. This practice of chaining refactorings in a sequence is usually known as batch, composite or cascaded refactoring.

Batching multiple refactorings together seems to be a regular developer practice [59]. Roughly 40% of refactorings are performed in batches using existing refactoring tools, which reveal themselves as insufficient due to their simplicity and necessity of being significantly improved [48].

One widespread use case of applying batch refactorings is upgrading legacy systems. These are frequently refactored manually by developers with the required knowledge or with the recourse of automatic tools [19]. Either way, developers do not seem comfortable with composing batch refactorings, as they primarily focus on applying refactorings they are familiar with and unaware of others that could be applied. Moreover, developers do not know the impact that the order of the micro-refactoring sequence has on the introduction and removal of code smells, nor are they aware of the effect of batch optimization techniques on code smells [28].

Automatic tools usually let practitioners select which refactorings they want to apply and automatically transform the code to follow a particular pattern. Such tools that support the transformation of several refactorings seem to be desired by developers [17], even if they feel pretty unenthusiastic about using them as they exist today [59]. These could improve the refactoring process in multiple ways, such as reducing the frequency that developers introduce refactorings by hand, which naturally decreases the chances of introducing bugs in code [59].

3.2.1 Formal Models

Chaining refactorings together to form more complex ones is not a trivial task. Each smaller pattern added to the sequence increases the overall complexity of the resulting refactoring, making it harder to ensure that functionality stays the same as the design improves. Cinneide and Nixon [19] present an model for decomposing a larger pattern into mini-patterns, as seen in Figure 3.2. Each mini-pattern has a mini-transformation associated, which is responsible for transforming the code from its current state into the pattern that follows it in the refactoring sequence. After applying a series of successive mini-transformations, the code will represent the design pattern the mini-patterns have been decomposed.

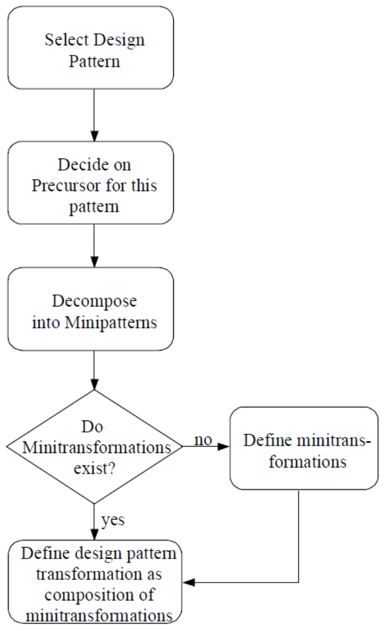

Select Design Pattern
Decide on Precursor for this pattern
Decompose into Minipatterns
Do Minitransformations exist?
no
Define minitrans-formations
yes
Define design pattern transformation as composition of minitransformations

Figure 3.2: Ó Cinnéide and Nixon's methodology for refactoring to patterns. [19]

It may happen that either a mini-pattern has not yet been applied to source code or that the practitioner has tried to fix the problem by introducing small, insufficient hacks instead of implementing a pattern. Therefore, the code may not be in a state where the pattern can easily be applied through refactoring. These two situations are referred to as green-field and anti-pattern situations. An intermediate situation is named a precursor. Precursors are essential when refactoring to patterns, as the code may already have some parts of the refactoring implemented but not all the way through. Identifying the correct precursor makes it possible to identify where the refactoring sequence was interrupted and which mini-transformation should be applied next.

Because multiple mini-refactorings can be combined to form various complex refactorings, it is critical to ensure that invariants are preserved amid transformations. After all, one of the characteristics of the refactoring process is that the code maintains its original functionality. Therefore, each mini-transformation must preserve the pre and post-conditions of successive mini-patterns.

The architecture of the proposed solution also refers to a set of helper functions and predicates used to collect relevant data from the code to support the application of refactorings, which are done by interacting with the code's Abstract Syntax Tree. Moreover, the authors argue that such refactoring to patterns system should reuse as much code as possible from existing refactorings [19], as many design patterns share common foundations. It can be viewed in Figure 3.3.

Kniesel and Koch [47] also defend the division of complex refactorings into smaller, simpler, more reusable units and suggest a formal model that allows creating them automatically and without depending on other programs.

To support chaining more minor refactorings together, the authors introduce the notion of conditional transformations. These pairs of pre-conditions and transformations associated with a pattern are necessary as mini-pattern pre-conditions may not be met when applying a refactoring.

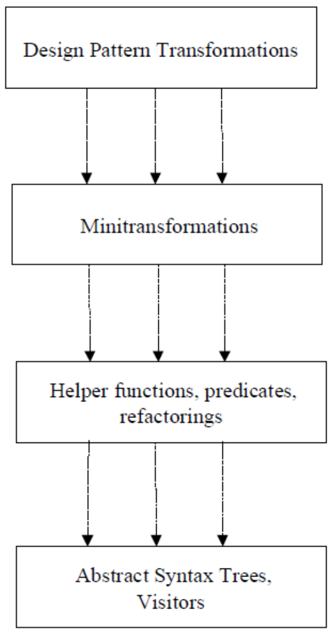

Design Pattern Transformations
Minitransformations
Helper functions, predicates, refactorings
Abstract Syntax Trees, Visitors

Figure 3.3: Proposed architecture for Ó Cinnéide and Nixon's batch refactoring tool. [19]

However, this does not mean that the refactoring cannot be applied. To ensure warrant flexibility for either aborting the refactoring when pre-conditions are not valid or continuing the transformation chain, the authors split refactoring sequences into two different types: AND and OR sequences. The first one may only be applied if there are no failing pre-conditions in the refactoring sequence: otherwise, the refactoring is rolled-back. The latter skips the application of patterns that have failing pre-conditions and tries to apply subsequent ones, not rolling back on previously and successfully applied ones.

The framework also allows for optimizing refactoring sequences by eliminating redundant pre-conditions, which increase tool performance. The model also makes it easy to revert a subset of transformations in a refactoring sequence. However, it is not desirable that the whole refactoring is rolled back when there is a failing AND sequence, as this hinders performance and wastes time. The problem is aggravated because pre-conditions of successive refactorings are naturally altered when code is introduced to apply the mini-patterns of previous refactorings. Back-propagating pre-conditions along the sequence - from last to first - is proposed to solve this issue. A representation of the tool's model framework is shown in Figure 3.4.

The authors argue that tools that implement this model should only require practitioners to specify which refactorings to chain and in which order. Moreover, the tool should manipulate any refactorings, and the composition should be independent of the program on which it will be applied and the programming language.

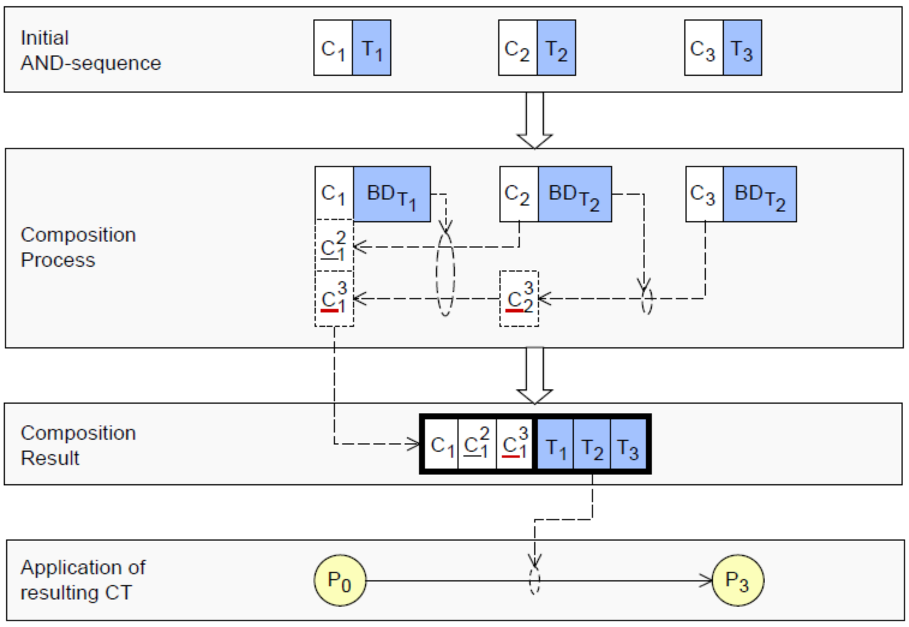


Figure 3.4: Workflow of the application of batch refactorings in ConTraCT. [47]

3.2.2 Architectures

Given that batch refactoring tools interact with more code structures throughout the refactoring sequence than traditional ones, Zannier and Maurer [86] suggest following a different approach. By empowering these tools with a much more extensive knowledge-base of the overall system architecture and knowledge of the design patterns to be applied, refactorings can be more appropriately applied. So that this can be more automatic and accessible, the set of atomic and sequential refactorings should generate code associated with all classes required for the correct implementation of a pattern. Therefore, refactoring tools should be developed to preserve the previously mentioned characteristics. With this in mind, the authors propose that knowledge should be split into three distinct architectural components:

- **Initial System Store**: contains general, more abstract information on the system requirements and structure;
- **Rule Store**: has information on rules and restrictions related to the refactoring's domain;
- **Target System Store**: stores information on created batch refactorings.

Fernandes et al. [28] explore alternatives for creating batch refactorings in a distributed way. The authors state that the application of batch refactorings happens, more often than not, during code reviews and with the help of automatic tools. They suggest three alternatives to improve the batch refactoring creation process during code reviews. The authors suggest the architecture of a suggestion tool that lets the user control which recommendations to accept and reject.

The first proposes that the developer accepts or asks for another suggestion for each refactoring in the batch sequence proposal. Depending on the answers to those questions, different sequences may be formed and ultimately applied by the practitioner. The second targets the code review process and is implemented similarly to the previous one. The difference is that these sequences are also recommended in code review platforms so that other reviewers can opine and discuss the characteristics of a given batch to form a consensus on which sequence is the best. The third allows multiple reviewers to collaborate on creating a refactoring sequence by letting each one either accept a suggestion or request an alternative. When all reviewers accept a suggestion, the corresponding micro-refactoring is added to the sequence. The next one gets discussed similarly until the tool exhausts its suggestions. Figure 3.5 represents the tool's architecture.

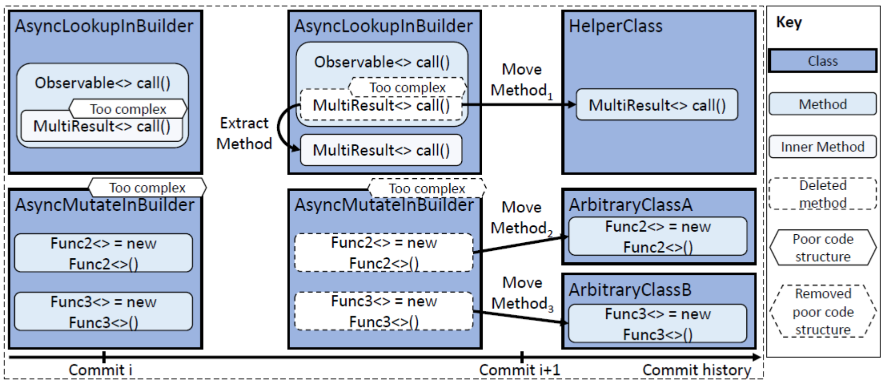


Figure 3.5: Creating batch refactorings for code reviews. [28]

For this tool to be helpful and integrate well into the development and code review experience, it needs to be well incorporated into IDEs. The tool must always be aware of code changes to suggest batches. In turn, the automatic recommendations incentivize practitioners to frequently discuss how and whether to apply a given batch.

3.2.3 Discussion

Batch refactoring is a technique that consists of chaining more minor refactorings together to form a more complex one. It serves as the basis for any complex refactoring creation tool, encompassing multiple responsibilities such as the specification, identification, and application of refactorings. While a practical methodology, practitioners are still uncomfortable applying it, as development tools do not sufficiently cater to their needs.

Formal models for batch refactoring tools are mostly based on applying small transformations throughout the refactoring sequence to ensure the preservation of invariants related to each smaller refactoring. Refactorings with part of their transformation sequence failing to comply with pre and post-conditions may either be canceled or skipped, according to the characteristics of the complex refactoring.

If these models are to be implemented and expanded by modern development environments, they must be as user-friendly and customizable as possible. Available refactoring tools of IDEs such as IntelliJ and Visual Studio do not support any model minimally close to what is proposed.

Ultimately, it seems that the crossing of the second refactoring Rubicon is not yet a reality and will not be unless significant developments occur in the environments' plugin ecosystems. This is ever more important given that developers may not know about these batch refactoring techniques, decreasing the chances that they will ever implement such automatic transformations if the environments do not facilitate and explicit the process to them well enough.

3.3 Refactoring Tools and Techniques

There are multiple tools offered to developers for refactoring code. Some examples are JDeodorant, JMove, FaultBuster, and Refactoring Navigator. However, as Tenorio, Bibiano, and Garcia [59] point out, these are too simplistic [13] and are not a good fit for creating, customizing, and applying more complex patterns [28]. Consequently, modern IDEs do not integrate these features into their toolchains. Other problems are the incorrect refactoring implementations that do not preserve invariants and too strong refactoring pre-conditions that do not detect nor support subtle variations of refactorings. Also, a refactoring opportunity may be identified, but the tool may not allow users to select from a set of these variants the one that best caters to that situation [13].

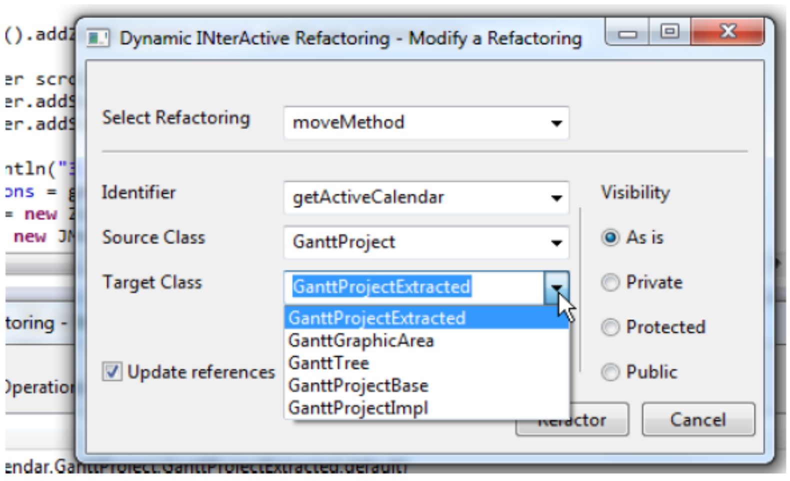
IDE extensibility toolkits such as Eclipse LTK are also hard to use and work at shallow abstraction levels, which may significantly decrease developer adoption rates [48]. These issues are a consequence of, among other reasons, IDEs not having good enough static analysis features, as well as not detecting and preserving invariants well enough during transformations [13]. The latter may even hinder the implementation of batch refactoring creation features in development environments. Practitioners specify new refactorings manually and statically concerning existing batch refactoring to patterns tools. This may be a problematic approach, as manually enumerating the primitive refactorings that make up a complex refactoring can be tedious and error-prone. Moreover, refactorings yet to be applied in the complex refactoring sequence may not be aware of the effects on code provoked by previous primitive refactoring applications [47].

While referring that most IDEs support the creation of new refactorings and the generation of code from templates, Hunold et al. [38] point out that most tools do not separate concerns as clearly as desired, which may lead to confusing architectures. Moreover, manually detecting candidate patterns for refactoring in source code is not easy, especially if the practitioner looking at the code is not the one who wrote it and if the code and architecture are not sufficiently well documented.

Refactorings can only be automatically suggested if patterns are found in code. In order to incentivize practitioners to refactor more often, the implementation is of utmost importance. Also important are ways to specify, in an easy and accessible way, custom refactorings created by developers and mechanisms to apply them automatically.

Regarding the pattern detection feature, Yoshida and Inoue [85] suggest following an approach based on code clone detection so that methods matching a specific pattern can be detected and refactored, referring to the CCFinder tool [43] as an example. It is then assessed if the functions found by the tool have common super-classes and if they have statements that declare object creation.

Acknowledging that manually refactoring code is a tedious procedure for practitioners, Mkaouer et al. [55] propose a tool, DINAR, for suggesting refactorings in a batch that is fully integrated with IDEs such as Eclipse. These recommendations are not linear, as the tool uses a set of metrics for ranking refactorings by their usefulness. The tool can be visualized in Figure 3.6.



Select Refactoring: moveMethod
Identifier: getActiveCalendar
Source Class: GanttProject
Target Class: GanttProjectExtracted
GanttProjectExtracted
GanttGraphicArea
GanttTree
GanttProjectBase
GanttProjectImpl
Update references [✓]
Visibility: [✓ As is] Private Protected Public

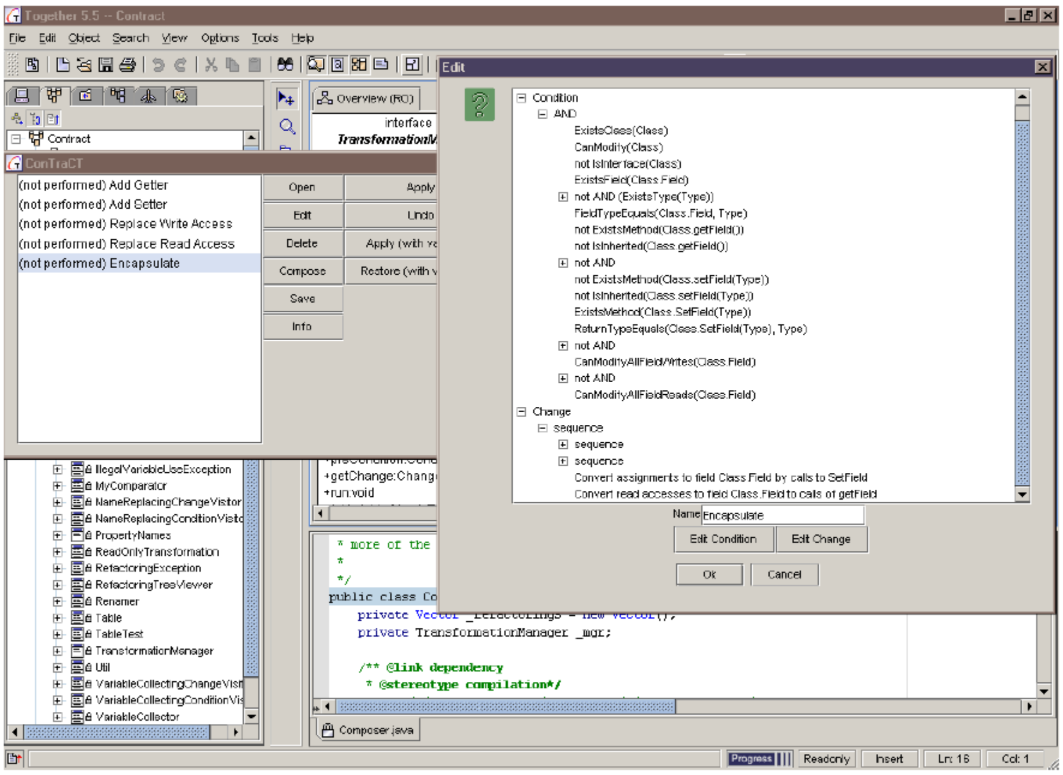
Figure 3.6: The DINAR batch refactoring suggestion tool. [55]

One of the authors' most significant contributions is a novel approach to identifying patterns and recommending refactorings. After identifying patterns that match those the tool can search for, refactorings are generated in a way that tries to both maximize code quality improvements and minimize the amount of code smells total amount of micro-refactorings, all while preserving the actual behavior of the program.

The practitioner can glance at the suggested batch refactorings list and accept or dismiss them. The tool allows learning from user feedback and tweaking the suggestions to maximize the user's acceptance rate to suggest efficient refactorings that have higher chances of being applied by the developer. Studies found it to offer a faster alternative to manual refactoring and alternative automatic tools.

Regarding the user experience and Live Refactoring capabilities, Kniesel and Koch [47] propose a tool that provides a code wizard with multiple pages for guiding the user through the creation of the refactoring, asking relevant questions about the characteristics of the patterns that will make up the refactoring. This wizard, represented in Figure 3.7, adds liveness to the refactoring creation process and allows developers not to have to repeat as much code.

The author's implementation, ConTraCT, was developed on top of jConditioner and the Eclipse platform. It allows users to visualize sequences in a tree layout and create a sequence using a concrete language. This implementation does not support Live Refactoring capabilities such as automatic suggestions or previews.



(not performed) Add Getter
(not performed) Add Setter
(not performed) Replace Write Access
(not performed) Replace Read Access
(not performed) Encapsulate

```
⊟ Condition
  ⊟ AND
      ExistsClass(Class)
      CanModify(Class)
      not IsInterface(Class)
      ExistsField(Class.Field)
    ⊞ not AND (ExistsType(Type))
      FieldTypeEquals(Class.Field, Type)
      not ExistsMethod(Class.getField())
      not IsInherited(Class.getField())
    ⊞ not AND
      not ExistsMethod(Class.setField(Type))
      not IsInherited(Class.setField(Type))
      ExistsMethod(Class.SetField(Type))
      ReturnTypeEquals(Class.SetField(Type), Type)
    ⊞ not AND
      CanModifyAllFieldWrites(Class.Field)
    ⊞ not AND
      CanModifyAllFieldReads(Class.Field)
⊟ Change
  ⊟ sequence
    ⊞ sequence
    ⊞ sequence
      Convert assignments to field Class.Field by calls to SetField
      Convert read accesses to field Class.Field to calls of getField
```

Name: Encapsulate

Figure 3.7: The user interface of the ConTraCT tool. [47]

Similar to other research, Tenorio, Bibiano, and Garcia [59] propose splitting patterns into smaller ones that can be orchestrated as developers see fit. The authors developed an interface prototype that allows practitioners to visualize, add and edit the refactoring flow, which comprises existing simple and batch refactorings that compose a particular complex transformation and is represented in Figure 3.8. These existing refactorings are provided as a list to users, who can pick the ones most relevant to the refactoring they are building.

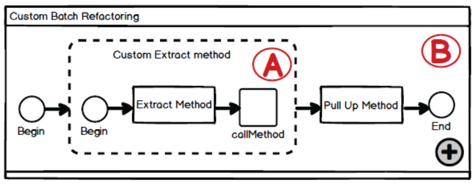


Figure 3.8: A prototype of a visual refactoring tool. [59]

Some of the benefits of this idea, according to the authors, are increased predictability, the ability to preview intermediate refactoring transformations, more flexibility in refactoring creation, and code reuse. However, programmers must program the automatic detection of refactoring opportunities in the IDE, which, for more complex ones, may not be a trivial task. The visual way of creating refactorings contributes to increasing the liveness of the development environment and, consequently, reduces the development feedback loop.

Tourwé and Mens [78] recognize that developers do not always identify refactoring opportunities. Even if they do, they often do not have the knowledge to refactor by hand. The authors propose another tool, as shown in Figure 3.9, for automatically finding patterns in code and suggesting the appropriate refactorings.

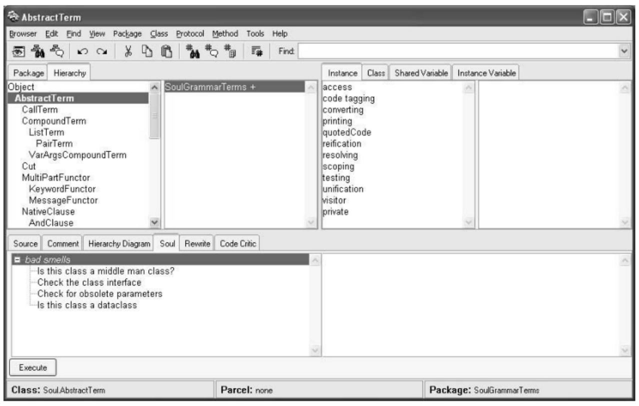


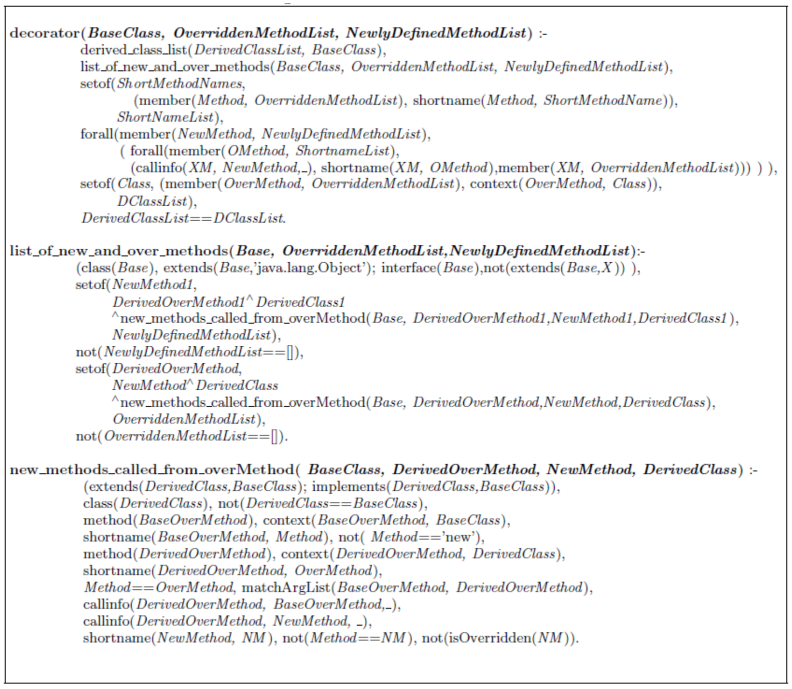
Figure 3.9: Tourwé and Mens' refactoring identification tool. [78]

This specific implementation follows a novel approach based on the concept of logic metaprogramming, combining an object-oriented language with a declarative meta-language to improve software systems. In the case of the authors' implementation, the SOUL logic metalanguage represents the program in an abstract and declarative way, which is then linked to the Smalltalk object-oriented language so that the abstract model can be in sync with the implementation and interact with the program's AST. The logic layer, thus, has full access to the source code.

This approach makes it easier to perform complex queries to code structures, which is vital to identify and retrieve the components that make up patterns. The authors developed a tool that follows the described strategy, which is also compatible with detecting suggesting batch refactorings. However, the tool does not support the automatic application of the refactoring suggestions, having the developer implement them manually. However, checking for batch refactorings may negatively affect performance. Also, as pattern identification is based on the declaration of logic predicates, when a new pattern is introduced, old rules may need to change to cope with it. This manual update process may be cumbersome to developers.

The tool is integrated into the VisualWorks Smalltalk IDE in the Refactoring Browser component. The IDE automatically handles the pattern searching and matching, allowing practitioners to select which patterns to look after, helping minimize performance concerns.

Rajesh and Janakiram [64] propose JIAD, a prototype refactoring tool that automatically detects patterns and ensures invariant preservation. In this implementation, patterns are identified using declarative programming rules written in Prolog, some of which are listed in Figure 3.10, which are used by the AspectJ compiler to traverse the AST and match with existing code. Logic predicate templates are provided to better describe object and aspect-oriented constructs such as classes, interfaces, and methods.



```
decorator(BaseClass, OverriddenMethodList, NewlyDefinedMethodList) :-
          derived_class_list(DerivedClassList, BaseClass),
          list_of_new_and_over_methods(BaseClass, OverriddenMethodList, NewlyDefinedMethodList),
          setof(ShortMethodNames,
                  (member(Method, OverriddenMethodList), shortname(Method, ShortMethodName)),
               ShortNameList),
          forall(member(NewMethod, NewlyDefinedMethodList),
                ( forall(member(OMethod, ShortnameList),
                  (callinfo(XM, NewMethod,_), shortname(XM, OMethod),member(XM, OverriddenMethodList))) ) ),
          setof(Class, (member(OverMethod, OverriddenMethodList), context(OverMethod, Class)),
               DClassList),
          DerivedClassList==DClassList.

list_of_new_and_over_methods(Base, OverriddenMethodList,NewlyDefinedMethodList):-
         (class(Base), extends(Base,'java.lang.Object'); interface(Base),not(extends(Base,X)) ),
         setof(NewMethod1,
              DerivedOverMethod1^DerivedClass1
              ^new_methods_called_from_overMethod(Base, DerivedOverMethod1,NewMethod1,DerivedClass1),
              NewlyDefinedMethodList),
         not(NewlyDefinedMethodList==[]),
         setof(DerivedOverMethod,
              NewMethod^DerivedClass
              ^new_methods_called_from_overMethod(Base, DerivedOverMethod,NewMethod,DerivedClass),
              OverriddenMethodList),
         not(OverriddenMethodList==[]).

new_methods_called_from_overMethod( BaseClass, DerivedOverMethod, NewMethod, DerivedClass) :-
          (extends(DerivedClass,BaseClass); implements(DerivedClass,BaseClass)),
          class(DerivedClass), not(DerivedClass==BaseClass),
          method(BaseOverMethod), context(BaseOverMethod, BaseClass),
          shortname(BaseOverMethod, Method), not( Method=='new'),
          method(DerivedOverMethod), context(DerivedOverMethod, DerivedClass),
          shortname(DerivedOverMethod, OverMethod),
          Method==OverMethod, matchArgList(BaseOverMethod, DerivedOverMethod),
          callinfo(DerivedOverMethod, BaseOverMethod,_),
          callinfo(DerivedOverMethod, NewMethod, _),
          shortname(NewMethod, NM), not(Method==NM), not(isOverridden(NM)).
```

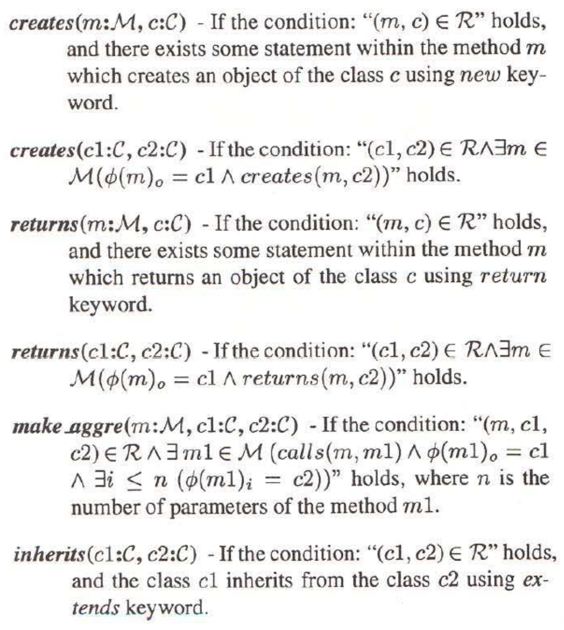
Figure 3.10: The JIAD tool for identifying refactoring opportunities. [64]

The referred tool allows the user to visualize the identified patterns and automatically apply them. However, it does not support batch refactoring and does not integrate with IDEs, which means it lacks Live Refactoring features.

Jeon et al. [42] propose a tool for identifying candidate patterns in source code for refactoring based on inference rules, which are then evaluated and selected for transforming source code, some of which are shown in Figure 3.11. The evaluation is based on a refactoring strategy algorithm that typically forms a batch refactoring sequence.

The inference rules that can be used are part of a pervasive set of logic predicates that correspond to operations in Java programs. This way, the behavior of a code block can be represented in a powerful formal model. When there are matches between rules and concrete code, the tool suggests a refactoring to the developer. The practitioner can then decide whether or not to apply the suggestion. If so, the transformations are applied through interactions with JavaCC, Java's compiler that provides access to the program's AST for manipulating source code.

Source-to-source transformations are used in many software engineering activities, such as forward engineering and language translation, as mentioned by Cordy et al. [21]. The refactoring activity can also greatly benefit from such transformations by specifying code changes in an appropriate language. TXL is an example of a source transformation language and processor that provides rapid code prototyping support. Following the extended Backus-Nauer form, the language uses unrestricted, context-free grammar. The workflow of the TXL processor can be visualized in Figure 3.12. The code to be transformed is described in the TXL language, which is then passed to the TXL processor that parses the input program and automatically outputs a source artifact with the desired transformations.



creates$(m{:}\mathcal{M}, c{:}\mathcal{C})$ - If the condition: "$(m, c) \in \mathcal{R}$" holds, and there exists some statement within the method m which creates an object of the class c using *new* keyword.

creates$(c1{:}\mathcal{C}, c2{:}\mathcal{C})$ - If the condition: "$(c1, c2) \in \mathcal{R} \wedge \exists m \in \mathcal{M}(\phi(m)_o = c1 \wedge creates(m, c2))$" holds.

returns$(m{:}\mathcal{M}, c{:}\mathcal{C})$ - If the condition: "$(m, c) \in \mathcal{R}$" holds, and there exists some statement within the method m which returns an object of the class c using *return* keyword.

returns$(c1{:}\mathcal{C}, c2{:}\mathcal{C})$ - If the condition: "$(c1, c2) \in \mathcal{R} \wedge \exists m \in \mathcal{M}(\phi(m)_o = c1 \wedge returns(m, c2))$" holds.

make_aggre$(m{:}\mathcal{M}, c1{:}\mathcal{C}, c2{:}\mathcal{C})$ - If the condition: "$(m, c1, c2) \in \mathcal{R} \wedge \exists m1 \in \mathcal{M}\ (calls(m, m1) \wedge \phi(m1)_o = c1 \wedge \exists i \leq n\ (\phi(m1)_i = c2))$" holds, where n is the number of parameters of the method $m1$.

inherits$(c1{:}\mathcal{C}, c2{:}\mathcal{C})$ - If the condition: "$(c1, c2) \in \mathcal{R}$" holds, and the class $c1$ inherits from the class $c2$ using *extends* keyword.

Figure 3.11: Some inference rules of the tool proposed by Jeon et al. [42]

TXL is a flexible language, allowing practitioners to reuse transformations previously declared using the language. It works by finding patterns in code that correspond to the rules defined in the TXL file, replacing them if a match is found.

The language has been used for multiple purposes, such as translating interfaces to multiple languages according to a specification, creating language dialects by implementing semantic extensions and obtaining higher-level entity abstractions from source code implementations.

By implementing a similar model to the one described by Kniesel and Koch [47], Li and Thompson [48] present a framework powered by a DSL for creating reusable, complex refactorings that propagate failures along the refactoring sequence and generate refactorings automatically. The tool, Wrangler, is implemented using the Erlang programming language and integrated into Eclipse and Emacs. Refactorings are also divided into mini and complex refactorings, and there are transformation rules that automatically convert source code. Moreover, complex refactorings provide the option of continuing or halting execution if one of the sequential transformations fails. Visualizing Figure 3.13 allows for a better understanding of these behaviors.

The authors have designed the DSL so that primitive refactorings have a code generator associated, which can be lazily applied to ensure that only the most up-to-date program information is used.

Unlike other implementations, this one does not derive nor propagate invariants and primitive refactorings, which helps increase expressibility even if it reduces overall performance. The tool works by receiving a complex refactoring script, which after being interpreted, outputs refactoring commands. These will then travel along with the AST until they match the abstract pattern templates with relevant implementation code and apply the transformations according to the refactoring sequence. This tool has Live Refactoring characteristics such as code previews, automatic

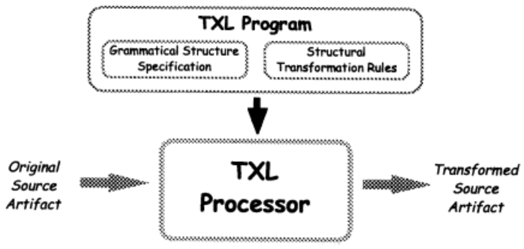

TXL Program
Grammatical Structure Specification
Structural Transformation Rules
Original Source Artifact
TXL Processor
Transformed Source Artifact

Figure 3.12: The TXL processor. [21]

refactoring application, and undoing.

Schäfer and de Moor [70] also provide an implementation with support for splitting patterns into smaller ones but take a different approach for chaining refactorings and propagating changes throughout them. The authors argue that the usual pre-condition-based approach is not good enough to cope with data flow and name-binding preservation problems. For instance, as successive micro-refactoring applications alter variable names, successive refactorings might not be successfully applied as, according to their context, the variables no longer exist. Moreover, when variables with the same name but in different scopes exist, intermediate refactorings may get confused and transform code that should be kept as is. The authors tackle these problems as dependency preservation ones. In the case of changing variable names, after that micro-refactoring is applied, the refactoring engine checks if the name keeps its binding with the variable declaration. If not, the refactoring is canceled, or the binding is updated to reflect that name change. Figure 3.14 shows an implementation of a refactoring with the JastAdd tool. On another note, Eclipse, for instance, does not ensure behavior preservation, which the authors find disturbing.

Another key difference is that refactorings are described using language extensions, in this case for Java 5, and the refactoring engine is built on top of the JastAddJ compiler. The engine accepts these language extensions as inputs, which are also used for handling intermediate refactorings. These facilitate the refactoring process as refactorings can be more expressively described without removing the simplicity and clarity of the refactoring specification model. When the refactored code is generated, it is stripped from these language extensions and converted into pure Java code.

By bench-marking the built-in Eclipse refactorings with equivalent ones created using this refactoring engine and language, the novel approach's implementations are much more explicit and concise, code-wise, making it much easier to understand the refactoring specifications and implementations. Also, these often have higher correctness levels than the built-in ones.

However, the developed engine lacks performance, with smaller refactoring taking up to 5 seconds to be applied. Additionally, the tool does not integrate with IDEs and cannot take advantage of Live Refactoring capabilities.

Kataoka et al. [44] propose a tool, Daikon, for automatically identifying candidate patterns for refactoring through the identification of program invariants in source code. The tool's invariant inference feature can be visualized in Figure 3.15. This approach identifies said invariants and can infer them directly from the source code through the tool. The tool relies on users to decide

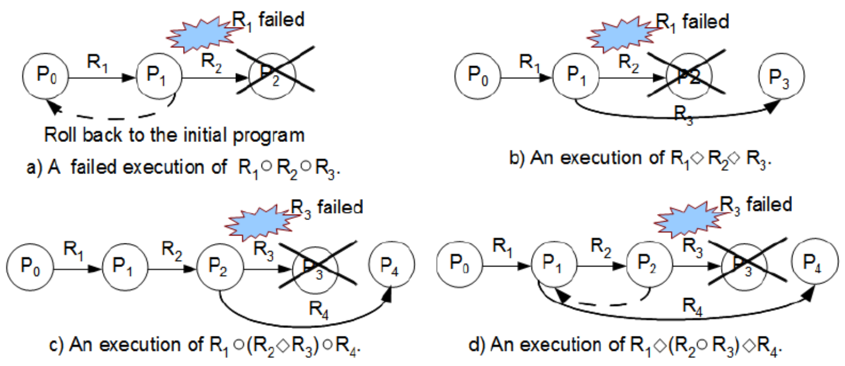

R1 failed
P0
R1
P1
R2
P2
Roll back to the initial program
a) A failed execution of R1○R2○R3.
R1 failed
P0
R1
P1
R2
P2
R3
P3
b) An execution of R1◇R2◇R3.
R3 failed
P0
R1
P1
R2
P2
R3
P3
P4
R4
c) An execution of R1○(R2◇R3)○R4.
R3 failed
P0
R1
P1
R2
P2
R3
P3
P4
R4
d) An execution of R1◇(R2○R3)◇R4.

Figure 3.13: Execution of a batch refactoring. [48]

whether or not to apply a refactoring.

The authors point out that, even if users decide not to apply the suggested refactorings, these may still be useful for pointing out aspects that should be better documented in the code. The tool is not an all-in-one package, consisting of a set of standalone Perl scripts for identifying invariants and, consequently, potential refactoring opportunities, and the Daikon tool for inferring invariants from source code. This means there is no IDE integration.

This solution relies on dynamic analysis techniques, meaning that transformations need to be verified to ensure behavior preservation by the user or through static analysis tools. The authors propose a mixture of dynamic and static analysis techniques to ensure refactoring correction and integrity.

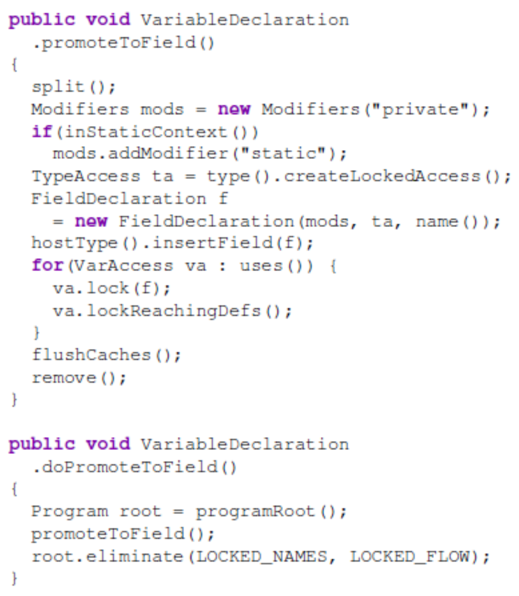
Derezinska [24] described a tool that enables automatic and manual refactoring. The selected approach analyzed how relevant each source code block was to a set of built-in design patterns. The program calculates a relevance metric describing the similarity between a code block and a design pattern.

Patterns are only applied if they respect the respective pre-conditions. The tool allows the programmer to automatically apply the code transformations that implement the pattern.

This tool extends the Eclipse environment that operates over the Java programming language. In particular, Eclipse's Refactoring Framework provides the tool with Live Refactoring capabilities such as automatic refactoring application, code previewing, and accessing a history of applied refactorings. The framework that is associated with this tool allows adding new refactorings quickly.

Wei et al. [50] propose approaches for refactoring two different design patterns from Gamma et al. [35] automatically. Again, the overall architecture is based on two main components: one for identifying patterns that constitute refactoring candidates and another that automatically transforms the source code. The author's implementation targets the Java programming language, but the logic can be applied to any other language. It extends the Eclipse development environment, making use of the JDT APIs. The refactoring process can be visualized in Figure 3.16.

The identification stage is performed by automatically assessing the program's AST and the



```
public void VariableDeclaration
  .promoteToField()
{
  split();
  Modifiers mods = new Modifiers("private");
  if(inStaticContext())
    mods.addModifier("static");
  TypeAccess ta = type().createLockedAccess();
  FieldDeclaration f
    = new FieldDeclaration(mods, ta, name());
  hostType().insertField(f);
  for(VarAccess va : uses()) {
    va.lock(f);
    va.lockReachingDefs();
  }
  flushCaches();
  remove();
}

public void VariableDeclaration
  .doPromoteToField()
{
  Program root = programRoot();
  promoteToField();
  root.eliminate(LOCKED_NAMES, LOCKED_FLOW);
}
```

Figure 3.14: Implementation of the refactoring Promote Temp to Field. [70]

application of source code transformations. A match happens when a set of conditional statements related to each design pattern are respected. Then, optimizations occur to ensure the optimal refactoring is selected. Multiple simple refactorings can be combined to form more complex refactorings.

Experiments over the developed tool demonstrate that refactored code is more maintainable and extensible by assessing a set of pre-defined metrics.

Hunold et al. [38] address a tool, TransFormr, for identifying patterns in code and automatically applying them. The authors mention that such features could bring significant improvements, such as refactoring legacy code bases. It also allows the visualization of class dependencies in patterns, as shown in Figure 3.17.

The authors divided the refactoring process into three stages in the proposed implementation. The first is the extraction phase, where an AST is generated from source code and categorized according to several characteristics such as business logic, database, or user interface. The next step involves applying the transformations associated with a specific pattern at the abstract representation level. Finally, source code is generated to represent said transformations.

Source transformations and pattern specifications are defined using the TXL language to annotate and extract the abstract model from the source code. While the user must decide if the suggestions should be applied, the tool provides several different ways of visualizing the suggested transformations. Moreover, it is also possible to create batch refactorings.

3.3.1 Evolutionary Algorithms for Code Transformations

The literature points out multiple attempts to improve the refactoring process. Some of the most interesting ones involve optimizing the identification and generation of patterns and refactoring

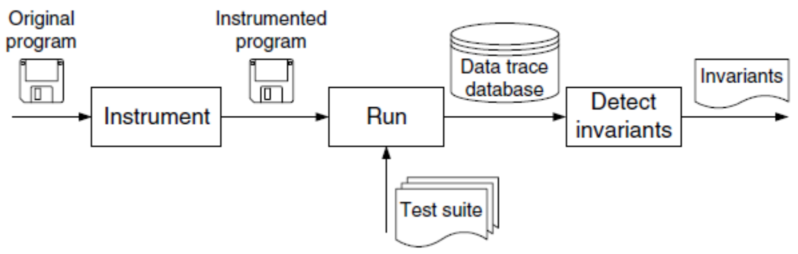


Figure 3.15: The invariant inference mechanism of the Daikon tool. [44]

sequences.

Ouni et al. [60] present a tool, MORE, that simultaneously identifies anti-patterns and introduces patterns. It handles refactoring as an optimization problem, using multiple algorithms, such as hill-climbing, simulated annealing, and genetic algorithms to maximize performance metrics and generate an optimal refactoring sequence. The architecture of the tool can be visualized in Figure 3.18.

The tool parses and analyzes source code to create an internal representation of the program to manipulate. An anti-pattern detector, implemented with rules based on logic predicates, matches code with the set of anti-patterns known by the tool. Similarly, there is also a pattern detector for finding instances of design patterns already present in the code. The tool also has knowledge on the list of refactorings to apply, which are aborted when any of the micro-refactorings that make up a sequence fails, and a quality evaluator for assessing how positive or negative the impact of refactoring was. Finally, a coherence constraints checker ensures behavior and semantic preservation.

Experiments reveal promising results. However, some situations may drift the overall program architecture from its original design in ways that are not beneficial overall. One example is when developers have to tweak refactoring applications to fit specific use cases better manually, which is a consequence of not being able to generalize refactorings enough.

Jensen and Cheng [41] propose an alternative way to generate micro-refactoring sequences other than following a rule-based approach. The authors propose following an evolutionary-programming approach and using genetic programming algorithms to identify the most-beneficial sequence to a given set of micro-refactorings. While refactoring tools that use this kind of algorithm exist, these do not allow creating batch refactorings. Moreover, the authors' approach focuses on finding the best sequence for applying them and creating refactorings. The authors have developed a tool based on these ideas, which features Live Refactoring capabilities through the automatic suggestion and application of refactorings while also allowing doing so manually.

The tool represents the program being refactored as a graph based on a UML class diagram augmented with semantic details like class instantiations and method calls. A transformation tree encodes the changes to apply to the graph mentioned above. After applying the changes, the graph assesses how much a set of metrics was improved. The rest of the process goes just like any genetic algorithm. Design patterns are detected through a Prolog program responsible for

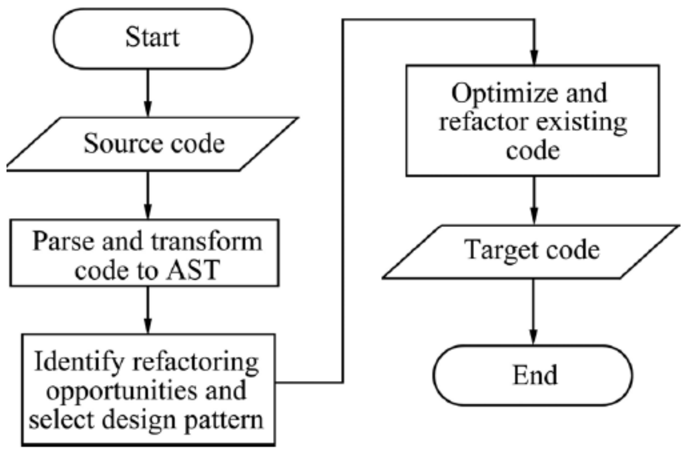

Start
Source code
Parse and transform code to AST
Identify refactoring opportunities and select design pattern
Optimize and refactor existing code
Target code
End

Figure 3.16: The refactoring process of the tool proposed by Wei et al.. [50]

matching patterns with the analyzed code.

Shimomura [71] has implemented a tool for suggesting and applying refactorings to design patterns for Java code based on genetic algorithms, albeit in a different manner. The novelty of this approach is that these algorithms are used to generate implementations of design patterns, which are then evaluated against a set of quality criteria. For each design pattern of the set, two sample sets of programs are created: a set of flawed programs that could benefit from applying the pattern and a set of sound patterns that have already been applied. Then, an algorithm that assesses program quality concerning a range of quality metrics compares good and bad samples. After that, a genetic algorithm verifies which metrics better serve to differentiate good and bad design patterns. Finally, these metrics are applied to the target program so the tool can know if a design pattern should or should not be recommended. Figure 3.19 represents the tool's architecture.

The authors identify a set of advantages of using such an approach. One of them is adding new metrics to the tool to use by the genetic algorithm. Another is that this implementation can provide developers feedback on which metrics are most beneficial to evaluate the possibility of applying patterns.

3.3.2 Model-Driven Refactoring

Another innovative technique for creating and applying batch refactorings to patterns is interacting with the program at the more abstract level of class diagrams instead of directly on code.

Model-Driven Software Engineering suggests, among other things, abstracting implementation into models for more effortless comprehensibility and evolution, which will have the corresponding code automatically generated by the environment. Consequently, refactoring said abstract structures into design patterns is a more general and language-independent process, and the transformation operations need to interact directly with the models.

El-Sharqwi et al. [27] propose a model for specifying design patterns and associated refactoring transformations using XML, a flexible, popular, and extensible language, and the designed

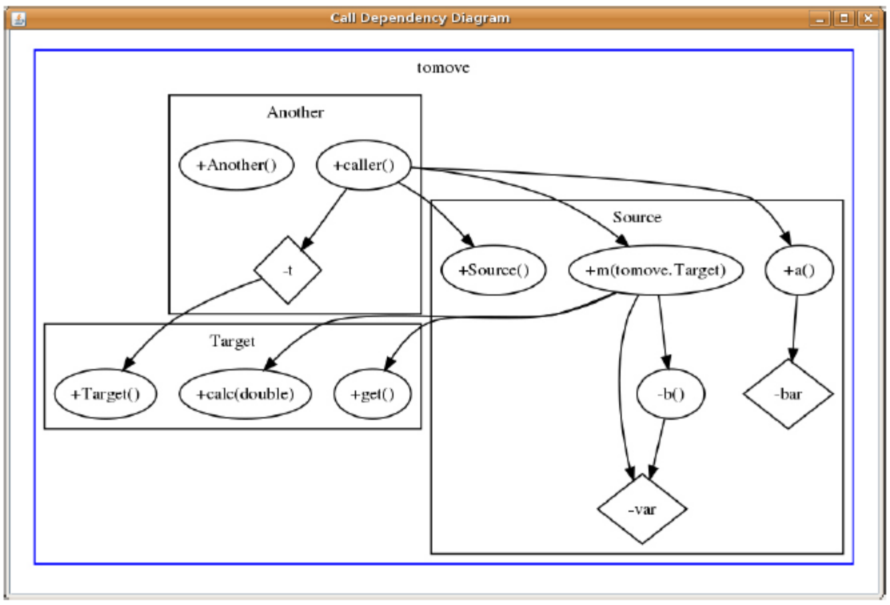


Figure 3.17: Visualizing class dependencies in the TransFormr tool. [38]

models can be serialized into UML, which provides clarity when visualizing models. Figure 3.20 shows an example of a pattern described in XML.

The detection of design patterns in code is implemented as a constraint satisfaction problem. The constraints are related to the relationships between models, such as inheritance and associations, and the respective variable domains. Model transformations are the collection of micro-refactorings that form more complex ones. Code is ultimately generated from the models, as usual with MDSE approaches. Implementation is based on top of NRefactory, a code generator and parser for the .NET platform.

France et al. [34] also address the issue of refactoring to patterns following a meta-modeling software development approach. The authors argue that automatic refactoring to patterns tools can help ensure that code smells are not introduced during transformations and preserve developers' time and energy from tedious and repetitive tasks. After identifying patterns in the model abstractions representing the source code, the authors suggest adding, removing, or reorganizing model elements to apply the refactorings. Figure 3.21 describes the proposed meta-modelling approach.

Meta-models are also used for defining patterns, and the associated transformations, which can be many, as a pattern may be embedded into the design in several ways. The refactoring patterns and the implementation abstractions of concrete code are specified using a UML model. The model's M2 level, an extension of UML's meta-model, represents the transformation language used to specify the required changes that compose a given transformation. The M1 level extends the UML model to represent abstract, related transformations to a concrete implementation in the M0 level. Transformations are suggested when there are matches between the models that define patterns and the abstractions of specific code implementation.

The developed tool has a few issues that prevent it from being production-ready. The first is

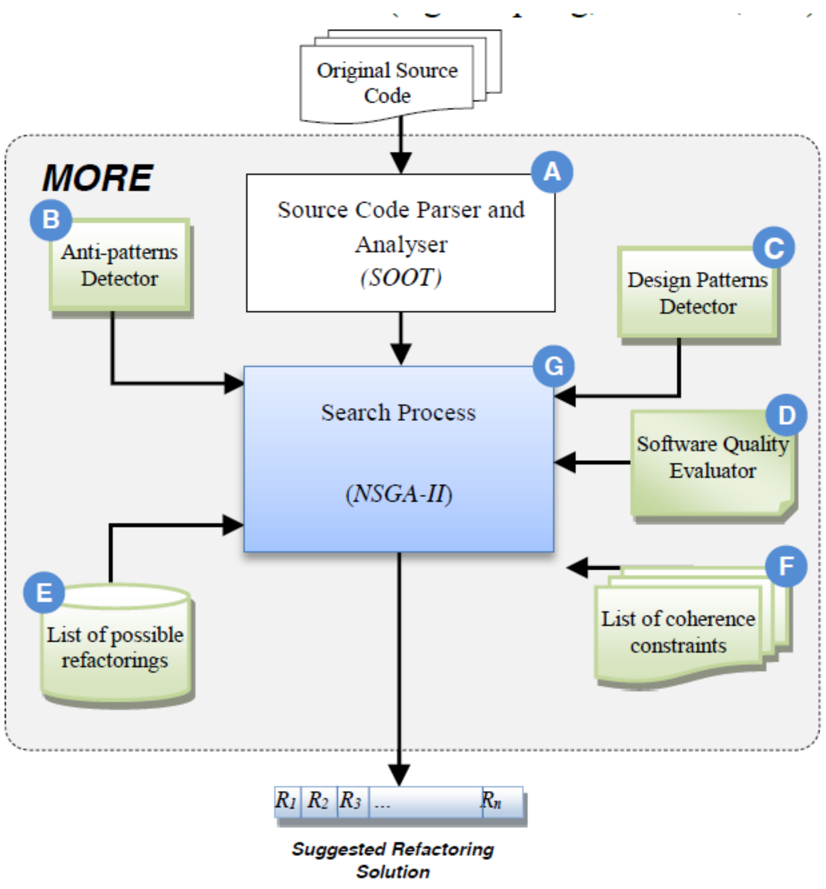

Original Source Code
MORE
Source Code Parser and Analyser (SOOT)
A
B
Anti-patterns Detector
C
Design Patterns Detector
G
Search Process
(NSGA-II)
D
Software Quality Evaluator
E
List of possible refactorings
F
List of coherence constraints
R1 R2 R3 ... Rn
Suggested Refactoring Solution

Figure 3.18: Architecture of the MORE tool. [60]

that properties are not preserved between refactorings, and the second is that there is no way to avoid conflicts when two different refactorings are suggested for the same use case. The solution also does not support batch refactorings or integration with modern IDEs. The authors believe this model will significantly decrease development time and costs with these issues fixed.

Refactoring meta-models instead of directly transforming code may make the process easier for developers. They can abstract from the concrete implementation details and focus on the project's overall architecture. Mens et al. [54] explain that tools for supporting model refactoring to patterns must firmly ensure consistency among the different model views, traceability among components, and behavior preservation during transformations. Figure 3.22 presents an example of a model-driven software refactoring scenario.

The authors declare that model refactoring to patterns is not trivial and that there are practically no tools for supporting the process. One of the reasons for the difficulties experienced while transforming models is that it is hard to specify a high-quality model and which metrics may be adequate to signal a potential refactoring opportunity. Even if Kerievsky [45] presents suggestions for refactoring to patterns, it is not assured that these will translate well when refactoring its respective abstract models without any modifications. Moreover, it is hard to synchronize and ensure model consistency when users manually edit code generated from the models so that the

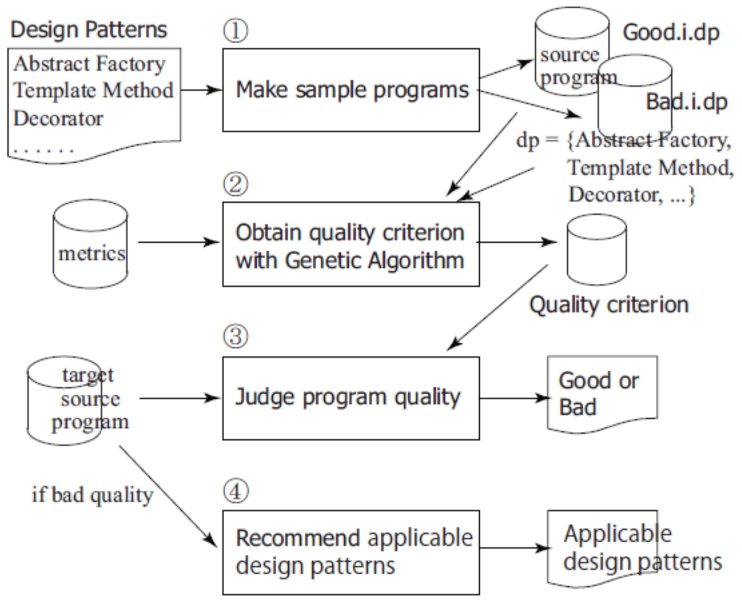


Figure 3.19: Architecture of the tool developed by Shimomura. [71]

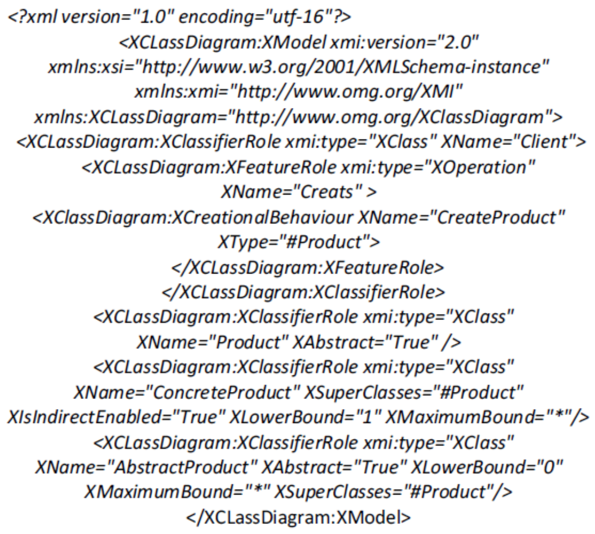

```
<?xml version="1.0" encoding="utf-16"?>
<XCLassDiagram:XModel xmi:version="2.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:XCLassDiagram="http://www.omg.org/XClassDiagram">
<XCLassDiagram:XClassifierRole xmi:type="XClass" XName="Client">
<XCLassDiagram:XFeatureRole xmi:type="XOperation"
XName="Creats" >
<XClassDiagram:XCreationalBehaviour XName="CreateProduct"
XType="#Product">
</XCLassDiagram:XFeatureRole>
</XCLassDiagram:XClassifierRole>
<XCLassDiagram:XClassifierRole xmi:type="XClass"
XName="Product" XAbstract="True" />
<XCLassDiagram:XClassifierRole xmi:type="XClass"
XName="ConcreteProduct" XSuperClasses="#Product"
XIsIndirectEnabled="True" XLowerBound="1" XMaximumBound="*"/>
<XCLassDiagram:XClassifierRole xmi:type="XClass"
XName="AbstractProduct" XAbstract="True" XLowerBound="0"
XMaximumBound="*" XSuperClasses="#Product"/>
</XCLassDiagram:XModel>
```

Figure 3.20: A pattern described in XML. [27]

refactoring better suits a specific need. Modeling languages such as UML may also have semantic variations from model to model, impacting the tool's assurance of behavior preservation.

The modeling language used for describing refactorings, patterns, and transformations must be generic enough to ensure flexibility for all problem domains. Tool implementations are bound to have good enough performance to be helpful in real-world scenarios. Such tools could greatly facilitate the refactoring creation and application process and chaining more minor refactorings into more complex ones. These refactorings could be orchestrated by interacting with the model as a graph, where operations would be characterized through graph transformation rules. Also, existing graph theory knowledge may help explain the created batch's parallel, sequential, and

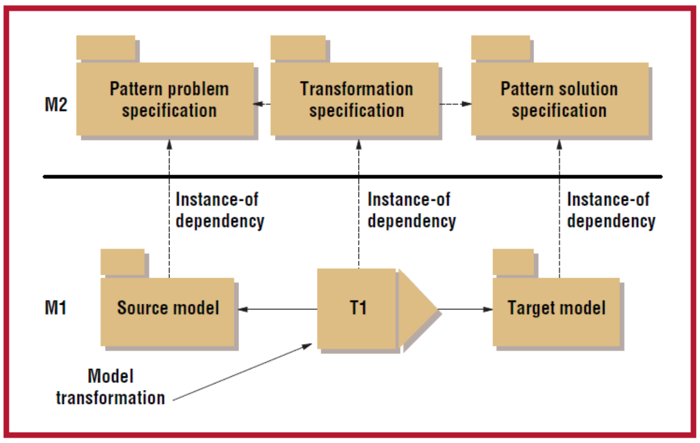


Figure 3.21: The architecture of France's meta-modelling approach. [34]

termination characteristics.

Tokuda and Batory [76] argue that redesigning software interfaces should become more interactive and automatic. The authors propose a new methodology for evolving designs based on editing class diagrams, which are abstractions of the code implementation, as seen in Figure 3.23. These abstractions allow the application of generic refactorings, which are not dependent on the implementation but operate over said abstractions. Doing so could reduce the burden of identifying where in the code the refactorings can be applied, fixing bugs introduced through manual refactorings, and testing changes.

The authors identify three operations for evolving object-oriented designs: applying design patterns, performing schema transformations, and identifying hot-spot meta-patterns for framework evolution. The developed tool covers all these use-cases and confirms that automated refactorings could be very effective in large-scale systems, especially when ensuring behavior preservation. Ensuring behavior preservation can help reduce manual validation efforts. However, the authors identified a set of transformations that could not be performed automatically, which is inherently limiting. The tool also cannot integrate with IDEs to support features such as hot-reloading code transformations when applying refactorings.

3.3.3 Discussion

Multiple tools are presented that implement one or more of the batch refactoring responsibilities that were previously stated. Refactoring identification approaches based on code clone detection, logic meta-programming, inference rules, and transformation languages are presented. Concerning transformation techniques, tools provide visual interaction mechanisms such as code wizards with step-by-step guides, user interfaces for composing batch refactorings, and a refactoring suggestion list directly in the IDE. Most tools try to ensure invariant preservation, albeit in different ways.

Regarding Live Refactoring capabilities, not many are compatible with development environments, and when they are, they lack features that other tools possess. IDEs are not good enough

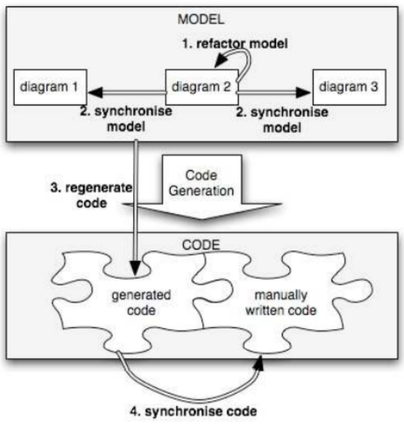


Figure 3.22: A sample scenario for model-driven refactoring. [54]

in this area, and their toolkits are hard to use. Environment integration is also primarily focused on the Eclipse platform, which suggests that other environments such as IntelliJ did not have research performed to implement a batch refactoring creation and application system. Moreover, some IDEs offer batch refactoring extensions that have incorrect algorithms and do not respect invariants, and extensibility features are challenging to use. However, some tools can integrate well into development environments, but these do not display complex batch refactoring capabilities.

Novel techniques are being used to develop innovative batch refactoring tools. Some of these are very robust, performing refactoring detection, suggestion, code generation, and even using code metrics to assess the quality of suggested refactorings. However, these are not integrated with development environments. Approaches that follow a model-driven philosophy emphasize creating, orchestrating, and customizing batch refactorings via interacting with higher-level abstractions such as UML models or graphs. Again, these tools' interaction capabilities with IDEs are limited, if not non-existing. Figure 3.25 summarizes the capabilities of each described tool.

Batch refactoring tools can aid the process of evolving code through source-to-source transformation tools or meta-modeling approaches. By abstracting patterns into reusable modules and then orchestrating them, the refactoring tool creation experience becomes more straightforward and independent of underlying code generation engines. Languages that can achieve this level of abstraction may drastically change the way refactoring tools are created.

While the architectural foundations of batch refactorings seem solid, most implementations are insufficient to provide a good development experience to IDE users and refactoring creators. Tools advance concrete issues of the batch refactoring creation process but are mostly incomplete. This, allied with the required knowledge necessary to create refactorings, may be pushing away practitioners from creating refactorings even more.

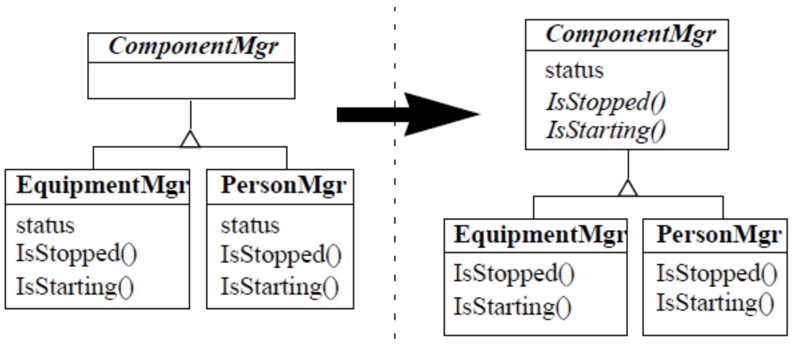


Figure 3.23: Applying a refactoring through a model-driven approach. [76]

3.4 Framework Evolution

Some of the most useful applications of refactoring to patterns techniques are related to framework evolution [77]. Change often triggers modifications in frameworks, and as software developers update their systems, they must alter existing framework code to ensure that features remain working as expected. A tool that enables refactoring to patterns must, in turn, implement a framework for representing the patterns that will be supported. It must be carefully designed so that it is expandable and maintainable.

3.4.1 Proposed Ideas

Roberts and Johnson [67] define frameworks as a set of abstract classes and their relationships that are part of a software system and constitute a reusable design. The authors recommend only developing a framework when there are enough applications to use a set of repeatable patterns. It is suggested that framework development should start only when there is a set of at least three applications that can be analyzed to identify patterns. The more examples, the more generalizable the framework can be, but too many examples can make it harder to finish developing it. Either way, examples are mandatory, as it is challenging to develop a framework based on abstract ideas.

The framework sections that are passable of being extended and updated are named hot-spots, and these should be correctly documented so that functionality remains unaltered [77]. Moreover, framework evolution may be more safely performed by specifying the transformations that have occurred since prior versions via meta-programming techniques and then be automatically applied to production code. Meta-patterns also have the advantage of allowing the re-usability of common patterns repeated across different framework transformations. In this case, meta-patterns are powered with information to match the concrete implementation code sections representing framework hot-spots. Not only does the proposed approach help propagate changes across framework versions, but it also allows for detecting existing upgrade conflicts. These characteristics reduce the chances of introducing bugs when manually upgrading frameworks and dismiss developers from performing this tedious and possibly damaging activity.

The TXL source transformation language, for instance, is one example of how these techniques can be used for framework development and evolution [21]. One of its special applications is meta-programming, which is achieved through annotating automatically generated code according to a TXL template and specifying batch code refactorings. This specification is achieved by defining sequences of refactorings in TXL that are used to transform source code. One of the most beneficial use cases of this source transformation language for defining refactorings is identifying maintenance hot spots in code.

For instance, when the Y2K bug hit the world, TXL was used for automatically identifying and converting affected code. A source transformation script is shown in Figure 3.24. This technique can also be used for maintaining frameworks over time. Practitioners can specify the source code blocks to be transformed and the necessary rules. The TXL processor automatically converts the code.

```
rule transformLessThanYYMMDD
   replace $ [repeat statement]
       IF {DATE-INEQUALITY-YYMMDD
              LeftOperand [name] < RightOperand [name]
         }DATE-INEQUALITY-YYMMDD
         ThenStatements [repeat statement]
         OptElse [opt else_clause]
       END-IF
       MoreStatements [repeat statement]

   construct RolledLeftOperand [name]
      LeftOperand [appendName "-ROLLED"]

   construct RolledRightOperand [name]
      RightOperand [appendName "-ROLLED"]

   by
       {TRANSFORM-INSERTED-CODE
       ADD LeftOperand ROLLDIFF-YYMMDD GIVING RolledLeftOperand
       ADD LeftOperand ROLLDIFF-YYMMDD GIVING RolledRightOperand
       }TRANSFORM-INSERTED-CODE
       IF {TRANSFORMED-DATE-INEQUALITY-YYMMDD
             RolledLeftOperand < RolledRightOperand
         }TRANSFORMED-DATE-INEQUALITY-YYMMDD
         ThenStatements
         OptElse
       END-IF
       MoreStatements
end rule
```

Figure 3.24: Source transformation for fixing the Y2K bug using TXL. [21]

Tourwé and Mens [77] propose a model for defining meta-patterns and instances of concrete implementations of those patterns. Additionally, meta-pattern transformations are defined for describing how to transition from one framework version state to another. Using meta-patterns makes the concrete implementations developed by practitioners independent of the overall abstract model of the patterns while ensuring that all constraints and invariants are preserved.

This model was implemented using a tool that supports the SOUL logic meta-programming environment for describing said patterns. Users must annotate code to know the used patterns, which are then listed to the practitioner. The tool also guides the user during the process, providing support for manual design evolution, detecting conflicts between versions, and filling hot spots automatically whenever possible.

3.4.2 Discussion

Frameworks are essential when developing large-scale software that can share components between different programs. To do this, patterns should first be identified in existing systems, which then can be generalized.

Refactoring code bases is a way to ensure that their evolution and maintainability happen while maintaining code clean, modular, and adaptable. This is done by specifying refactoring patterns that identify and transform critical components of a program that must be updated, as the transformation of vast blocks of code can be done automatically.

The way developers specify the transformations they wish to apply must be simple and accessible. While approaches based on source-to-source transformation languages, such as TXL, are compelling and customizable, developers must learn a different language. Many may not have the time or mindset to do so.

Framework evolution may greatly benefit from the model-based refactoring techniques mentioned in Section 3.3. Good enough abstraction of refactoring patterns can, for instance, allow the evolution of frameworks by scripting the changes in a domain-specific language. This may tremendously reduce the time and effort developers spend upgrading framework components, as most of these changes could be programmed and applied automatically through such abstractions.

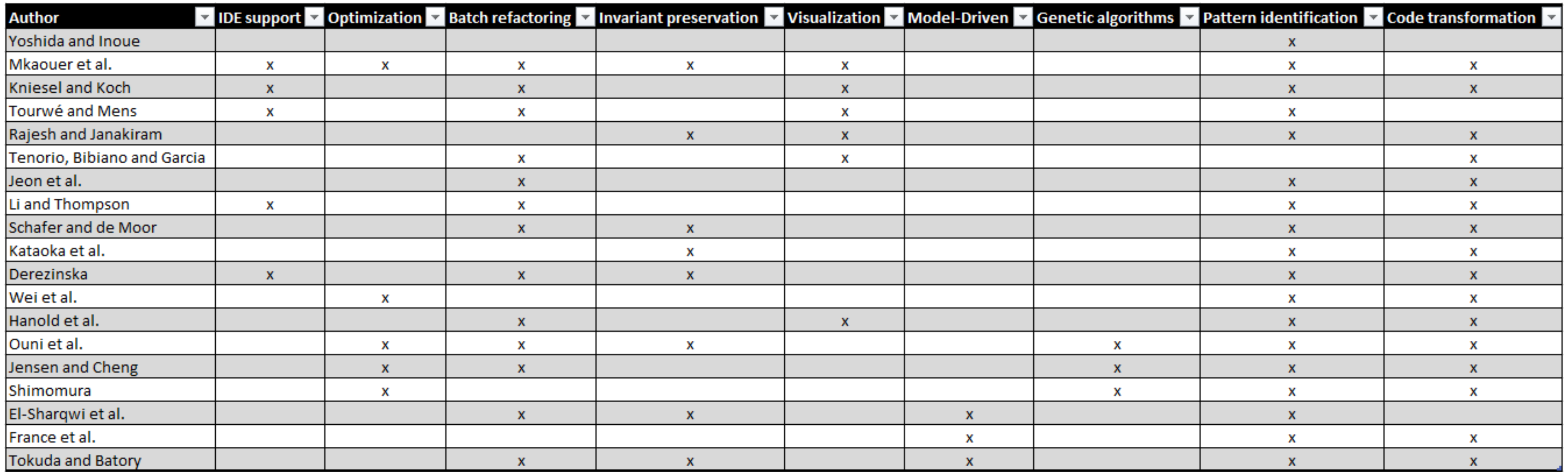
In general, evolving frameworks by refactoring to patterns is an exciting methodology for increasing developer productivity, reducing overall company costs, and adding security and speed to the process.

3.5 Summary

In this section, state of the art on batch refactoring architectures and formal models were presented, primarily based on splitting patterns into multiple ones and ensuring these individual ones preserve their invariants.

Additionally, multiple refactoring to patterns tools and methodologies were presented, along with explanations of their inner workings. Among them, novel techniques such as genetic algorithms and model-driven software development approaches were identified.

Finally, refactoring to patterns techniques for ensuring framework evolution and maintainability were identified, and the impact they may have on software development.



Author	IDE support	Optimization	Batch refactoring	Invariant preservation	Visualization	Model-Driven	Genetic algorithms	Pattern identification	Code transformation
Yoshida and Inoue								x	
Mkaouer et al.	x	x	x	x	x			x	x
Kniesel and Koch	x		x		x			x	x
Tourwé and Mens	x		x		x			x	
Rajesh and Janakiram				x	x			x	x
Tenorio, Bibiano and Garcia			x		x				x
Jeon et al.			x					x	x
Li and Thompson	x		x					x	x
Schafer and de Moor			x	x				x	x
Kataoka et al.				x				x	x
Derezinska	x		x	x				x	x
Wei et al.		x						x	x
Hanold et al.			x		x			x	x
Ouni et al.		x	x	x			x	x	x
Jensen and Cheng		x	x				x	x	x
Shimomura		x					x	x	x
El-Sharqwi et al.			x	x		x		x	
France et al.						x		x	x
Tokuda and Batory			x	x		x		x	x

Figure 3.25: Summary of refactoring tools capabilities.

Chapter 4

Problem Statement

Change is the only constant in the world, which naturally applies to the software engineering reality. Change affects software projects in multiple ways, such as teams increasing in size, stakeholders changing their minds, competition getting more robust, tight deadlines, requirements that are often modified, and the introduction of new technologies [18].

The effects of change can also be felt due to the behavior of software developers. They may overlook how implementation details affect requirements satisfiability and may not know that the code has bad smells. The costs are clear, and one of the most obvious ones is the steep decrease in productivity [52].

4.1 Context

Given that code is one of the most concrete expressions of requirements, it should be as resilient to change as possible. The problem is that constant change may make teams not pay much attention to ensuring that the code is robust, strong, modular, and adaptable, as the limited time of each developer may instead be allocated to develop yet another feature. This is tempting, as time is money, and features must be added to keep stakeholders happy. The issue is accentuated by the swift development pace that envelops the industry.

Multiple techniques are used to ensure that code is kept in a good, clean and robust state. Refactoring is one of them, as well as the application of design patterns. The fact is that design patterns and refactorings are intrinsically related: patterns are implemented to improve design, and applying a pattern is performing a transformation that does not alter the code's behavior but improves its design, which is the definition of refactoring.

Therefore, refactorings should be applied often. Multiple tools offer Live Refactoring capabilities such as automatic suggestion, application, and preview of refactorings, which can significantly help practitioners, as demonstrated in previous chapters. However, creating refactoring plugins still leaves much to be desired.

4.2 Open Issues

Unfortunately, some issues prevent the creation of refactoring plugins in an accessible and straightforward manner. In Chapter 3, state of the art on batch refactoring, refactoring tools and techniques, and framework evolution revealed some deficiencies with solutions to the creation of refactoring tools. Along with other problems identified in Chapter 1, the following open issues can be exposed:

High Amount of Pre-Required Knowledge. Creating refactorings from scratch is not a trivial task, as it requires knowledge in the most varied areas, such as compilers, design patterns, refactorings, language design, IDE and compiler APIs, and general software engineering experience. Not all developers are comfortable with all of these areas, requiring them to get a grasp on them before implementing something. This large, pre-required body of knowledge may detract them from even considering building refactoring tools of their own.

Large Time Expenditure. The topics mentioned above, especially compilers and language design, are not trivial to understand, requiring considerable learning time. This is time that many practitioners cannot spend, decreasing the likelihood they will ever create a refactoring tool.

Varying Compiler and IDE APIs. Each compiler and each IDE have its APIs. Therefore, if practitioners want to build refactoring tools targeting multiple languages and APIs, they must study the inner workings of them all. Even if they are similar, they will indeed have their peculiarities, and such complex pieces of software will have design differences that require study.

Not Accessible to the Ordinary Developer. Not all developers will have the skills necessary to build refactoring tools, reducing the chances of these being implemented. If the process was more straightforward, requiring only the developer to know the behaviors the refactoring should have, junior developers (and perhaps non-developers) could create these tools. Additionally, practitioners are not comfortable with existing refactoring creation tools.

Ever-Evolving APIs. APIs change over time and many are even dropped or replaced. This forces practitioners to constantly update their tools to comply with newer software versions. This effort could, instead, be spent on improving existing refactoring tools or creating new ones.

Limited Modularity and Reusability. As practitioners implement refactoring tools by hand, some code will likely get duplicated, as refactoring tool algorithms may share common behaviors. It would be helpful if practitioners could reuse these behaviors as they develop new tools. Moreover, creators could use already existing behaviors by simply including them in the tool they are creating, reducing the amount of code needed to be written by hand. Additionally, large portions of boilerplate must be repeatedly rewritten to implement basic functionality. Unfortunately, this is not yet a reality and may discourage practitioners from creating new refactoring plugins.

Limited Refactoring Tool Creation Support from IDEs. Modern IDEs such as Eclipse, Visual Studio, and IntelliJ do not provide support for quickly creating complex refactorings. In the case of Visual Studio, it provides users with a template for creating a refactoring tool. However, it requires users to interact with compiler and IDE APIs, which, as previously mentioned, are not accessible to most practitioners.

Limited Live Refactoring Support. Most complex refactoring creation tools do not integrate well - if at all - with modern development environments such as Visual Studio.

Some Existing Refactoring Tools Are Flawed. Some of the existing tools for creating complex refactorings have algorithmic errors. Therefore, these tools are not providing refactorings but inaccurate code transformations, as they are not improving the design of existing code nor maintaining original functionality.

Limited Refactoring Availability in IDEs. Most IDEs only provide support for the most popular refactorings and do not cater to more complex ones. It is safe to say that it is implausible that IDE manufacturers will develop plugins for niche refactorings in the foreseeable future.

4.3 Hypothesis and Research Questions

Chapter 3 helped identify current practices regarding creating complex refactoring tools. The information in that chapter revealed that the patterns and mini-patterns paradigm were thoroughly considered among academics. This technique can be used not only for the programming activity but even for the code review process. Additionally, it was recognized that IDEs are not good enough in supporting the creation of batch refactorings, lacking, for instance, accessible toolkits. However, they provide useful Live Refactoring features that are much useful to practitioners.

The process of abstracting refactoring behaviors and isolating them into concrete modules was revealed to be helpful, as well as the process of orchestrating said modules. This orchestration could be done visually or by refactoring behavior scripting languages, which are not always clear and accessible enough. Moreover, this kind of language that abstracts concrete implementation details of creating complex refactorings seems to be very fruitful when applied to framework evolution. Changes can be scripted using such a language, and frameworks can be automatically updated.

When the issues above regarding the creation of refactoring plugins are examined, it becomes clear that there is much potential to improve the creation process, further incentivizing practitioners to build their own.

The following hypothesis can, thus, be declared:

A tool for building complex refactoring plugins via behavior orchestration, using Live Refactoring environment features, can incentivize the creation of such plugins and make the development process more accessible and simple.

In order to confirm or reject this hypothesis, the next set of research questions can be raised in order to better direct the quest for validating the hypothesis:

RQ1: Can a tool for generating refactoring plugins make the plugin development process more accessible and straightforward?

RQ2: Will practitioners benefit from plugins with built-in Live Refactoring features such as live previews, automatic refactoring opportunity detection, and one-click refactoring application?

RQ3: Will a refactoring plugin generator incentivize practitioners to build refactoring plugins of their own?

4.4 Proposal

Considering the previously described research questions and open issues, the dissertation will focus on developing a tool for creating refactoring plugins. This tool should improve the creation of refactoring plugins by making it more accessible, not requiring large amounts of knowledge on many topics. Additionally, users of the tool should be able to get results quickly. The tool's interface should be stable and independent of the concrete implementation so that users do not need to be constantly relearning it and to allow it to support multiple programming languages. Finally, it should integrate with modern development environments and use their Live Refactoring capabilities.

4.5 Validation

An expert assessment with software engineering professionals will be performed to validate the hypothesis and provide answers to the research questions. The participants, who ideally will have different years of experience and academic formation, will be required to use the tool for building several refactoring plugins of varying complexity. The experiments will be timed so that it is possible to assess to which extent the developed tool has improved the creation process. These results will be collected through a survey filled by participants at the end of the experiment. The validation methodology will be further described in Chapter 6.

Chapter 5

Proposed Solution

A potential solution would be to abstract the creation process to a point where the developer would only need to focus on chaining the behaviors needed to implement a particular refactoring, resulting in the generation of a refactoring plugin ready to be connected to an IDE. This approach takes cues from the research in Chapter 3. A possible abstraction could take the form of a simple language that could generate refactoring plugins by chaining behaviors together in the shape of reusable modules. When chained, these would allow the creation of both simple and batch refactorings.

Therefore, RPCL - Refactoring Plugin Creation Language - was developed for building custom refactoring plugins. When connected with the Visual Studio IDE, these plugins can use its Live Refactoring capabilities to suggest, preview, and transform code according to the refactoring algorithm. The language aims to democratize the refactoring plugin creation process, making it accessible to every developer by lowering the knowledge entry barrier. This is achieved by dispensing the user from the need to grasp complex knowledge in multiple programming areas. The ultimate goal is to improve code quality and the overall practitioner's development experience.

5.1 Live Refactoring Features

As mentioned in Chapter 1, while refactorings are helpful, they are only valuable if applied. Therefore, even if there is a language capable of generating refactoring plugins, if, when used, the tool cannot offer refactoring suggestions in a quick, straightforward, and obvious way, it will not be advantageous. With this in mind, RPCL was designed to be able to generate plugins with multiple Live Refactoring capabilities built-in and ready to be used when plugged into the IDE. The available Live Refactoring features are the following:

Automatic Detection of Refactoring Opportunities. Refactorings resultant from using a refactoring plugin that was generated with RPCL will alert users of refactoring opportunities that enhance code quality and maintainability. This is done by triggering an automatic suggestion whenever the cursor hovers an expression of the kind that is explicit in the RPCL script. For instance, it makes sense only to trigger a refactoring that adds a parameter to a method declaration if the user's cursor is hovering over a method declaration - provided it complies with the conditions defined by the practitioner when creating the refactoring plugin. The suggestion is offered through the appearance of a light-bulb contextual menu. This menu lists all available refactorings for that expression type when clicked. An example is available in Figure 5.5.

Live Previews. This feature, also built into all RPCL-generated plugins, allows users to visualize how the refactoring, when applied, will transform the source code. These previews are color-coded to effectively separate which segments will be added (highlighted in green) and removed (highlighted in red). Respecting the Live Refactoring philosophy, the previews are dynamic. This means that the code previews are applied to code in real-time - these are not renders of concrete situations but representations of how the refactoring will transform the code. Figure 5.5 is an example of the refactoring preview feature.

Automatic Application of Refactorings. While detecting and previewing refactorings is useful, applying refactoring transformations is no less convenient. Looking again at Figure 5.5, when clicking on a line of the live contextual menu that appears whenever it is appropriate, the refactoring is automatically applied. It is also possible to revert the refactoring application automatically, modifying the code to the state the code was immediately before applying it. Ultimately, the application of refactorings is simply another text edit operation. This reversion operation can be performed, for instance, through the CTRL + Z keyboard shortcut or the Edit menu in the Visual Studio menu bar.

5.2 Technological Components

RPCL is a DSL built on top of the C# programming language. It is an internal DSL, as opposed to an external one, to take advantage of the practitioners' familiarity with the language. The language's syntax is valid C# code. This means that the refactoring plugins can be written in a C# file, provided that RPCL's library, which contains all the available modules, is imported. Because this is an internal DSL, there is no need to parse the language, as it is simply a C# script. Thus, the refactoring plugin can be generated by simply compiling and running the code. In the Visual Studio IDE, this process is as easy as pressing the Play button in the menu bar.

Using C# as the base language provides multiple advantages. One of them is having access to the .NET Platform Compiler SDK, which provides powerful meta-programming manipulation tools. Being a compelling platform that enables developers to interact with multiple C# compiler APIs, Roslyn lets users manipulate ASTs by adding, transforming, and deleting nodes. RPCL leverages C#'s syntax features and builds on top of them.

Additionally, it is an object-oriented language. This should ease the developers' learning experience as it is a paradigm that most software engineers are familiar with. It is also a popular language [10], which also contributes to easing the learning journey. Finally, it is easy to read and charged with features such as method chaining, making it easier to combine multiple commands in a logical sequence and successively use the results of previous operations.

Internally, Roslyn stores code structures in the form of an AST. Nodes are used to store the implementation of classes, methods, and expressions, among others [81]. Each syntax node has a kind that identifies it, as each C# construct is composed of a particular set of expressions and tokens. For instance, an if-statement expects a boolean condition expression and a set of body statements. Optionally, there can be an else-statement, which also expects a body of statements. These are surrounded by the appropriate tokens, such as curly braces and parenthesis. A list of some of these node kinds is present in Figure 5.1. Figure 5.2 showcases Syntax Visualizer, a Visual Studio extension that allows visualizing how different syntax nodes can be nested together to form a program.

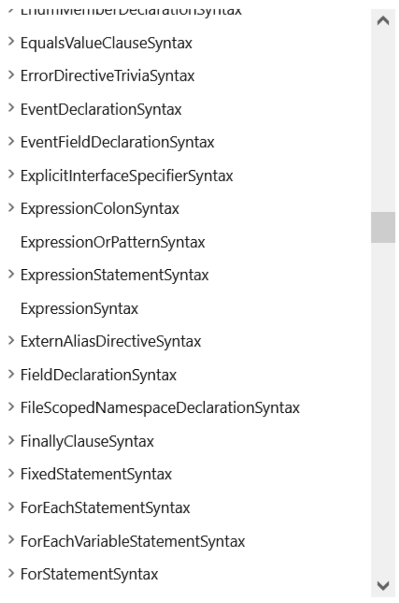
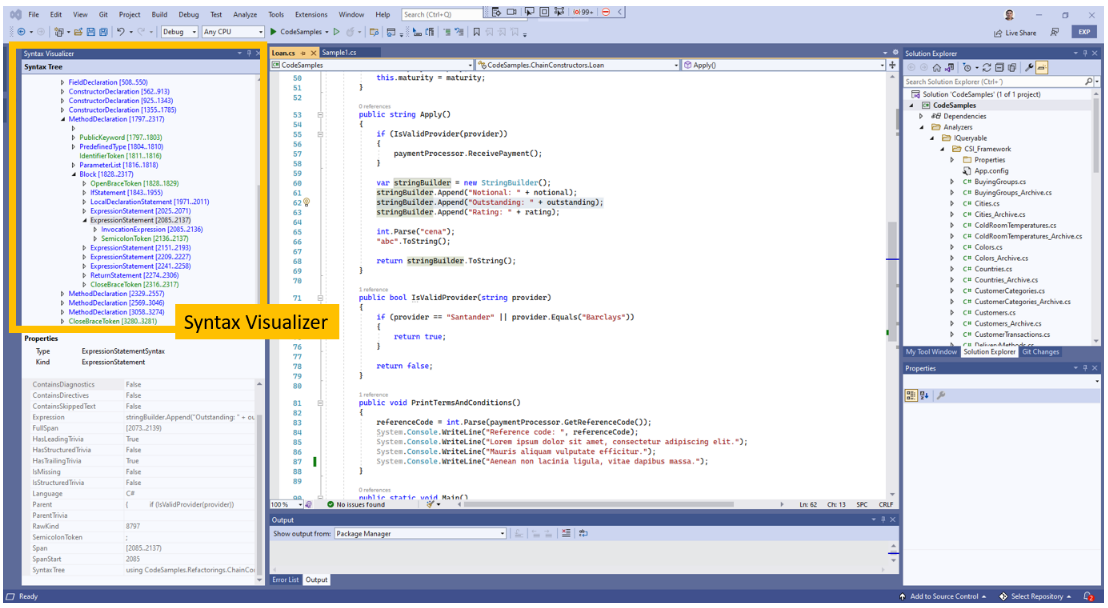

EnumMemberDeclarationSyntax
EqualsValueClauseSyntax
ErrorDirectiveTriviaSyntax
EventDeclarationSyntax
EventFieldDeclarationSyntax
ExplicitInterfaceSpecifierSyntax
ExpressionColonSyntax
ExpressionOrPatternSyntax
ExpressionStatementSyntax
ExpressionSyntax
ExternAliasDirectiveSyntax
FieldDeclarationSyntax
FileScopedNamespaceDeclarationSyntax
FinallyClauseSyntax
FixedStatementSyntax
ForEachStatementSyntax
ForEachVariableStatementSyntax
ForStatementSyntax

Figure 5.1: List of some of Roslyn's syntax node kinds. [7]

Given that Roslyn was selected as the compiler platform that powers RPCL, it was a natural decision to pick Visual Studio 2022 as the development IDE so that the refactoring plugins can take advantage of its Live Refactoring capabilities. Visual Studio provides multiple APIs for all of the Live Refactoring features described in Section 5.1. These are independent of the behavior of the refactoring plugin and are automatically added to it. Moreover, the IDE supports the display of contextual menus such as the light bulb indicators and messages in a specific application pane whenever the developer finds it appropriate.



```
FieldDeclaration [508..550)
ConstructorDeclaration [562..913)
ConstructorDeclaration [925..1343)
ConstructorDeclaration [1355..1785)
MethodDeclaration [1797..2317)
    PublicKeyword [1797..1803)
    PredefinedType [1804..1810)
    IdentifierToken [1811..1816)
    ParameterList [1816..1818)
    Block [1828..2317)
        OpenBraceToken [1828..1829)
        IfStatement [1843..1955)
        LocalDeclarationStatement [1971..2011)
        ExpressionStatement [2025..2071)
        ExpressionStatement [2085..2137)
            InvocationExpression [2085..2136)
            SemicolonToken [2136..2137)
        ExpressionStatement [2151..2193)
        ExpressionStatement [2209..2227)
        ExpressionStatement [2241..2258)
        ReturnStatement [2274..2306)
        CloseBraceToken [2316..2317)
MethodDeclaration [2329..2557)
MethodDeclaration [2569..3046)
MethodDeclaration [3058..3274)
CloseBraceToken [3280..3281)
```

Properties

Type	ExpressionStatementSyntax
Kind	ExpressionStatement

ContainsDiagnostics	False
ContainsDirectives	False
ContainsSkippedText	False
Expression	stringBuilder.Append("Outstanding: " + ou
FullSpan	[2073..2139)
HasLeadingTrivia	True
HasStructuredTrivia	False
HasTrailingTrivia	True
IsMissing	False
IsStructuredTrivia	False
Language	C#
Parent	{ if (IsValidProvider(provider))
ParentTrivia	
RawKind	8797
SemicolonToken	;
Span	[2085..2137)
SpanStart	2085
SyntaxTree	using CodeSamples.Refactorings.ChainCo

```csharp
        this.maturity = maturity;
    }

    0 references
    public string Apply()
    {
        if (IsValidProvider(provider))
        {
            paymentProcessor.ReceivePayment();
        }

        var stringBuilder = new StringBuilder();
        stringBuilder.Append("Notional: " + notional);
        stringBuilder.Append("Outstanding: " + outstanding);
        stringBuilder.Append("Rating: " + rating);

        int.Parse("cena");
        "abc".ToString();

        return stringBuilder.ToString();
    }

    1 reference
    public bool IsValidProvider(string provider)
    {
        if (provider == "Santander" || provider.Equals("Barclays"))
        {
            return true;
        }

        return false;
    }

    1 reference
    public void PrintTermsAndConditions()
    {
        referenceCode = int.Parse(paymentProcessor.GetReferenceCode());
        System.Console.WriteLine("Reference code: ", referenceCode);
        System.Console.WriteLine("Lorem ipsum dolor sit amet, consectetur adipiscing elit.");
        System.Console.WriteLine("Mauris aliquam vulputate efficitur.");
        System.Console.WriteLine("Aenean non lacinia ligula, vitae dapibus massa.");
    }

    0 references
    public static void Main()
```

```
Solution 'CodeSamples' (1 of 1 project)
  CodeSamples
    Dependencies
    Analyzers
      IQueryable
        CSI_Framework
          Properties
          App.config
          BuyingGroups.cs
          BuyingGroups_Archive.cs
          Cities.cs
          Cities_Archive.cs
          ColdRoomTemperatures.cs
          ColdRoomTemperatures_Archive.cs
          Colors.cs
          Colors_Archive.cs
          Countries.cs
          Countries_Archive.cs
          CustomerCategories.cs
          CustomerCategories_Archive.cs
          Customers.cs
          Customers_Archive.cs
          CustomerTransactions.cs
```

Figure 5.2: Visual Studio's Syntax Visualizer extension.

5.3 Language Design

Creating a language is a process that must receive enough attention and ponderation. The language should be flexible enough but not to a point where it loses its intended focus, in this case, building refactoring plugins. A well-thought language induces harmony when used without limiting the user's freedom and creativity. The following sub-sections go over the thought process of creating RPCL.

5.3.1 Behavior Modules

The language's building blocks are its modules, representing the behaviors that characterize a refactoring. Modules may be either conditional or transformational. Conditional modules are preconditions that must be verified for the refactoring to be suggested and applied. These module names begin with the "If" word. On the other hand, transformational modules define the code manipulations that the refactoring will perform on existing code, such as adding or removing specific code segments. The names of these do not begin with any specific word.

Internally, each module is a C# method. These take advantage of the Roslyn APIs for interacting with a program's AST. Possible interactions include searching for methods with a specific name, variables of a particular type, and expressions of various kinds, such as invocations and binary expressions. Each module was named so that it becomes immediately apparent to the creator what its behavior is.

An essential characteristic of behavior modules worthy of mention is that each one is atomic, meaning they work independently. However, this does not mean they do not have code in common. For instance, one module may require the construction and addition of a method declaration necessary for the functioning of another module. In these cases, each module queries the AST to determine if that declaration is already present in the AST so that the tree can access it. If so, that declaration is used; if not, it is added to the AST.

Two special modules make up the entire set:

TriggerOnAll. Before invoking any other modules, there must be an invocation to this one, which specifies which kind of code declarations (such as classes, methods, or expressions) the refactoring plugin should alert the user that a refactoring opportunity is available. For instance, if the module targets method declarations, the refactoring will only be offered when the practitioner's cursor hovers over a method declaration where the conditional modules (or refactoring preconditions) are verified.

Assemble. This module should always be invoked after all behavioral modules are stated, as it is responsible for generating a file containing the generated refactoring plugin's code. If not used, the file's content can be accessed by assigning the result of the module chaining to a variable.

While not a module, the @Previous construct is also encompassed by special rules. This is to be used whenever the practitioner wants to reference an object previously referenced by another module. For instance, it may happen that, during the development of a refactoring plugin, the practitioner does not know beforehand the name that a variable will have when the plugin is used, but only its expected type. In these situations, the @Previous construct helps chain refactorings together and allows preconditions to be built on top of each other, exchanging information from the previous reference and sharing context. However, in cases where the variable's name is a constant, if the refactoring should only be applied for objects with that name, then referencing objects using the @Previous keyword would be unnecessary, and passing the variable name would suffice.

No one size fits all, and thus some modules may not completely cater to the needs of developers. With this in mind, the available refactoring plugin modules are customizable. Such customization is achieved via parametrization. This design decision aims to augment the flexibility of the modules so that they are not tied to the refactoring plugins they were extracted from and can be reused and adapted to other use cases.

With RPCL, no previous compiler knowledge and Roslyn and Visual Studio APIs are required to create new refactoring plugins. However, this is not true when creating new modules. Suppose there are enough built-in modules for most common preconditions and transformations. In that case, creating new ones should be less of a necessity. However, this need will inevitably arise. Practitioners may want to create new modules or tweak existing ones to comply with the intended design. In the latter case, they may tweak the generated plugin to adapt it accordingly, avoiding the need to create new modules, but this may, sometimes, be unavoidable. As the language evolves, the goal is for it to provide as many modules as possible to decrease the likelihood of being necessary to create new ones. All modules that are created should seamlessly integrate with the built-in ones.

5.3.2 Refactoring Plugin Creation Process

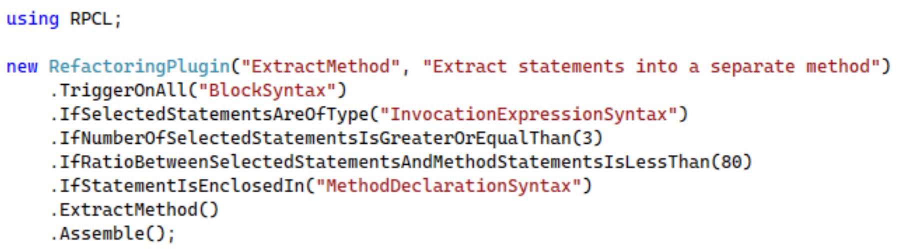
Refactoring plugins are built by declaring a new RefactoringPlugin object, which receives its name and description of what it will do. The user should chain the available modules together to define the refactoring behaviors from which the plugin will be based. The object also stores contextual information on behavior modules as they are chained. When executed, all behaviors return an updated RefactoringPlugin object.

The RefactoringPlugin class, from which new plugins are created, contains a blank AST that is progressively built as the behavioral modules are called. Internally, the class' constructor starts by automatically generating boilerplate code common to all refactoring plugins, such as the necessary blocks for integrating with the Visual Studio refactoring APIs and importing libraries required for any plugin to function. After that, each method is executed in the order it was declared (the natural flow of chaining methods together). When a method is executed, it generates the appropriate code by invoking Roslyn APIs to verify the preconditions and transform existing code. Each module's code is generated as syntax nodes, which are appended to the AST of the file that, at the end of the chain, will contain the concrete implementation of the refactoring plugin. This behavior is similar to the Pipes and Filters design pattern, characterized by the successive transformations of original data input as it passes the pipeline's filers, resulting in a refined output.

5.4 Abstraction Levels

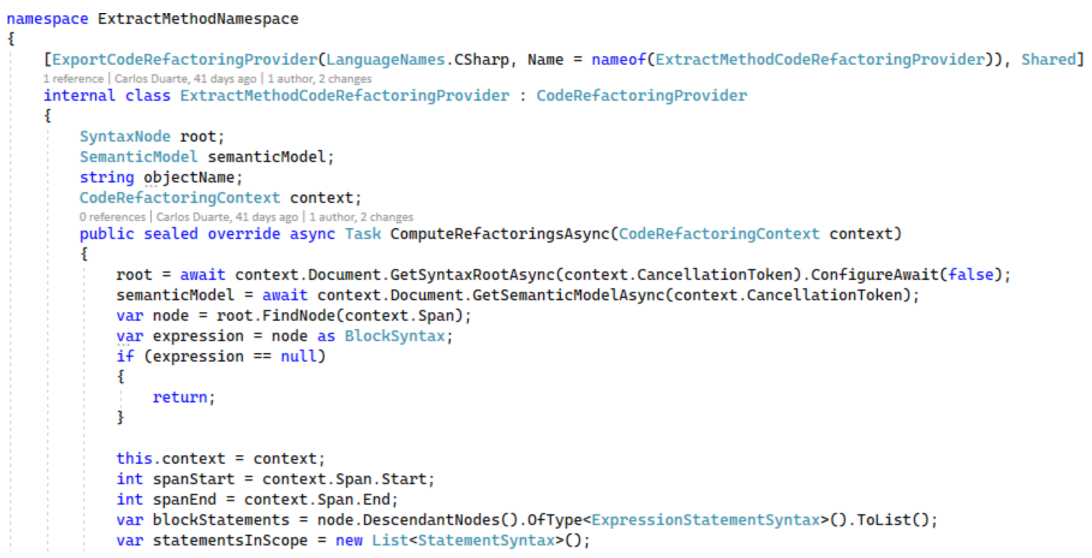
When dealing with meta-programming issues, it can quickly become confusing to identify at which abstraction level operations are being performed. In this case, three different abstraction levels can be identified. Figures 5.3, 5.4 and 5.5 provide concrete examples of these abstraction levels using the Extract Method refactoring plugin as an example, whose functioning is described later in Section 5.5. The first level, pictured in Figure 5.3, is where the refactoring plugin is created through the RPCL language. With it, multiple refactoring patterns are chained together to build refactoring plugins of varying complexities. The result of running this code corresponds to the second abstraction level, as seen in Figure 5.4. This level is represented by a file generated when compiling the RPCL script, containing a lot more advanced code that interacts with compiler and IDE APIs. It may not be intelligible to the common practitioner. It also contains the code responsible for providing the refactoring plugins with Live Refactoring functionality. This file can then be plugged into Visual Studio, which will provide the capabilities mentioned above to any C# project, allowing users to apply the refactorings that the plugin they built seamlessly provides. This corresponds the third abstraction level, explicit in Figure 5.4.

Naturally, there is a sequence of interactions between abstraction levels, with the first interacting with the second and the second interacting with the third. These interactions are, naturally, unidirectional. However, non-consecutive levels cannot interact with each other. It would not make sense to be otherwise, given that each level builds on top of the previous one. For instance, the third level can only provide Live Refactoring capabilities built into the first abstraction layer if an intermediate layer - the second one - transforms the RPCL specification into concrete code that performs the appropriate API calls.



```
using RPCL;

new RefactoringPlugin("ExtractMethod", "Extract statements into a separate method")
    .TriggerOnAll("BlockSyntax")
    .IfSelectedStatementsAreOfType("InvocationExpressionSyntax")
    .IfNumberOfSelectedStatementsIsGreaterOrEqualThan(3)
    .IfRatioBetweenSelectedStatementsAndMethodStatementsIsLessThan(80)
    .IfStatementIsEnclosedIn("MethodDeclarationSyntax")
    .ExtractMethod()
    .Assemble();
```

Figure 5.3: The first abstraction level. Corresponds to creating a refactoring plugin using the RPCL language, which is a composition of behavioural modules.



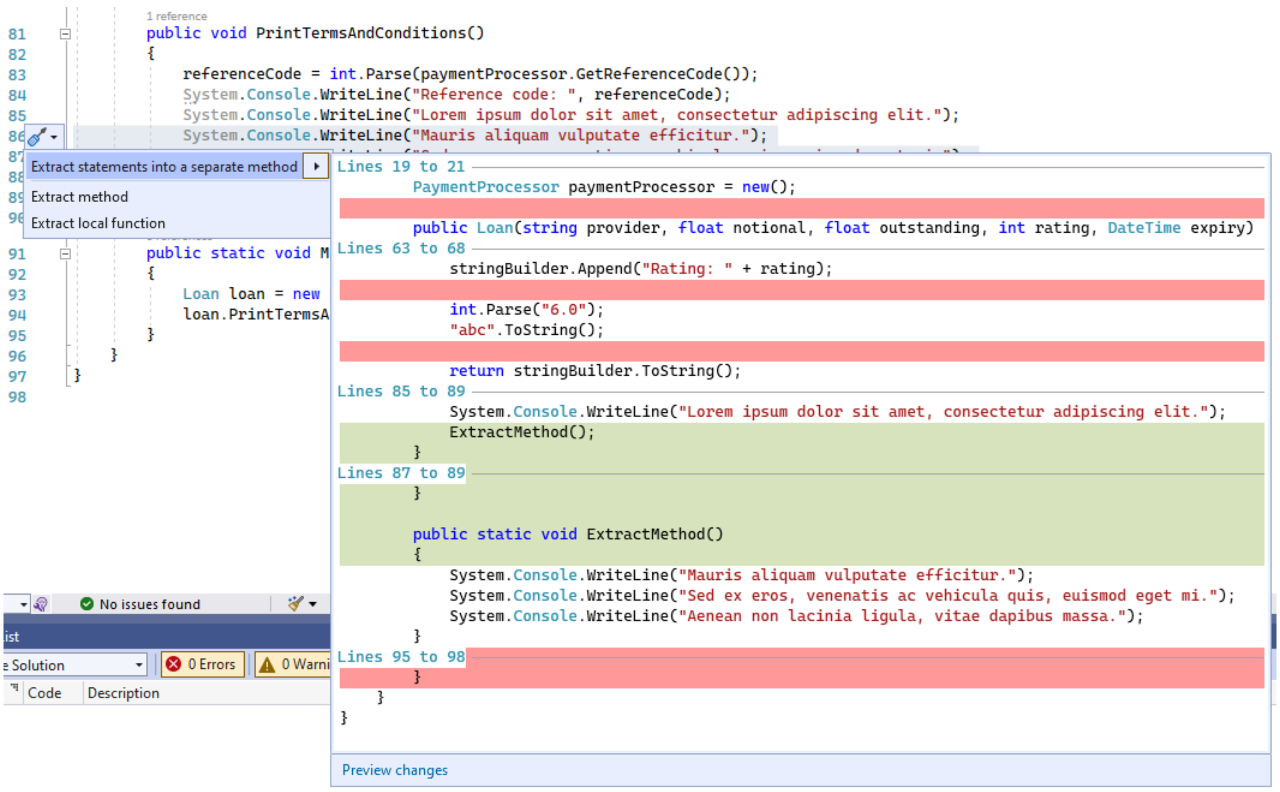
```
namespace ExtractMethodNamespace
{
    [ExportCodeRefactoringProvider(LanguageNames.CSharp, Name = nameof(ExtractMethodCodeRefactoringProvider)), Shared]
    1 reference | Carlos Duarte, 41 days ago | 1 author, 2 changes
    internal class ExtractMethodCodeRefactoringProvider : CodeRefactoringProvider
    {
        SyntaxNode root;
        SemanticModel semanticModel;
        string objectName;
        CodeRefactoringContext context;
        0 references | Carlos Duarte, 41 days ago | 1 author, 2 changes
        public sealed override async Task ComputeRefactoringsAsync(CodeRefactoringContext context)
        {
            root = await context.Document.GetSyntaxRootAsync(context.CancellationToken).ConfigureAwait(false);
            semanticModel = await context.Document.GetSemanticModelAsync(context.CancellationToken);
            var node = root.FindNode(context.Span);
            var expression = node as BlockSyntax;
            if (expression == null)
            {
                return;
            }

            this.context = context;
            int spanStart = context.Span.Start;
            int spanEnd = context.Span.End;
            var blockStatements = node.DescendantNodes().OfType<ExpressionStatementSyntax>().ToList();
            var statementsInScope = new List<StatementSyntax>();
```

Figure 5.4: The second abstraction level. Contains the generated refactoring code from the RPCL specification, making use of both Roslyn and Visual Studio APIs. The file is ready to be plugged into the IDE so that the refactoring is available to users automatically.

5.5 Implementing Refactoring Plugins with RPCL

In order to build refactoring plugins with RPCL, there must exist modules so they can be chained together. With this in mind, some refactoring plugins were created from scratch, that is, by manually using Roslyn and Visual Studio APIs to detect refactoring opportunities and transform source code. This was, naturally, not an easy task, as all issues identified with current refactoring plugin creation processes in Section 4.2 were present.

After implementing each refactoring plugin, the code blocks that conferred the plugin particular behaviors (such as detecting a method with a given name or a class that implements a



```
public void PrintTermsAndConditions()
{
    referenceCode = int.Parse(paymentProcessor.GetReferenceCode());
    System.Console.WriteLine("Reference code: ", referenceCode);
    System.Console.WriteLine("Lorem ipsum dolor sit amet, consectetur adipiscing elit.");
    System.Console.WriteLine("Mauris aliquam vulputate efficitur.");

public static void M
{
    Loan loan = new
    loan.PrintTermsA
}
    }
}
```

```
Lines 19 to 21
    PaymentProcessor paymentProcessor = new();

    public Loan(string provider, float notional, float outstanding, int rating, DateTime expiry)
Lines 63 to 68
        stringBuilder.Append("Rating: " + rating);

        int.Parse("6.0");
        "abc".ToString();

        return stringBuilder.ToString();
Lines 85 to 89
        System.Console.WriteLine("Lorem ipsum dolor sit amet, consectetur adipiscing elit.");
        ExtractMethod();
    }
Lines 87 to 89
    }

    public static void ExtractMethod()
    {
        System.Console.WriteLine("Mauris aliquam vulputate efficitur.");
        System.Console.WriteLine("Sed ex eros, venenatis ac vehicula quis, euismod eget mi.");
        System.Console.WriteLine("Aenean non lacinia ligula, vitae dapibus massa.");
    }
Lines 95 to 98
    }
  }
}
```

Figure 5.5: The third abstraction level. After plugging in the refactoring code into Visual Studio, it becomes available to be automatically suggested, previewed, and applied by users in whichever project they are working on, provided the necessary preconditions are met.

particular interface) were identified and isolated. Each gave origin to a behavior module. Naturally, the more refactoring plugins were implemented, the more behavior patterns were identified and, consequently, the more modules became available.

The implemented refactoring plugins revolve around the Chain Constructors, String Comparison, Extract Method, Ensure Dispose Call, Type Conversion, Creation Method, and Append ToList to IQueryable refactorings, whose descriptions will be presented next.

5.5.1 Chain Constructors

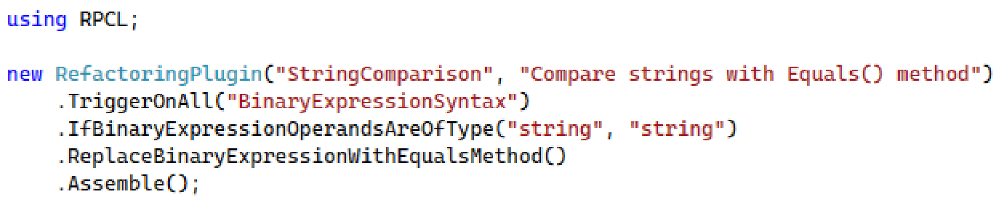
```
using RPCL;

new RefactoringPlugin("ChainConstructors", "Chain constructors")
    .TriggerOnAll("ConstructorDeclarationSyntax")
    .IfExistsCatchAllConstructor()
    .ChainConstructors()
    .Assemble();
```

Figure 5.6: Declaration of the Chain Constructors refactoring using RPCL.

One of the many refactorings described by [45] is Chain Constructors, a beneficial transformation to decrease duplicate code. The refactoring is triggered on constructor declarations whenever a catch-all constructor exists. A catch-all constructor can replace another's body by being called in place of the body statements. For instance, consider two constructors: the catch-all constructor and a simple one (a non-catch-all constructor). The simple one has, in its body, variable assignments. The catch-all constructor can receive, as parameters, all variables being assigned to the simple one and assigns them inside its own body. Therefore, the simple one can invoke the catch-all constructor with all the correct parameters to reduce duplicate code.

The code transformations may be applied if a catch-all constructor exists inside a class. These consist of creating a new constructor that receives, as parameters, all variables present in the body of all other constructors, which are initialized to some value. Then, all other constructors are stripped from their body statements, calling the catch-all constructor instead. The RPCL implementation of this refactoring can be seen in Figure 5.6.

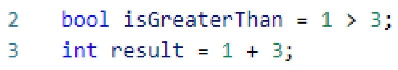
5.5.2 String Comparison



```
using RPCL;

new RefactoringPlugin("StringComparison", "Compare strings with Equals() method")
    .TriggerOnAll("BinaryExpressionSyntax")
    .IfBinaryExpressionOperandsAreOfType("string", "string")
    .ReplaceBinaryExpressionWithEqualsMethod()
    .Assemble();
```

Figure 5.7: Declaration of the String Comparison refactoring using RPCL.

The String Comparison refactoring aims to dissuade developers from comparing strings with the equality operator (==) and invoke the Equals() method instead. This can ensure that the exception can be dealt with accordingly if any of the operators is not of type string. The RPCL code that defines such refactoring is represented in Figure 5.7.



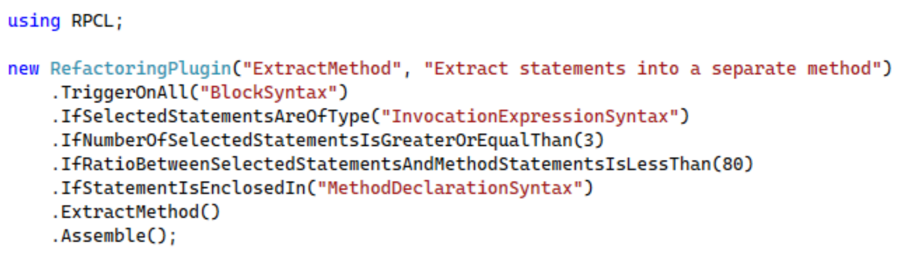
```
bool isGreaterThan = 1 > 3;
int result = 1 + 3;
```

Figure 5.8: Example of the Greater Than and Addition binary expressions in C#.

This refactoring should trigger on all binary expressions, which have an operator surrounded by two operands, such as the Equality operator. The refactoring can be triggered if the operands are both of type string. When these preconditions are met, the binary expression is replaced with a call to the Equals() method.

5.5.3 Extract Method

Yet another implemented refactoring is Extract Method, which is represented in Figure 5.9. It aims to remove a set of statements manually selected by the practitioner using the mouse and put them into a new method, which is then called exactly where those statements were in the original method. The refactoring was implemented following the algorithm presented by Fernandes et al. [30], which was inspired by Salgado's [69].



```
using RPCL;

new RefactoringPlugin("ExtractMethod", "Extract statements into a separate method")
    .TriggerOnAll("BlockSyntax")
    .IfSelectedStatementsAreOfType("InvocationExpressionSyntax")
    .IfNumberOfSelectedStatementsIsGreaterOrEqualThan(3)
    .IfRatioBetweenSelectedStatementsAndMethodStatementsIsLessThan(80)
    .IfStatementIsEnclosedIn("MethodDeclarationSyntax")
    .ExtractMethod()
    .Assemble();
```

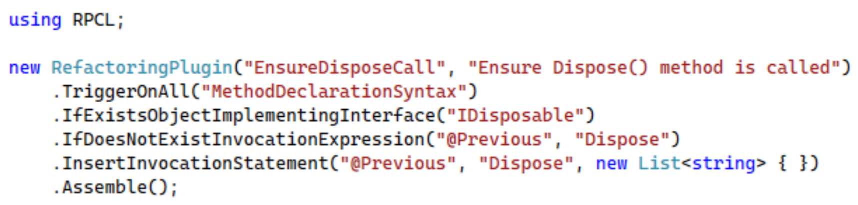
Figure 5.9: Declaration of the Extract Method refactoring using RPCL.

The refactoring is defined according to the following set of behaviors:

1. Candidate nodes must be valid statements according to the API in use.
2. Nodes must have associated parent nodes, meaning they must not be the root of a syntax tree.
3. Expressions cannot be trivial statements such as variable declarations.
4. The block of statements highlighted by the user cannot correspond to more than 80% of the original method's statements.
5. The block of statements highlighted by the user should contain at least three statements.
6. The block of statements must be made up of consecutive statements.

The first condition is valid as the refactorings target only Invocation Statements, a subset of valid Expression Statements. This fact also automatically validates conditions two and three. The sixth condition is also valid, as the refactoring is only triggered whenever a set of statements, also known as a syntax block, is highlighted with the mouse. The remaining conditions are assured by invoking the rest of the conditional modules.

5.5.4 Ensure Dispose Call



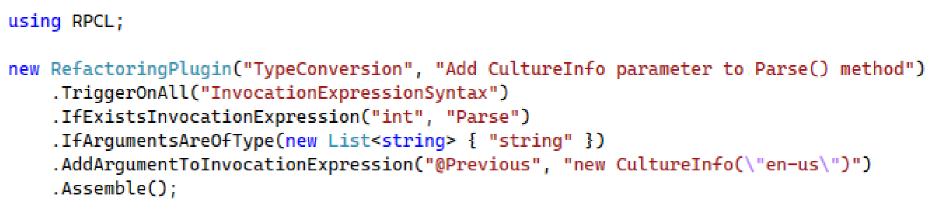
```
using RPCL;

new RefactoringPlugin("EnsureDisposeCall", "Ensure Dispose() method is called")
    .TriggerOnAll("MethodDeclarationSyntax")
    .IfExistsObjectImplementingInterface("IDisposable")
    .IfDoesNotExistInvocationExpression("@Previous", "Dispose")
    .InsertInvocationStatement("@Previous", "Dispose", new List<string> { })
    .Assemble();
```

Figure 5.10: Declaration of the Ensure Dispose Call refactoring using RPCL.

Figure 5.10 demonstrates an implementation of the Ensure Dispose Call refactoring through the orchestration of RPCL's behavioral modules. The refactoring is triggered only when a Method Declaration is in mouse-focus. Then, inside the method body, there must be an object declaration that implements the IDisposable interface. If so, the refactoring assesses whether that object invokes the Dispose method, achieved through the @Previous construct.

After stating all precondition behavioral modules, a transformation module is disclosed. Since Invocation Statements can vary significantly, the module allows users to customize several parameters, such as the object on which the Invocation Statement will be invoked, the name of the method to be invoked, and a list of the arguments to be passed. The argument list does not have a size limit, meaning that any arguments can be passed in any quantity, provided they are strings. In practice, the string type requirement is not a limitation as, in RPCL, no actual object is being passed, only its name. In the second abstraction level, the object's name will be introduced into the invocation, but this time without quotation marks. In essence, this means that the string literal passed to RPCL is converted into a concrete type when used by the following abstraction level.

5.5.5 Type Conversion



```
using RPCL;

new RefactoringPlugin("TypeConversion", "Add CultureInfo parameter to Parse() method")
    .TriggerOnAll("InvocationExpressionSyntax")
    .IfExistsInvocationExpression("int", "Parse")
    .IfArgumentsAreOfType(new List<string> { "string" })
    .AddArgumentToInvocationExpression("@Previous", "new CultureInfo(\"en-us\")")
    .Assemble();
```

Figure 5.11: Declaration of the Type Conversion refactoring using RPCL.

The Type Conversion refactoring aims to solve the problem of idiomatic peculiarities when parsing strings. The method Parse is characteristic of multiple types, such as int or float. This concrete refactoring aims to append a CultureInfo argument whenever the Parse method is called on a float object. This is useful as some information may vary from region to region. One example is the variety in decimal separators, which the dot or the comma characters may represent.

This refactoring is triggered only on Invocation Expressions. Additionally, the object where the invocation expression is being called must be named "float", as the Parse method is invoked on the float object. However, it only makes sense to invoke such a method if there is not already a CultureInfo object being passed to it. Therefore, the refactoring will only be invoked if there is only one argument in the invocation, which must be of type string. The method's first argument is of that type, corresponding to the string being parsed.

After all the preconditions are stated in RPCL, the transformation module adds the CultureInfo parameter. Note that the string has an expression for creating an object of that type and not to an existing object. The thought process is the same as with Ensure Dispose Call refactoring, allowing for passing objects of any kind. This creation string will then be processed internally and appear in the final refactoring plugin.

5.5.6 Creation Method

The Creation Method refactoring, like Chain Constructors, is part of the set presented by Kerievsky [45]. It intends to move the code inside the body of a constructor and put it into a new method, which is invoked in the original method. Due to this, the refactoring is only triggered when

```
using RPCL;

new RefactoringPlugin("CreationMethod", "Generate a creation method from a constructor")
    .TriggerOnAll("ConstructorDeclarationSyntax")
    .GenerateCreationMethodFromConstrctor("CreateGeneratedConstructor")
    .Assemble();
```

Figure 5.12: Declaration of the Creation Method refactoring using RPCL.

hovering constructor declarations, which is the only relevant precondition. Finally, the method with the extracted code is created, and an invocation to it is inserted in the method that originally contained it.

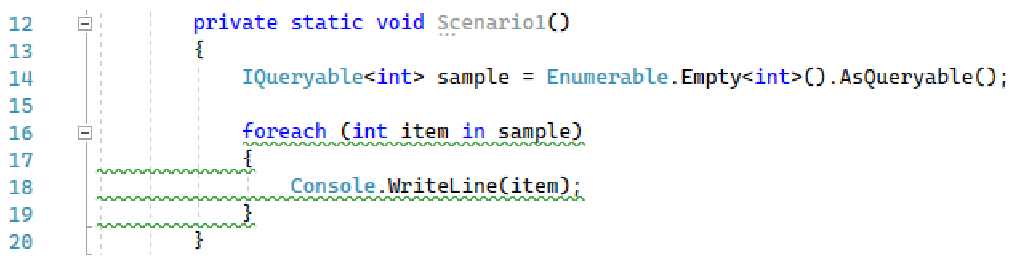
5.5.7 Append ToList to IQueryable

```
using RPCL;

new RefactoringPlugin("AppendToListToIQueryable", "Append the ToList() invocation to IQueryable object")
    .TriggerOnAll("ForEachStatementSyntax")
    .IfVariableBeingIteratedOverIsOfType("IQueryable")
    .AppendToListToExpression()
    .Assemble();
```

Figure 5.13: Declaration of the Append ToList to IQueryable refactoring using RPCL.

The Append ToList to IQueryable refactoring, built with the RPCL script present in Figure 5.13 converts the data from an object that implements the IQueryable interface into a List whenever that object is being iterated in a ForEach loop. Due to that, the refactoring will only trigger when hovering such loop statements. Then, it is assessed whether the variable being iterated over in the ForEach loop is of an IQueryable type. If so, the ToList() invocation call is appended to the variable being iterated.

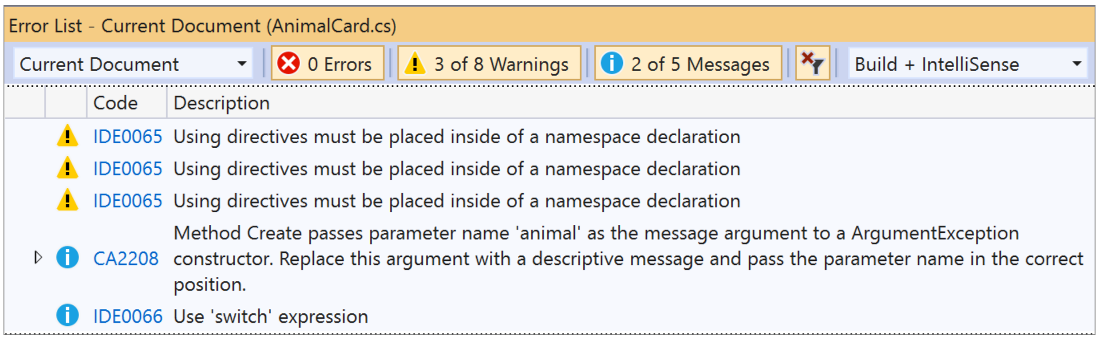
5.6 Experimenting with Code Analyzers



```
private static void Scenario1()
{
    IQueryable<int> sample = Enumerable.Empty<int>().AsQueryable();

    foreach (int item in sample)
    {
        Console.WriteLine(item);
    }
}
```

Figure 5.14: Visual aids of the IQueryable analyzer.

Refactorings are not the only instrument in Visual Studio's toolchain for directing developers' attention to code enhancements. Like refactorings, code analyzers [26] wander through code to find improvement opportunities. When found, they are displayed in a message pane in Visual Studio, as seen in Figure 5.15, with the difference that they parse all project files and not only the one that is opened at the moment. Moreover, the code editor highlights the relevant code segments to visually indicate where the suggestions were found in the code, exemplified in Figure 5.16. The practitioner also has the option to apply a transformation that applies the improvements suggested by the analyzers, which functions in the same manner as refactorings, as shown in Figure 5.17.


Error List - Current Document (AnimalCard.cs)

Current Document | 0 Errors | 3 of 8 Warnings | 2 of 5 Messages | Build + IntelliSense

Code	Description
IDE0065	Using directives must be placed inside of a namespace declaration
IDE0065	Using directives must be placed inside of a namespace declaration
IDE0065	Using directives must be placed inside of a namespace declaration
CA2208	Method Create passes parameter name 'animal' as the message argument to a ArgumentException constructor. Replace this argument with a descriptive message and pass the parameter name in the correct position.
IDE0066	Use 'switch' expression

Figure 5.15: Visual Studio's message pane. [26]

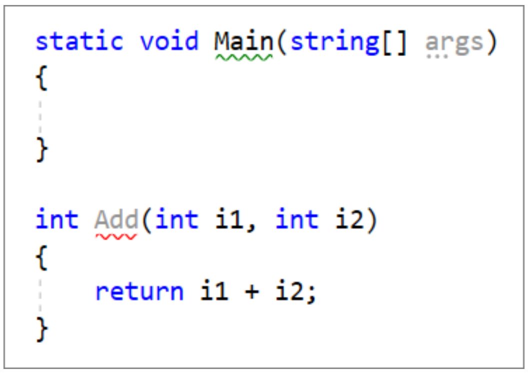
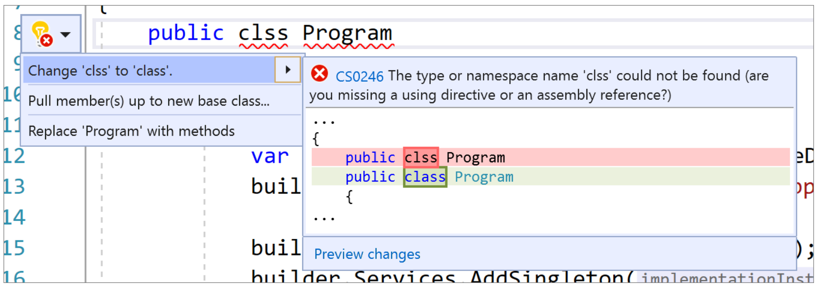

```
static void Main(string[] args)
{

}

int Add(int i1, int i2)
{
    return i1 + i2;
}
```

Figure 5.16: Visual squiggles suggesting a refactoring opportunity in the Visual Studio IDE. [26]


```
public clss Program
```
Change 'clss' to 'class'.
Pull member(s) up to new base class...
Replace 'Program' with methods
Preview changes

CS0246 The type or namespace name 'clss' could not be found (are you missing a using directive or an assembly reference?)

```
...
{
    public clss Program
    public class Program
    {
...
```

Figure 5.17: The application of a code transformation from an analyzer, identical to refactorings. [26]

For example, the Append ToList to IQueryable refactoring was implemented as a code analyzer. The main difference compared to refactoring implementation is how the opportunity is signaled. The visual cues may make identifying such opportunities more straightforward than having to hover over syntactic expressions manually, as can be visualized in Figure 5.14. Additionally, the list of all identified opportunities in the message pane may help practitioners get a more straightforward overview of all possible improvement chances and aid in prioritizing them.

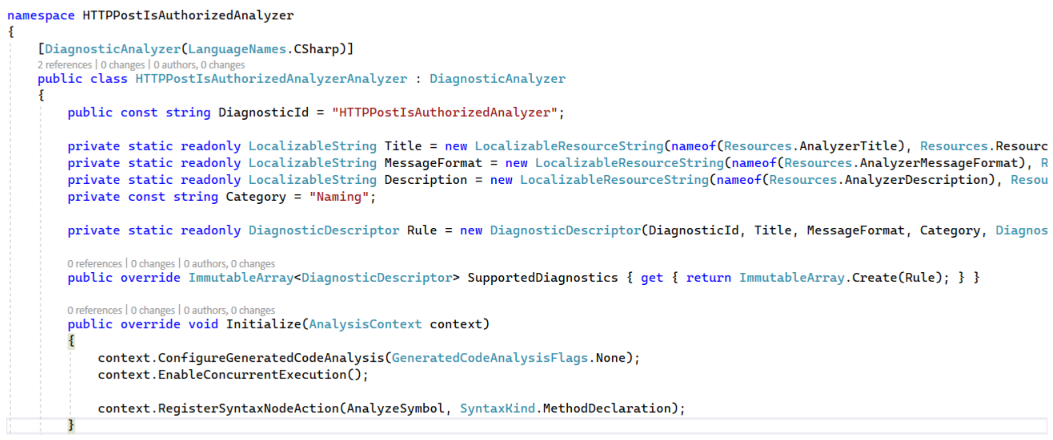
5.7 An Example of Using RPCL for Preventing Severe Security Vulnerabilities

During the development of RPCL, a situation emerged where DevScope, the company that supported the development of this dissertation, requested the development of a code analyzer to ensure that HttpPost routes were only accessible when adequately authorized. In more concrete terms, the company asked that, for every method with the HttpPost annotation, it was ensured that there was an invocation to the IsAuthorized() method. Not only would this analyzer warrant that no files possessed this vulnerability when deployed, but it would also help both experienced developers and newcomers to the company not forget always to perform this security check.

When the analyzer was run, it immediately and automatically found many files that missed this security check. As the code base over which the analyzer ran on top of was massive, it would have been tricky and tedious to verify if all routes complied with the security requirements manually.

While, in this case, a code analyzer was built to deal with this issue, a refactoring plugin could also have been built, similar to the tools described in Section 5.5. However, the dissertation was in a phase dedicated to experimenting with code analyzers. Nevertheless, separating behaviors into modules could be seamlessly used to build any refactoring plugin, including one for this concrete case. As for the code transformation that would occur when these conditions were verified, one possible implementation could be to append the invocation of the IsAuthorized() method to the body of all methods that comply with the two restrictions mentioned above.

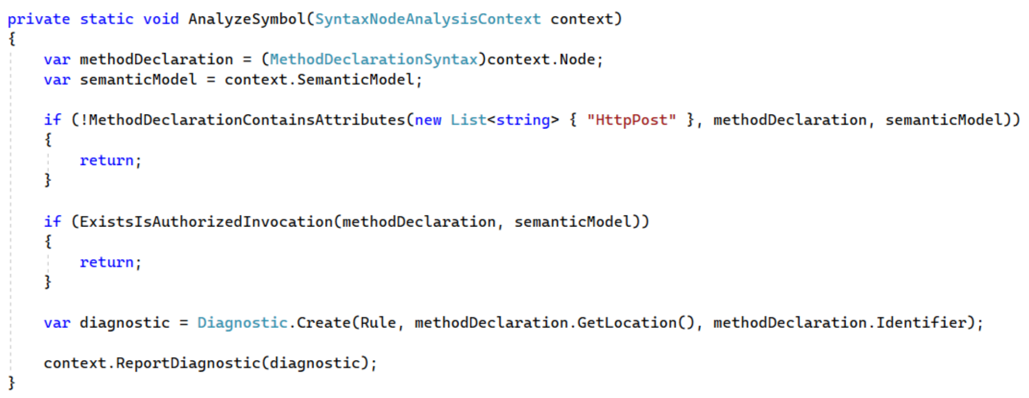
The implementation of the analyzer was very similar to that of a refactoring. Figures 5.18, 5.19 and 5.20 reveal some of the code that makes it up. Apart from the boilerplate code of the first two figures, there are few differences from refactoring code. This is good as it ensures that modules can be reused interchangeably between refactorings and code analyzers.



```
namespace HTTPPostIsAuthorizedAnalyzer
{
    [DiagnosticAnalyzer(LanguageNames.CSharp)]
    2 references | 0 changes | 0 authors, 0 changes
    public class HTTPPostIsAuthorizedAnalyzerAnalyzer : DiagnosticAnalyzer
    {
        public const string DiagnosticId = "HTTPPostIsAuthorizedAnalyzer";

        private static readonly LocalizableString Title = new LocalizableResourceString(nameof(Resources.AnalyzerTitle), Resources.Resourc
        private static readonly LocalizableString MessageFormat = new LocalizableResourceString(nameof(Resources.AnalyzerMessageFormat), R
        private static readonly LocalizableString Description = new LocalizableResourceString(nameof(Resources.AnalyzerDescription), Resou
        private const string Category = "Naming";

        private static readonly DiagnosticDescriptor Rule = new DiagnosticDescriptor(DiagnosticId, Title, MessageFormat, Category, Diagnos

        0 references | 0 changes | 0 authors, 0 changes
        public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get { return ImmutableArray.Create(Rule); } }

        0 references | 0 changes | 0 authors, 0 changes
        public override void Initialize(AnalysisContext context)
        {
            context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.None);
            context.EnableConcurrentExecution();

            context.RegisterSyntaxNodeAction(AnalyzeSymbol, SyntaxKind.MethodDeclaration);
        }
```

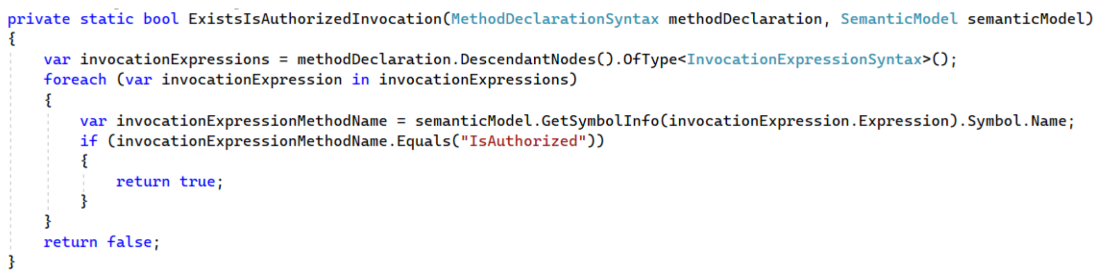
Figure 5.18: Template code that makes up a Visual Studio code analyzer.

Having routes that deal with sensitive content without security checks is a recipe for disaster, so production code must not have this security flaw. This is a real-life scenario in which a code analyzer saved the company from a potential disaster in minutes.



```
private static void AnalyzeSymbol(SyntaxNodeAnalysisContext context)
{
    var methodDeclaration = (MethodDeclarationSyntax)context.Node;
    var semanticModel = context.SemanticModel;

    if (!MethodDeclarationContainsAttributes(new List<string> { "HttpPost" }, methodDeclaration, semanticModel))
    {
        return;
    }

    if (ExistsIsAuthorizedInvocation(methodDeclaration, semanticModel))
    {
        return;
    }

    var diagnostic = Diagnostic.Create(Rule, methodDeclaration.GetLocation(), methodDeclaration.Identifier);

    context.ReportDiagnostic(diagnostic);
}
```

Figure 5.19: The logic for a code analyzer alerting the user.



```
private static bool ExistsIsAuthorizedInvocation(MethodDeclarationSyntax methodDeclaration, SemanticModel semanticModel)
{
    var invocationExpressions = methodDeclaration.DescendantNodes().OfType<InvocationExpressionSyntax>();
    foreach (var invocationExpression in invocationExpressions)
    {
        var invocationExpressionMethodName = semanticModel.GetSymbolInfo(invocationExpression.Expression).Symbol.Name;
        if (invocationExpressionMethodName.Equals("IsAuthorized"))
        {
            return true;
        }
    }
    return false;
}
```

Figure 5.20: The code for finding all IsAuthorized() invocation expressions.

Chapter 6

Validation

The developed language aimed to improve the creation experience of building refactoring tools, thus making the process simpler, more reusable, and modular. In order to validate the previously stated research hypothesis, an experiment was performed with multiple software developers of varying experience and academic backgrounds. This experiment is described in this section, encompassing the procedures as well as the collected results.

6.1 Objectives

This experiment aimed to assess whether using a language to orchestrate refactoring behaviors effectively decreased the difficulty, length, and accessibility of the refactoring plugin creation process.

One important aspect to be validated was how easy it was for practitioners to understand, regardless of their experience, what each available module did. This is critical as one of the project's main goals is to allow users to abstract from the implementation details and focus only on orchestrating the desired behaviors. The experiment aimed to understand how clear the modules were and how well practitioners could link each module to the behavior they had to add to the refactoring plugin.

Additionally, the refactorings should not take too long to implement, given that one of the leading project goals was to decrease the pre-required knowledge base to implement such plugins, as well as the required amount of code for the refactoring plugin creation. This should lead to decreased development times.

6.2 Guidelines

In order to validate the language, a set of guidelines were defined to provide answers to the research questions stated in Chapter 4.

Device. In order to experiment, a computer with the Windows 10 operating system and the Visual Studio 2022 IDE was required. Such a laptop was used for these experiments.

Local. The experiments were performed remotely. The meetings were set up via a Microsoft Teams call, through which testers could use the Request Control feature of this program to remotely control the IDE of the laptop mentioned above, property of the experiment's organizer.

Participants. All testers participated freely and voluntarily, and demonstrated interest in experimenting with the developed language. Each experiment was performed with one participant at a time. They had varying experience and academic degrees, but all were active professional software developers. None had previously experimented with the language.

Tool exposure. Testers were given a walk-through guide describing the project and experiment's flow. It also contained a similar description to the one verbally provided to users, a list of all the available language modules, a description of how to create a refactoring, and the set of refactorings to implement plugins for. Participants also had the chance to watch a step-by-step demo of the implementation of a refactoring plugin that was not part of the experiment list. The testers were free to ask any questions they considered pertinent. Then, they took on three different tasks to implement refactorings of increasing complexity.

Duration. The time allocated for completing all tasks was 40 minutes, counted after a 20-minute introduction and demonstration of the refactoring plugin creation process and totaling 60 minutes of interaction. The time that each developer spent completing each task was counted independently.

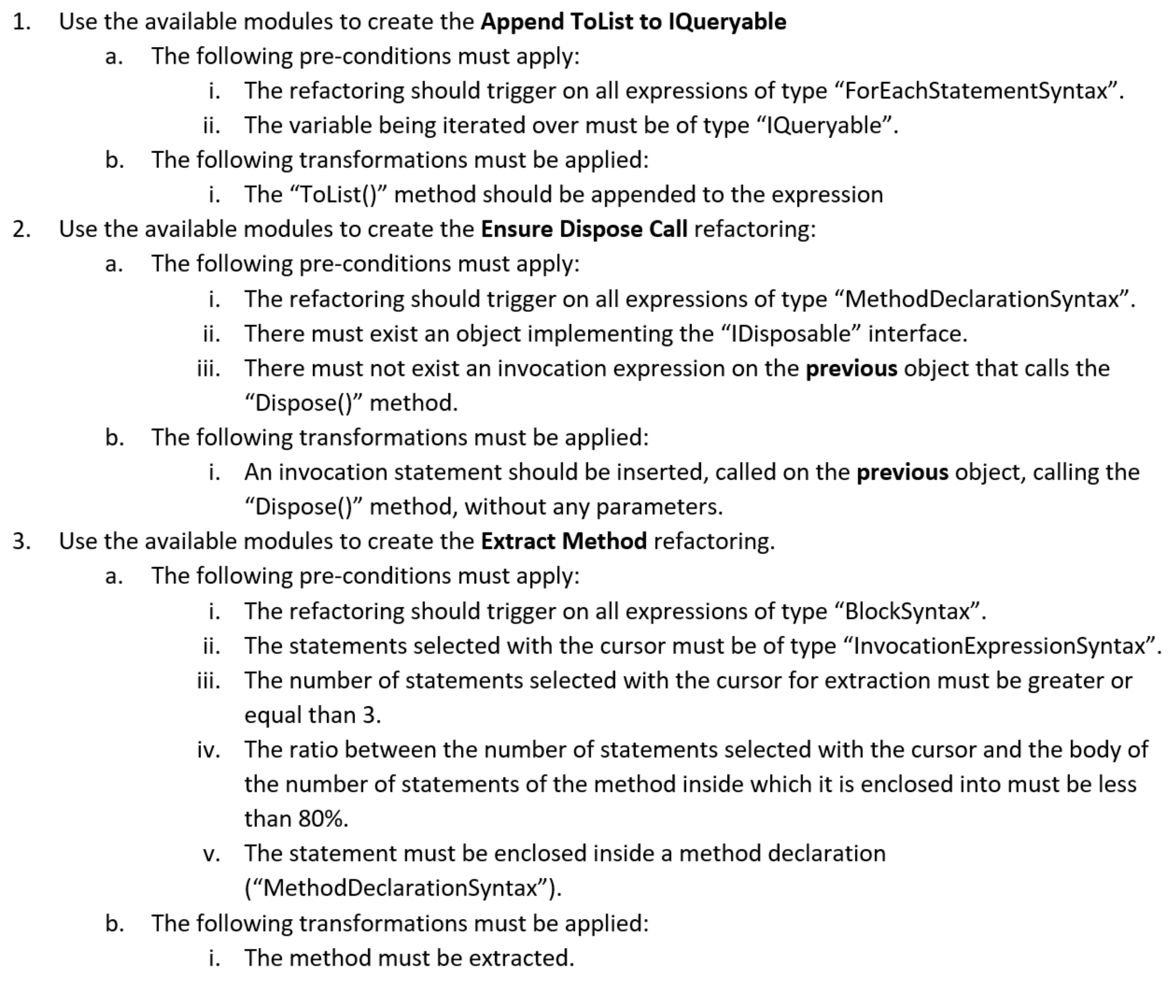
Questionnaire. A questionnaire was handed to all testers at the end of the experiment. It aimed to collect feedback on usability, ease of use, and relevance. Participants were also asked how many years they had exercised their profession and academic degree. All questions were open-ended.

6.3 Tasks

Participants were required to perform three different tasks. Each focused on orchestrating behavior modules for a particular refactoring, resulting in the generation of a refactoring plugin for each.

For a participant to know which modules to chain together in each task, the required behaviors were described in the provided walk-through guide, as shown in Figure 6.1. The complete guide is available in Appendix B, and the refactorings to be implemented in each task are described in Chapter 5.

The first task required the practitioner to implement the Append ToList to IQueryable refactoring plugin. The refactoring can be implemented by orchestrating the modules as per Figure 5.13. The second task is similar but focuses on the Ensure Dispose Call refactoring, while the third



1. Use the available modules to create the **Append ToList to IQueryable**
   a. The following pre-conditions must apply:
      i. The refactoring should trigger on all expressions of type “ForEachStatementSyntax”.
      ii. The variable being iterated over must be of type “IQueryable”.
   b. The following transformations must be applied:
      i. The “ToList()” method should be appended to the expression
2. Use the available modules to create the **Ensure Dispose Call** refactoring:
   a. The following pre-conditions must apply:
      i. The refactoring should trigger on all expressions of type “MethodDeclarationSyntax”.
      ii. There must exist an object implementing the “IDisposable” interface.
      iii. There must not exist an invocation expression on the **previous** object that calls the “Dispose()” method.
   b. The following transformations must be applied:
      i. An invocation statement should be inserted, called on the **previous** object, calling the “Dispose()” method, without any parameters.
3. Use the available modules to create the **Extract Method** refactoring.
   a. The following pre-conditions must apply:
      i. The refactoring should trigger on all expressions of type “BlockSyntax”.
      ii. The statements selected with the cursor must be of type “InvocationExpressionSyntax”.
      iii. The number of statements selected with the cursor for extraction must be greater or equal than 3.
      iv. The ratio between the number of statements selected with the cursor and the body of the number of statements of the method inside which it is enclosed into must be less than 80%.
      v. The statement must be enclosed inside a method declaration (“MethodDeclarationSyntax”).
   b. The following transformations must be applied:
      i. The method must be extracted.

Figure 6.1: The refactorings to be implemented by practitioners during the experiment, extracted from the walk-through guide.

task relates to the Extract Method refactoring. The RPCL code that practitioners should type in order to build each of these refactoring plugins is displayed in Figures 5.10 and 5.9, respectively. The order of the tasks reflects the complexity of each refactoring, from simpler to more complex. It is important to note that, in the case of these experiments, higher complexity does not mean the refactorings are harder to implement. Instead, it means that they require more modules to be chained together, further restraining the circumstances which allow a refactoring suggestion to be triggered.

6.4 Results

This section covers the participants’ thoughts concerning their experience during the experiment. The latter was performed with 10 different software developers.

6.4.1 Participant Characterization

In order to better validate the research hypothesis, the participants' academic degrees and development experience were collected.

Regarding work experience, 1 participant had been working professionally for less than 1 year, 7 participants had between 1 and 5 years of experience, and 2 participants had 10 or more years of experience. Concerning their academic qualifications, 8 possessed a Bachelor's degree, while 1 had a Master's degree, and 1 did not pursue any academic path. One of the Bachelors did not have formation in Computer Science, but Sociology.

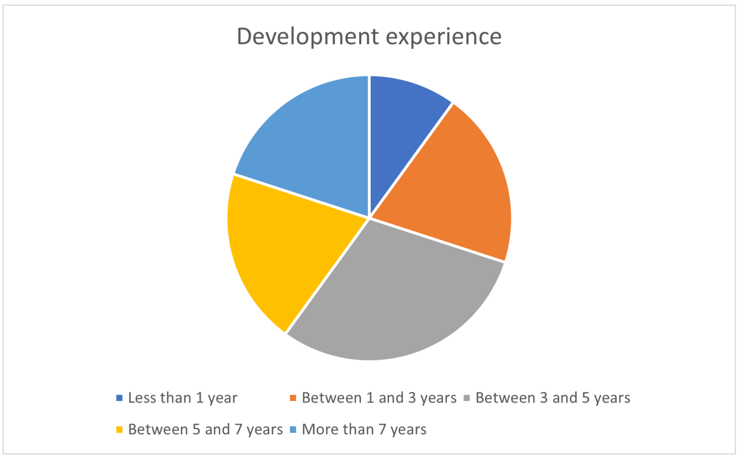


Figure 6.2: The varying levels of experience of participants, in years.

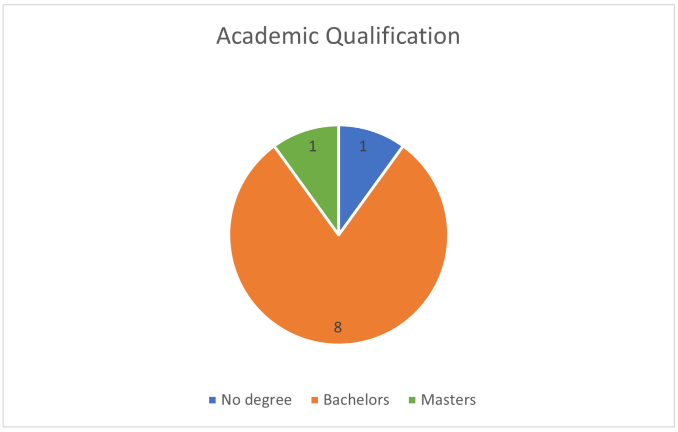


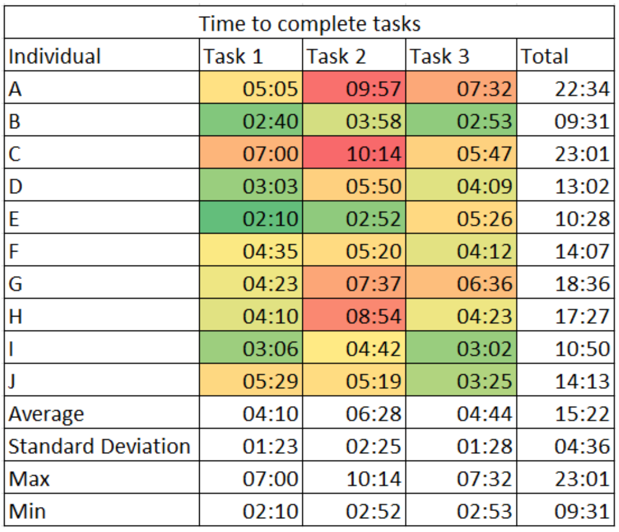
Figure 6.3: The academic degrees of participants.

6.4.2 Task Completion Times

As mentioned in this chapter, participants had 40 minutes to implement all 3 refactoring plugins. The time each participant took to implement each of them was measured and will be described next. It is also synthesized in Figure 6.4.

Regarding Task 1, participants spent, on average, 04:10 seconds implementing the refactoring plugin, with a standard deviation of 01:23 seconds. Task 2 was completed, on average, in 06:28 seconds, with a standard deviation of 02:25. Finally, Task 3 took, on average, 04:44 seconds to be finalized, with a standard deviation of 01:28.

Analyzing each participant's time to complete all three tasks is also helpful. On average, participants took 15:22 seconds to finish the experiment, with a standard deviation of 04:36 seconds. The fastest participant spent a total of 09:31 seconds on all tasks combined, while the slowest took 23:01 seconds.



Time to complete tasks				
Individual	Task 1	Task 2	Task 3	Total
A	05:05	09:57	07:32	22:34
B	02:40	03:58	02:53	09:31
C	07:00	10:14	05:47	23:01
D	03:03	05:50	04:09	13:02
E	02:10	02:52	05:26	10:28
F	04:35	05:20	04:12	14:07
G	04:23	07:37	06:36	18:36
H	04:10	08:54	04:23	17:27
I	03:06	04:42	03:02	10:50
J	05:29	05:19	03:25	14:13
Average	04:10	06:28	04:44	15:22
Standard Deviation	01:23	02:25	01:28	04:36
Max	07:00	10:14	07:32	23:01
Min	02:10	02:52	02:53	09:31

Figure 6.4: The time it took to each experiment participant to implement all three refactoring plugins, as well as the average and standard deviation values.

6.4.3 Survey Answers

At the end of the experiment, participants filled out a form with 4 open-ended questions regarding their experience with the developed language. As mentioned, these questions aimed to provide answers to the research questions that were proposed in Chapter 4 and focus on assessing the language's clarity, accessibility, and utility of the built-in Live Refactoring features, among other aspects.

The survey questions will be listed below, but the entire document containing the participant's answers is available in Appendix A.

SQ1: How easy was it to build refactoring tools? Was RPCL flexible and clear enough?

SQ2: Did you find the built-in Live Refactoring capabilities (live previews, automatic refactoring opportunities detection, one-click refactoring application) useful?

SQ3: Do you consider that RPCL incentivizes practitioners to build refactoring tools of their own, making the process easier? Why/Why not?

SQ4: In your opinion, what are the main benefits of using RPCL? And what are the main drawbacks?

These questions intend to answer the research questions declared in Chapter 4. Therefore, survey question SQ1 provides answers to RQ1, survey question SQ2 provides answers to RQ2, and SQ3 provides answers to RQ3. The last survey question aims to understand the language's characteristics that most made a difference to participants and the most significant improvement points.

6.4.3.1 SQ1 Answers

All participants agreed that the refactoring plugin creation process was straightforward. Concerning the language's simplicity and clarity, all practitioners agreed that the module orchestration process was "very intuitive", easy, and "self explanatory", with one adding that this simplicity became apparent once "you know how it works". Another participant mentioned that the language helped understanding how refactorings work, as "Personally I had never worked with refactoring [sic] and this tool made my understanding a lot easier". Additionally, it was referred that the language made it possible "to create something in minutes that otherwise would've taken hours to build and test". The documentation provided in the walk-through guide also made it easier to get to know the tool, as it was found to be "clear". Regarding the existing modules, it was pointed out that they "are extremely useful" and flexible due to "the amount of 'modules' provided out of the box".

One participant mentioned some clarity problems, "namely module names and parameter names, which could be confused with standard programming constructs, instead of the equivalent Roslyn concepts". Another issue regarding flexibility was that the language "is almost too flexible: conditionals and transformational expressions can be chained at will without any compile time checks, even when not compatible". Finally, there was a concern about "how easy it will be to develop new code to handle new refactorings".

6.4.3.2 SQ2 Answers

The totality of participants agreed that the "built-in experience with Visual Studio seems as optimal as possible", making the process easier and "allowed to identify exactly the result obtained from the use of refactoring [sic]". One of the participants pointed out that the tool could be of great use for new developers on a team that may not "be aware of the best practices" and "more experienced developers when they are distracted".

6.4.3.3 SQ3 Answers

Once again, there is consensus among participants regarding the incentives the language gives them to develop their plugins. All agreed that, as the language allows practitioners "to build them in a few lines of code", it "will make more developers start refactoring their code" more often.

The tool "makes the process easier and more intuitive, because it abstracts all of the plumbing logic necessary to program these features", meaning that "the programmer doesn't need to have the knowledge to do a refactoring from scratch and thus be able to do his own refactorings". Another participant found that "not only does the tool make creating refactoring tools far easier than conventional methods, it also makes it much more readable". These are all incentives to program refactorings, given that, as another practitioner mentions, "writing refactoring code is a very tedious task and consumes much time". It was mentioned that the incentives might come from the language's ease of use, as the many available modules also make the language more accessible than manually creating refactoring plugins. This is because "developers wouldn't like to try to create one refactoring tool if RPCL doesn't provide full or near coverage [sic]".

One issue during the experiments was that the language might sometimes be too abstract and simple, "blocking core functionality". This same participant also mentions that it is "important to find a middle ground between exposing underlying functionality, and making a simple user interface for the user".

6.4.3.4 SQ4 Answers

Regarding the most positive aspects, practitioners mention that it makes the process easier and quicker, as "the main benefits are definitely how easy and efficient it is to create refactoring tools". The ease of use was reinforced by another participant, mentioning that "anyone with knowledge of C#, after doing two refactorings, becomes already a pro [sic]". This, according to another participant, is critical to incentivizing users to refactor more often. Another participant mentions that "this project is a great opportunity to more easily implement wide code rules and suggestions". At the same time, it is "very helpful to teach younger developers how to code with the best practices" and provides an "expected increase in code quality". Time savings and a smooth learning curve were also pointed about as positive points of the language. The Live Refactoring features and integration with Visual Studio were "relatively simple and intuitive", working "as expected".

Concerning the language's characteristics that could be improved, it was mentioned that the language "could be extended further for 'power users'". While "flexible enough for plenty of use cases", the language "could be improved by providing more Intellisense opportunities, such as by "using types and lambda selectors instead of strings, and restricting chaining modules that are not compatible with each other". Another aspect mentioned was that the language currently only supports Visual Studio as the target IDE, which may limit the potential user base. Ideally, the tool would also be available in other IDEs such as IntelliJ's Rider or Microsoft's Visual Studio Code. Regarding the available RPCL modules, one practitioner suggested the "separation of the trigger's modules in several depending on the trigger condition". This is so that no strings used to specify the type of a certain language construct would have to be manually inserted, as is the case with the parameters that some modules receive (mostly related to Roslyn syntax kinds). Given that "in the current way the programmer would have to read a possible documentation to identify the string that corresponds to the desired situation", the participant argues that if these "were modules already created, he could see what the possibilities are, in the IDE itself". Also related to this observation, a different participant mentioned that the "need to memorize some internal names" may hinder the experience. Another comment focuses on the module availability, as two participants suggested that more modules would be needed for the language to cope with a lot more refactoring use cases. A final comment pointed out the required "maintenance or the expertise needed to keep building new refactorings", which may compromise the long-term use of the tool.

6.5 Threats to Validity

While the experiment was carefully curated in order to minimize errors and maximize accuracy, there are a few aspects that can negatively affect the validity of the results:

Sample size. Given that the experiment was performed with active software developers, it was not as easy to schedule meetings with as many practitioners as desired. Ideally, the experiment would be performed with a more significant number of software developers, contributing to more heterogeneity of personal characteristics, experience, and academic background.

Sample characterization. Due to the previous point, there was not as much diversity in the population as desired, as most participants had a Bachelor's degree compared to the single participant that had a Master's degree and the individual that did not have any degree at all. Moreover, most software developers who accepted participating in the experiment had between 2 and 5 years of experience. More variety would have been beneficial for the validity of the research questions.

Time pressure. All experiment tasks were timed to assess how much the tool made it quicker to build refactoring plugins for Visual Studio. All participants were informed of the matter before the beginning of the experiment. This may have affected their performance, as additional pressure may have been felt to complete all tasks in time.

Refactoring plugin selection. Because developers did not have much time available to experiment, it was not possible for them to try to implement all of the possible refactoring plugins. Participants did not test all modules even if the refactoring plugins were carefully selected to present varying difficulty levels. This is not ideal, as it does not allow to take as many definitive conclusions about the language's characteristics as intended.

6.6 Discussion

Overall, the experiment seemed to validate all of the research questions introduced in Chapter 6. Creating a refactoring plugin from scratch implies knowledge of compilers, design patterns, refactorings, language design, traditional software engineering skills, and both the target compiler and IDE platforms. Nevertheless, each participant's time to finish all required tasks was very short.

All participants agreed that the language was straightforward, flexible, and easy to use. They were able to implement all of the suggested refactorings. Again, a complete and favorable agreement was found regarding whether practitioners would be more incentivized to build refactoring plugins of their own and more often if they had access to RPCL.

One participant commented that the language lacked clarity in some cases. One example where this problem is revealed has to do with some modules that receive an "objectType" as a parameter. The participant thought that this was one of C#'s types, such as "string", "int", or "float". However, the name intended the user to specify the syntax node's type, such as "InvocationExpressionSyntax", "BinaryExpressionSyntax" or "ForEachStatementSyntax". The critique is valid, and the name of this parameter should have been "objectKind", as that is the name used by Roslyn internally to specify the types of syntax nodes available in the language.

Another participant said that not enough safety checks were being made to ensure the integrity of the refactoring language. Again, it is a good point. For instance, if a module that passes the @Previous construct as an argument is invoked, and if there is no previous declaration of a module that references a given variable, the construct will not have any variable to interact with. Currently, no checks are being made concerning this. However, all syntax checks offered by C# are being used.

A recurrent issue verified during the experiments was that most participants did not read or analyze all of the available modules in the walk-through guide. This led to them often picking modules somewhat related to the behavior to be implemented but was not the solution to the problem. This may indicate that had the modules been described in another order, the error rate would be lower, but this is not a reasonable solution. Decreasing the number of available modules may not be a feasible solution, too, as it would decrease design flexibility. An alternative would be to bet on complete and well-organized documentation that would better segment each type of module and the structures it interacts with.

Another interesting behavior identified during the experiments (which is also a consequence of the previous point) was that when participants identified a module that seemed general enough, they tended to put it in the script they were creating automatically. However, when they looked at the module list more thoroughly, they found other modules that were mode specific to their use case. This raises the question of whether it is better to build less but more flexible modules or more but less general ones. The described behavior seems to indicate that the first approach is the best.

Additionally, some participants did not understand the purpose of some modules. For instance, one picked the "IfArgumentsAreOfType" module instead of "IfObjectIsOfType". While the issue for this particular case may be that the developer did not know the difference between an argument and an object, it still means that the names were not clear enough and could be further improved to minimize these issues. This is tremendously important. Naming clarity is even more important to educate users as they use the language. One developer mentioned that RPCL improved its knowledge on refactorings, which demonstrates well the potential of the generated refactoring plugins in educating new and experienced developers throughout their daily tasks.

The development of new modules forces the user to know about the compiler and IDE APIs, and currently, there is no way to abstract the module creation process. The developed language is only as powerful as the modules it provides. It is expected that, as more modules are created, the more power users will have, and the more refactorings will be possible to be created.

The developers that took part in the experiment had varying experience levels. Some were recent software engineering graduates, while others had over a decade of experience. Despite that, all could implement the required refactorings without spending much time. It is worth reinforcing that successive tasks produced refactorings of increased complexity. The last task presented users with the challenge of implementing the Extract Method refactoring - a complex one to implement - and the focus of multiple research papers. The fact that participants were all able to implement it signals that the language is effective in allowing the creation of complex code transformations.

6.7 Summary

This chapter described both the validation process and its results. Firstly, the validation objectives were defined, and the experiment guidelines were explicit. This was followed by a description of the experiment tasks the participants were expected to take. Then, the results were laid out and analyzed, culminating with a list of factors that could potentially impact the validity of the experiments, as well as a brief discussion on some relevant issues relating to the validation results.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This dissertation exposed the essential background information on refactorings, design patterns, refactoring to patterns, meta-programming, code generation, and live software development. These techniques were identified as the base body of knowledge to build a system for creating complex refactorings. While refactorings and design patterns alone can improve software quality, applying successive refactorings towards a given design pattern can provide extended benefits to experts in these topics or inexperienced practitioners.

The literature review helped to better understand current methodologies, techniques, and approaches to refactoring plugin creation. Information on batch refactoring techniques and existing implementations allowed for better comprehension of how to develop such plugins. Novel techniques demonstrated the potential that refactoring to patterns could have on a software system.

Consequently, a language for building refactoring plugins was built, focusing on modularity, reusability, and accessibility. By orchestrating the set of desired refactoring behaviors, it became possible to automatically generate a refactoring plugin that, when plugged into Visual Studio, provided users with Live Refactoring suggestions that, according to them, enriched their experience. The language was also found simple to use, yet powerful nonetheless.

All of the refactoring plugins that participants were asked to implement during the experiments were able to do so in a total of about 15 minutes, which is a surprisingly low amount of time for implementing three different plugins.

Overall, it is possible to conclude that RPCL improved the user experience of creating refactoring plugins and incentivized developers to do so while benefiting from the built-in Live Refactoring features. Thus, the hypothesis is positively validated.

7.2 Main Contributions

With this dissertation, a comprehensive literature review was performed, presenting state of the art on creating complex refactoring tools. Additionally, the RPCL language was developed, providing multiple benefits to the refactoring plugin creation process. The following list describes the main contributions of this dissertation:

Literature Review. In Chapter 3, state of the art regarding refactoring creation tools was exposed. It provided indispensable insights on current and past trends for generating simple and complex refactoring tools. Due to its extension, the literature review significantly contributes to this dissertation.

Language for Generating Refactoring Plugins. RPCL can successfully create refactoring plugins. Both the language and the validation process make up a distinct contribution to both the academic and industrial world:

- **Abstracting the Refactoring Plugin Creation Process.** As mentioned, creating refactoring plugins is daunting for many, requiring users to know many different areas of computer science. RPCL abstracted that complexity and allowed participants that had never used the language to implement three different refactoring plugins in an average time of fifteen minutes. The language makes creating refactoring plugins an easy, fast, and accessible experience by automatically generating them from an easy-to-write and easy-to-read script.
- **Live Refactoring Features Integration.** Refactoring plugins generated via RPCL are equipped with automatic refactoring opportunity detection, preview, and application. This makes it seamless for the practitioner to use the generated refactoring plugin to identify and apply refactorings. It is also trivial for the developer using RPCL to create a refactoring plugin to integrate these Live Refactoring features without explicitly coding anything.
- **Survey.** By conducting experiments with participants and asking them to fill in a survey, it was possible to understand what they thought about the developed internal DSL, mainly concerning its usability and ease of use.

7.3 Main Difficulties

While working on this dissertation was a joyful experience, the ride was not always easy. However, the lower points were tremendously useful to understand better the current issues regarding the creation of refactoring plugins. Some of the hardest Rubicon rivers to cross were the following:

Understanding the Roslyn APIs. Even with solid knowledge of compilers, translating that into a concrete API such as Roslyn's demanded a study period of study. Due to the sheer size of the Roslyn project, there was often no documentation available, requiring experimenting a lot. This made it much harder to implement any refactoring plugin, but eventually, the problem was surpassed successfully. Being Roslyn such an enormous project, it was a challenge to understand it at global and more concrete scales, but it was yet another obstacle that did not remain in the way.

Deprecated Visual Studio APIs. Many of Visual Studio's APIs, such as those responsible for accessing and manipulating document and project information, were deprecated. Additionally, other extension points, such as building custom interfaces, used ancient technology and could not be used for this project.

Creating Modules from the Implemented Refactoring Plugins. Identifying common behaviors between refactoring plugins was difficult, as every plugin has its peculiarity, and every refactoring algorithm is different. Getting to a point where modules were generalizable enough was challenging. The same applies to parameterizing modules to become more flexible and adapt to more circumstances. This reinforces that module design should be performed with sufficient sobriety and clairvoyance.

7.4 Future Work

As naturally happens with all software projects, they evolve and should be maintained to keep working as intended. Additionally, multiple areas could be improved to make the refactoring plugin creation experience more enjoyable.

Additional Module Development. The more modules the language has, the more freedom the creator will have when orchestrating its refactorings. While many modules were implemented during the implementation phase of this dissertation, many more can be constructed. Flexibility is improved even further if the extra developed modules are flexible enough using parametrization. However, the choice of modules to be developed should be made with criteria, as the parametrization mentioned above can render the creation of additional modules redundant.

Improved Module Flexibility via Parameterization. Some of the modules developed for this dissertation could benefit from increased flexibility. For instance, the IfExistsObjectImplementingInterface module could also receive the object name as a parameter. This would allow deeper filtering of all candidate objects that implement a particular interface. Increased parameterization can also decrease the likelihood that the modules that RPCL provides are not enough for practitioners using the language, which could decide on building modules on their own. This, of course, is no trivial task and could make them give up on the idea and stop implementing the refactoring plugin altogether.

Increased Support for @Previous Construct. Not all available modules support the @Previous construct, which is not ideal given how useful it can be. Expanding the number of modules that support it can add even more power to the language.

Expanded IDE Support. As verified in the experiments, not all practitioners use Visual Studio as their primary IDE. As RPCL aims to abstract the refactoring plugin creation process, the ideal scenario would be to be able to generate plugins for multiple IDEs while keeping the language's syntax intact. This could be done by specifying the target IDE at the beginning of the script, with RPCL injecting the appropriate API calls into the final refactoring plugin file for providing Live Refactoring features.

Expanded Language Support. Similar to the previous point, it would be helpful if RPCL could target more languages than C#. Again, in an ideal scenario, RPCL's syntax would be the same, only the plumbing would be different, generating code structures for other languages such as Java.

Integrated Code Analyzer and Refactoring Features. While refactoring APIs help suggest potential refactoring opportunities whenever the user's cursor is near a given code segment, the warning messages that code analyzers provide are extremely useful. Additionally, they can show suggestions for the file currently opened and for all files in a given project. This may lead to additional time savings and a grasp of the project's overall status regarding code quality.

Documentation. The experiments revealed that excellent documentation is imperative for this project. It was verified that many practitioners did not go over all the available modules, leading them to pick the wrong ones due to not taking sufficient time. Nevertheless, the fact is that if participants are rushing, then something is not as well as it could be regarding module understandability and readability. Therefore, it would make sense to further segment module documentation into more concrete use cases so that developers could open the doors they want to enter and avoid those that have nothing to do with their intents and purposes.

Improve Module Naming Clarity. Some modules were chosen by mistake due to practitioners not understanding what they did by simply reading their names. While this was not usual, it still happened and suggests that naming could be further improved. If the module names are clear enough, developers may find which modules they need by simply typing keywords characteristic of the required behavior. The practitioner who achieved the fastest completion time was the only one who used this capability extensively. Others that used this also benefited from decreased time expenditures.

Publishing RPCL as a NuGeT Package. NuGeT packages are Microsoft's format used for distributing Visual Studio extensions. By packaging RPCL as an extension, its distribution can be done in a much more streamlined way, potentially increasing the user base. Also, when new modules are developed, they can be delivered to users via extension updates automatically handled by Microsoft.

Building a Visual Interface for Module Orchestration. It could be even more user-friendly if the refactoring behavior modules could be orchestrated by simply dragging and dropping them into a canvas and ordering them as developers see fit. Such a feature would make it even more accessible for anyone to create custom refactoring plugins and learn how they operate.

Extending the Survey to a Broader Audience. While the experiment sample provided valuable information on the efficiency and accessibility of the language, its size would ideally have been bigger. To do this, the process could be more streamlined so that participants could partake in the experiment alone, without a guide to help them in the process. Doing the experiments asynchronously would also be a way to increase the participant pool, as scheduling calls with developers is not always easy due to tight deadlines and calendar conflicts.

The previous list's extension demonstrates the potential for improvement concerning the development of refactoring plugins. These are exciting times for research on refactoring techniques and tooling, and indeed the user experience of creating such plugins will keep improving and becoming more accessible.

References

[1] Code refactoring | IntelliJ IDEA. https://www.jetbrains.com/help/idea/refactoring-source-code.html.

[2] Crossing Refactoring's Rubicon. https://martinfowler.com/articles/refactoringRubicon.html.

[3] Duplicate Code. https://refactoring.guru/smells/duplicate-code.

[4] Gang of Four Design Patterns Diagram. https://i.pinimg.com/originals/9a/c8/ea/9ac8ea1e4f898c9fe1e90fccd1

[5] JavaCompiler (Java Platform SE 6). https://docs.oracle.com/javase/6/docs/api/javax/tools/JavaCompiler.html.

[6] Jekyll • Simple, blog-aware, static sites. https://jekyllrb.com/.

[7] MethodDeclarationSyntax Class (Microsoft.CodeAnalysis.CSharp.Syntax) | Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.csharp.syntax.methoddeclarations dotnet-4.2.0.

[8] Refactoring source code in Visual Studio Code. https://code.visualstudio.com/docs/editor/refactoring.

[9] Secure and resizable cloud compute – Amazon EC2 – Amazon Web Services. https://aws.amazon.com/ec2/.

[10] Stack Overflow Developer Survey 2021. https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021.

[11] TechCohere - Microsoft Visual Studio 2019 - Code refactoring. https://techcohere.com/post/microsoft-visual-studio-2019-code-refactoring.

[12] Meta-Object Facility. *Wikipedia*, January 2021.

[13] Aharon Abadi, Ran Ettinger, and Yishai A. Feldman. Re-approaching the refactoring Rubicon. In *Proceedings of the 2nd Workshop on Refactoring Tools - WRT '08*, pages 1–4, Nashville, Tennessee, 2008. ACM Press.

[14] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development: Tightening the feedback loops. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, Programming '19, pages 1–6, New York, NY, USA, April 2019. Association for Computing Machinery.

[15] Diogo Amaral, Gil Domingues, João Dias, Hugo Ferreira, Ademar Aguiar, and Rui Nóbrega. Live Software Development Environment for Java using Virtual Reality:. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 37–46, Heraklion, Crete, Greece, 2019. SCITEPRESS - Science and Technology Publications.

[16] Kent Beck, First Class, Software James, O. Coplien, and Ron Crocker. Industrial experience with design patterns. In *In Proceedings of the 18th International Conference on Software Engineering, ICSE '96*, pages 103–114. IEEE Computer Society, 1996.

[17] Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. A Quantitative Study on Characteristics and Effect of Batch Refactoring on Code Smells. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, September 2019.

[18] George Candea and Patrice Godefroid. Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances. In Bernhard Steffen and Gerhard Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, Lecture Notes in Computer Science, pages 505–531. Springer International Publishing, Cham, 2019.

[19] Mel Ó Cinnéide. A Methodology for the Automated Introduction of Design Patterns. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, Oxford, UK, 1999. IEEE.

[20] Marshall P. Cline. The pros and cons of adopting and applying design patterns in the real world. *Communications of the ACM*, 39(10):47–49, October 1996.

[21] James R Cordy, Thomas R Dean, Andrew J Malton, and Kevin A Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13):827–837, October 2002.

[22] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[23] Robertas Damaševi. Taxonomy of the Fundamental Concepts of Metaprogramming. *Information Technology and Control*, 37(2):9, 2008.

[24] Anna Derezińska. A Structure-Driven Process of Automated Refactoring to Design Patterns. In Jerzy Świątek, Leszek Borzemski, and Zofia Wilimowska, editors, *Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems Architecture and Technology – ISAT 2017*, volume 656, pages 39–48. Springer International Publishing, Cham, 2018.

[25] Carlos Duarte. Live Requirements Engineering. https://caduonrails.com/blog/live-sd/, December 2021.

[26] Mika Dumont. Code analysis using Roslyn analyzers - Visual Studio (Windows). https://docs.microsoft.com/en-us/visualstudio/code-quality/roslyn-analyzers-overview.

[27] Mohamed El-Sharqwi, Hani Mahdi, and Islam El-Madah. Pattern-based model refactoring. In *The 2010 International Conference on Computer Engineering & Systems*, pages 301–306, Cairo, Egypt, November 2010. IEEE.

[28] Eduardo Fernandes, Anderson Uchoa, Ana Carla Bibiano, and Alessandro Garcia. On the Alternatives for Composing Batch Refactoring. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 9–12, Montreal, QC, Canada, May 2019. IEEE.

[29] Sara Fernandes. A live environment for inspection and refactoring of software systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1655–1659, Athens Greece, August 2021. ACM.

[30] Sara Fernandes, Ademar Aguiar, and André Restivo. A Live Environment to Improve the Refactoring Experience. In *Companion of the 6rd International Conference on Art, Science, and Engineering of Programming (Programming '22)*, page 8. ACM, 2022.

[31] Stephen R. Foster, William G. Griswold, and Sorin Lerner. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 222–232, Zurich, June 2012. IEEE.

[32] Martin Fowler. *Domain-Specific Languages*. Pearson Education, September 2010.

[33] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, November 2018.

[34] R. France, S. Chosh, E. Song, and D.K. Kim. A metamodeling approach to pattern-based model refactoring. *IEEE Software*, 20(5):52–58, September 2003.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, October 1994.

[36] Xi Ge and Emerson Murphy-Hill. BeneFactor: A flexible refactoring tool for eclipse. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion - SPLASH '11*, page 19, Portland, Oregon, USA, 2011. ACM Press.

[37] Kevin Hazzard and Jason Bock. *Metaprogramming in .NET*. Manning Publications, Shelter Island, NY, 2013.

[38] Sascha Hunold, Björn Krellner, Thomas Rauber, Thomas Reichel, and Gudula Rünger. Pattern-Based Refactoring of Legacy Software Systems. In Will van der Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, Clemens Szyperski, Joaquim Filipe, and José Cordeiro, editors, *Enterprise Information Systems*, volume 24, pages 78–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[39] Standish Group Int. The Chaos Report, 2015.

[40] Shalinka Jayatilleke and Richard Lai. A systematic review of requirements change management. *Information and Software Technology*, 93:163–185, January 2018.

[41] Adam C. Jensen and Betty H.C. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation - GECCO '10*, page 1341, Portland, Oregon, USA, 2010. ACM Press.

[42] Sang-Uk Jeon, Joon-Sang Lee, and Doo-Hwan Bae. An automated refactoring approach to design pattern-based program transformations in Java programs. In *Ninth Asia-Pacific Software Engineering Conference, 2002.*, pages 337–345, December 2002.

[43] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[44] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 736–743, Florence, Italy, 2001. IEEE Comput. Soc.

[45] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Education, August 2004.

[46] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491, pages 222–289. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[47] Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51, August 2004.

[48] Huiqing Li and Simon Thompson. A Domain-Specific Language for Scripting Refactorings in Erlang. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Juan de Lara, and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212, pages 501–515. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[49] Stephen W. Liddle. Model-Driven Software Development. In David W. Embley and Bernhard Thalheim, editors, *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*, pages 17–54. Springer, Berlin, Heidelberg, 2011.

[50] Wei Liu, Zhi-gang Hu, Hong-tao Liu, and Liu Yang. Automated pattern-directed refactoring for complex conditional statements. *Journal of Central South University*, 21(5):1935–1945, May 2014.

[51] Pedro Lourenço, João Dias, Ademar Aguiar, and Hugo Ferreira. CloudCity: A Live Environment for the Management of Cloud Infrastructures. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 27–36, Heraklion, Crete, Greece, 2019. SCITEPRESS - Science and Technology Publications.

[52] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, August 2008.

[53] Panita Meananeatra. Identifying refactoring sequences for improving software maintainability. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 406–409, September 2012.

[54] Tom Mens, Gabriele Taentzer, and Dirk Müller. Challenges in Model Refactoring. In *Proc. 1st Workshop on Refactoring Tools, University of Berlin*, volume 98, July 2007.

[55] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 331–336, Vasteras Sweden, September 2014. ACM.

[56] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. In Bertrand Meyer, Jerzy R. Nawrocki, and Bartosz Walter, editors, *Balancing Agility and Formalism in Software Engineering*, volume 5082, pages 252–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[57] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. Does refactoring improve reusability. In *Ninth International Conference on Software Reuse (ICSR-9*, pages 11–15, 2006.

[58] Umesha Naik and D Shivalingaiah. Comparative Study of Web 1.0, Web 2.0 and Web 3.0. In *6th International CALIBER -2008*, page 9, 2008.

[59] Daniel Oliveira, Ana Carla Bibiano, and Alessandro Garcia. On the Customization of Batch Refactoring. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 13–16, May 2019.

[60] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Mel Ó Cinnéide, Kalyanmoy Deb, and Katsuro Inoue. A Multi-Objective Refactoring Approach to Introduce Design Patterns and Fix Anti-Patterns. In *North American Search Based Software Engineering Symposium 00*, page 16. Elsevier, 2015.

[61] Ing J F Overbeek. *Meta Object Facility (MOF)*. PhD thesis, University of Twente, 2006.

[62] Gustavo H. Pinto and Fernando Kamei. What programmers say about refactoring tools?: An empirical investigation of stack overflow. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools - WRT '13*, pages 33–36, Indianapolis, Indiana, USA, 2013. ACM Press.

[63] L. Prechelt, B. Unger, W.F. Tichy, P. Brossler, and L.G. Votta. A controlled experiment in maintenance: Comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 27(12):1134–1144, December 2001.

[64] J. Rajesh and D. Janakiram. JIAD: A tool to infer design patterns in refactoring. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming - PPDP '04*, pages 227–237, Verona, Italy, 2004. ACM Press.

[65] Jacek Ratzinger, Michael Fischer, and Harald Gall. Improving Evolvability through Refactoring. In *MSR'05*, page 5, 2005.

[66] Jacek Ratzinger, Thomas Sigmund, and Harald C Gall. On the Relation of Refactoring and Software Defects. In *MSR'08*, page 4, May 2008.

[67] Don Roberts and Ralph Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, 1986.

[68] Beppe Salerno. Crossing the Rubicon. https://www.tourissimo.travel/blog/crossing-the-rubicon.

[69] Sérgio António Dias Salgado. *Towards a Live Refactoring Recommender Based on Code Smells and Quality Metrics*. Masters in Informatics and Computing Engineering, Faculty of Engineering of the University of Porto, 2022.

[70] Max Schaefer and Oege de Moor. Specifying and implementing refactorings. *ACM SIGPLAN Notices*, 45(10):286–301, October 2010.

[71] Takao Shimomura, Kenji Ikeda, and Muneo Takahashi. An Approach to GA-Driven Automatic Refactoring Based on Design Patterns. In *2010 Fifth International Conference on Software Engineering Advances*, pages 213–218, August 2010.

[72] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858–870, Seattle WA USA, November 2016. ACM.

[73] K R Srinath. Python – The Fastest Growing Programming Language. *International Research Journal of Engineering and Technology (IRJET)*, 04(12):5, 2017.

[74] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley, Chichester, England ; Hoboken, NJ, 2006.

[75] Steven L. Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34, May 2013.

[76] L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. In *14th IEEE International Conference on Automated Software Engineering*, pages 174–181, Cocoa Beach, FL, USA, 1999. IEEE Comput. Soc.

[77] T. Tourwe and T. Mens. Automated support for framework-based software. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 148–157, September 2003.

[78] T. Tourwe and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 91–100, March 2003.

[79] Mohsen Vakilian and Ralph E Johnson. Composite Refactorings: The Next Refactoring Rubicons. page 1, June 2012.

[80] Marek Vokáč, Walter Tichy, Dag I. K. SjØberg, Erik Arisholm, and Magne Aldrin. A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns—A Replication in a Real Programming Environment. *Empirical Software Engineering*, 9(3):149–195, September 2004.

[81] Bill Wagner. Get started with syntax analysis (Roslyn APIs). https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/get-started/syntax-analysis.

[82] Bill Wagner. The .NET Compiler Platform SDK (Roslyn APIs). https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/.

[83] Bill Wagner. Reflection (C#). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection.

[84] Bill Wagner. Source Generators. https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview.

[85] Norihiro Yoshida and Katsuro Inoue. Towards an Investigation of Opportunities for Refactoring to Design Patterns. In *Proceedings 1st International Workshop on Software Patterns and Quality (SPAQu'07)*, page 3, Nagoya, Japan, December 2017. Information Processing Society of Japan.

[86] Carmen Zannier and Frank Maurer. Tool Support for Complex Refactoring to Design Patterns. In *Processes in Software Engineering, Proceedings of the 4 Th International Conference XP 2003, Genova, Italy (2003), LNCS 2675*, pages 123–130. Springer, 2003.

Appendix A

Survey Results

Survey Results

For how long have you been a software developer?

- 1 ano
- 5 years
- 3 years
- About 10 months
- 1 ano
- 10y
- 14
- 5 years
- 4 years
- 4

What is your degree? (Bachelors, Masters, Doctorate...)

- Licenciatura em Sociologia
- Masters
- Bachelors
- Bachelors in informatics engineering and I'm doing a master's degree in AI
- Licenciatura
- I've no degree.
- Bachelor
- Bachelors
- Bachelors
- Bachelors

How easy was it to build refactoring tools? Was RPCL flexible and clear enough?

- Bastante fácil, após leitura da documentação todo processo é bastante intuitivo
- It was a very easy and simple process. The method names were intuitive and self explanatory.
- RPCL was easy to use and suitably flexible given the amount of ""modules"" provided out of the box. There were some problems with clarity, namely module names and parameter names, which could be confused with standard programming constructs, instead of the equivalent Roslyn concepts. It is also difficult to assert which Roslyn ""kind"" types to use on the various conditional methods available. Knowing these kinds requires some Roslyn knowledge if not provided with a cheat sheet, documentation or examples. The most frequent ""kinds"" could be included in string constants and made available to the user. These constants could be documented with C# XML docs, providing small examples of usage when the programmer uses Intellisense/auto-complete in Visual Studio. Additionally, on an effort to improve flexibility, RPCL is almost too flexible: conditionals and transformational expressions can be chained at will without any compile time checks, even when not compatible. This experience could be

improved by using type parameters instead of type strings and expression lambdas for configuration.
- Personally I had never worked with refactoring and this tool made my understanding a lot easier. The need for only logical understanding and the very representative module names make the tool very easy and clear to use.
- Os métodos tinham nomes clarificativos e foi fácil perceber como funciona após ler a documentação.
- Was very easy, course one time that you know how it works. Was flexible inside what is done.
- It was easy. We just need to follow the guidelines and it's easy to build refactorings. IT seems to be flexible but I don't know how easy it will be to develop new code to handle new refactorings.
- It was very easy, the documentation was clear and the already existing methods on RPCL are extremely useful.
- It was easy in parts, We have very specific methods to help us, and de RPCL was be clear too.
- Super easy, given the documentation provided, I managed to create something in minutes that otherwise would've taken hours to build and test.

Did you find the built-in Live Refactoring capabilities (live previews, automatic refactoring opportunities detection, one-click refactoring application) useful?

- Sim, todas estas funcionalidades facilitam a implementação/testagem do refractoring
- Yes, it is very helpful.
- Yes! The built-in experience with Visual Studio seems as optimal as possible at this stage.
- Yes, the built-in Live Refactoring capabilities responded quite well according to the specified conditional modules and allowed to identify exactly the result obtained from the use of refactoring.
- Sim, estas funcionalidades são muito úteis para a utilização no dia a dia.
- Yes, I could see many useful cases for this.
- yes specially for the new developers on the team who couldn't be aware of the best practices and even for the more experienced developers when they are distracted.
- Yes, made the refactoring process a lot easier, after creating the refactoring tasks.
- Yes, it helps us to programming better.
- Yes, very useful. It enables the developer to take action immediately in a very useful and non-intrusive matter.

Do you consider that RPCL incentivizes practitioners to build refactoring tools of their own, making the process easier? Why/Why not?

- Sim, uma vez que torna fácil todo o processo de criação de Refractoring não sendo necessário ao developer gastar muito tempo com isso
- The level of abstraction that comes with this project is very helpful and encouraging. Implementing new code practices is much simpler using RPCL.
- RPCL as a tool definitely makes the process easier and more intuitive, because it abstracts all of the plumbing logic necessary to program these features. However, in an effort to abstract that

plumbing, RPCL can become either too abstract, becoming plumbing of its own, or too simple, blocking core functionality. Its important to find a middle ground between exposing underlying functionality, and making a simple interface for the user.
- Yes, the ease of use of this tool means that the programmer doesnt need to have the knowledge to do a refactoring from scratch and thus be able to do his own refactorings.
- O facto de os refactors serem sugestões do IDE facilitam o incentivo na utilização desta prática.
- In general yes, one time that there are many "methods" to cover many situations. Because day to day developers wouldn't like to try to create one refactoring tool if the RPCL doesn't provide full or near coverage.
- Yes if they want to invest time. For sure it could help a lot when the team is focused or have someone focused to create and maintain refactorings
- Yes. Writing refactoring code is a very tedious task and consumes a lot of time, having a tool to build them in a few lines of code will make more developers start refactoring their code.
- I think that a lot of developers have a pattern, so they can make a refactoring method to help them.
- I did, not only does the tool make creating refactoring tools far easier than conventional methods, it also makes it much more readable.

In your opinion, what are the main benefits of using RPCL? And what are the main drawbacks?

- Torna o processo de criação e aplicação de refractors fácil e rápido, e é precisamente o oposto destes pontos que por vezes afasta os developers de alteram código que á muito já deveria ter sido alterado
- This project is a great opportunity to more easily implement company wide code rules and suggestions. It will be very helpful to teach younger developers how to code with the best practices.
- RPCL provides a simple interface for complex Roslyn and Visual Studio features in a neat, simpler package. Integration with Visual Studio seems relatively simple and intuitive, while live features within the IDE work as expected. At the current stage it is flexible enough for plenty of use cases, but could be extended further for ""power users"". Usability while orchestrating refactors could be improved by providing more Intellisense opportunities, by using types and lambda selectors instead of strings, and restricting chaining modules that are not compatible with each other. RPCL could also be extended to support additional IDEs as ""targets"", like Rider or Visual Studio Code.
- In my opinion, the tool is quite easy to use. Anyone with knowledge of csharp, after doing two refactorings, becomes already a pro and ready to do anything else. As an improvement to point out, and I don't know if it's an improvement or personal preference, it would be the separation of the trigger's module in several depending on the trigger condition. Because in the current way the programmer would have to read a possible documentation to identify the string that corresponds to the desired situation, while if they were modules already created, he could see what the possibilities are, in the IDE itself.
- Facilita muito o uso das técnicas de melhoria do código escrito. A grande desvantagem é as limitações existentes dos refactors possíveis, mas é normal visto que é o início do projeto.

- The benefits are time savings and learning curve. The disadvantages are that you need to memorize some internal names and the coverage of situations is not complete.
- The drawbacks could be related to the maintenance or the expertise needed to keep building new refactorings. The benefits are the expected increase in code quality.
- The main benefit is the ease with which a programmer can start refactoring their code. I did not find any drawbacks in the little time I used RPCL. It delivers what it promises.
- Productivity and clean code
- The main benefits are definitely how easy and efficient it is to create refactoring tools. I didn't find any major drawbacks, that being said, a cleaner interface would make the process even easier.

Appendix B

Expert Assessment of RPCL

Expert Assessment of RPCL

Introduction

Refactoring is the process of improving the design of existing code while maintaining its original functionality.

There exist multiple refactoring plugins that assist the automation of refactorings. For instance, modern IDEs suggest refactorings to users whenever the conditions for their application are verified. However, it may happen that the IDE does not provide practitioners with the refactorings they need. This forces them to build refactorings of their own.

Creating such refactoring plugins is also no trivial task, requiring the creator to have good-enough understanding of not-so-trivial topics such as compilers, programming languages, refactorings, and design patterns. Moreover, the developer must be comfortable with the APIs of the development environment the refactorings are targeting, given that each language has its own API for manipulating source code in a metaprogramming fashion.

RPCL is an internal domain-specific language based in C# (which reads just like English, albeit with the C# syntactic rules) targeting for generating refactoring implementations compatible with the Visual Studio IDE. The language does not refactor code directly but generates a C# file to be enveloped in a Visual Studio plugin that offers refactoring suggestions, previews, and transformations. It was created to make the refactoring creation process as seamless and straightforward as possible, reducing the amount of pre-required knowledge and increasing the pool of developers able to create refactorings. These are created by chaining pre-conditions and behaviors, much like in object-oriented languages you chain methods over a certain object. These behaviors, also referred to as modules, are reusable blocks of code that interact with the compiler and with Visual Studio APIs, performing most of the heavy lifting of the refactoring creation process. You, as the user, must only worry about chaining together the behaviors the refactoring should have. This way, you can focus on the logic behind the refactoring, and not on programming complex algorithms and understanding obscure and prone-to-obsoletion APIs.

Interview flow

The interview will begin with a briefing of what RPCL is and how it can be used to create refactorings. Then, you will be asked to implement a set of refactorings using RPCL, and to try and use them in the development environment. This will be a guided process, meaning that, after you create the refactorings, the generated file will be attached to Visual Studio by the experiment moderator on a project built for experimenting with refactoring creation, so that you can try and use it in a real-life scenario. Finally, a set of questions will be asked to better understand and evaluate your experience creating refactorings using RPCL. These will be open questions, meaning you will be able to give whatever feedback you find useful, be it positive or negative.

You'll be interacting with the IDE via a Microsoft Teams call, which will allow you to take control of the IDE as if it were running in your own computer.

Available RPCL modules

RPCL modules can be of 3 types: base, conditional, and transformational.

- Base
    - `TriggerOnAll(string expressionKind)`
- Conditional
    - `IfExistsObjectImplementingInterface(string interfaceName)`

    - `IfDoesNotExistInvocationExpression(string objectName, string methodName)`
    - `IfExistsInvocationExpression(string objectName, string methodName)`
    - `IfBinaryOperandsAreOfType(string firstOperandType, string secondOperandType)`
    - `IfObjectIsOfType(string objectName, string objectType)`
    - `IfObjectIsNamed(string objectName)`
    - `IfArgumentsAreOfType(List<string> argumentTypes)`
    - `IfBinaryExpressionOperandsAreOfType(string firstOperandType, string secondOperandType)`
    - `IfExistsCatchAllConstructor()`
      `IfSelectedStatementsAreOfType(string statementType)`
    - `IfNumberOfSelectedStatementsIsGreaterOrEqualThan(int numberOfSelectedStatements)`
    - `IfRatioBetweenSelectedStatementsAndMethodStatementsIsGreaterThan(int percentage)`
    - `IfRatioBetweenSelectedStatementsAndMethodStatementsIsLessThan(int percentage)`
    - `IfStatementIsEnclosedIn(string structureType)`
    - `IfParentNodeIsNotOfType(string parentNodeType)`
    - `IfVariableBeingIteratedOverIsOfType(string variableType)`
- Transformational
    - `InsertInvocationStatement(string objectName, string methodName, List<string> arguments)`
    - `AddArgumentToInvocationExpression(string objectName, string argument)`
    - `ReplaceBinaryExpressionWithEqualsMethod()`
    - `GenerateCreationMethodFromConstrctor(string creationMethodName)`
    - `ChainConstructors()`
    - `ExtractMethod()`
    - `AppendToListToExpression()`

Anatomy of a RPCL refactoring

The syntax for creating a new refactoring using RPCL is identical to the syntax of creating an object using C#, and follows the following template:

```
using RPCL; // MANDATORY. Imports the RPCL library

new RefactoringPlugin("RefactoringPluginNameWithoutSpaces", "A brief description of what the refactoring does")
    .TriggerOnAll("NameOfTheSyntacticKind") // MANDATORY MODULE. The argument should be the name of the targeted syntactic kind
    // Sequential chaining of the available conditional RPCL modules
    // Sequential chaining of the available transformational RPCL modules
    .Assemble(); // MANDATORY MODULE. Writes the generated plugin into a file
```

Walkthrough guide

A class that represents a bank loan (Loan) serves as the basis for this experiment. The class sports many variables and methods.

Over the course of this experiment, you will be creating a set of refactorings using RPCL. The following list contains the refactorings to be implemented, together with a brief description of what they are supposed to do, the pre-conditions and the transformations to apply. You must only chain behaviors/modules of the RPCL language on the RefactoringPlugin object.

After creating a refactoring script, you must run the RefactoringPlugin. It will then output the refactoring code that interacts both with the C# compiler as well as the Visual Studio APIs for previewing code changes, applying transformations and automatic refactoring detection.

NOTE: The refactoring name must not contain any spaces!

1. Use the available modules to create the **Append ToList to IQueryable:**
   a. The following pre-conditions must apply:
      i. The refactoring should trigger on all expressions of type “ForEachStatementSyntax”
      ii. The variable being iterated over must be of type “IQueryable”
   b. The following transformations must be applied:
      i. The “ToList()” method should be appended to the expression
2. Use the available modules to create the **Ensure Dispose Call** refactoring:
   a. The following pre-conditions must apply:
      i. The refactoring should trigger on all expressions of type “MethodDeclarationSyntax”
      ii. There must exist an object implementing the “IDisposable” interface
      iii. There must not exist an invocation expression on the previous object that calls the “Dispose()” method
   b. The following transformations must be applied:
      i. An invocation statement should be inserted, called on the previous object, calling the “Dispose()” method, without any parameters
3. Use the available modules to create the **Extract Method** refactoring:
   a. The following pre-conditions must apply:
      i. The refactoring should trigger on all expressions of type “BlockSyntax”
      ii. The statements selected with the cursor must be of type “InvocationExpressionSyntax”
      iii. The number of statements selected with the cursor for extraction must be greater or equal than 3
      iv. The ratio between the number of statements selected with the cursor and the body of the number of statements of the method inside which it is enclosed into must be less than 80%
      v. The statements must be enclosed inside a method declaration (“MethodDeclarationSyntax”)
   b. The following transformations must be applied:
      i. The method must be extracted