



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

ASSIGNMENT OF BACHELOR'S THESIS

Title: Code refactoring using Codiscent's projective technologies
Student: Tomáš Buňata
Supervisor: doc. Ing. Robert Pergl, Ph.D.
Study Programme: Informatics
Study Branch: Computer Science
Department: Department of Theoretical Computer Science
Validity: Until the end of summer semester 2020/21

Instructions

Refactoring became an everyday process of keeping the code clean, consistent and understandable. A range of tools for a code refactoring exists, however they are tightly bound with concrete language and IDE. The goal of this explorative thesis is to apply Projective Technologies of Codiscent for code refactoring.

1. Perform a review of current approaches and tools for refactoring and Reverse Engineering Studio (RES) and Generative Engineering Studio (GES) of Codiscent.
2. Create a suitable format to define rules for refactoring.
3. Design templates for RES and GES to perform the refactoring.
4. Show your solution on a representative subset of typical code refactorings as discussed in [1].
5. Formulate conclusions.

References

- [1] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., & Gamma, E. (1999). Refactoring: Improving the Design of Existing Code (1 edition). Reading, MA: Addison-Wesley Professional.
- [2] Červenka, J. (2015). Aplikace projektivních technologií pro objektově-orientovaný návrh webového uživatelského rozhraní. Bakalářská práce FIT ČVUT. URL: <https://dspace.cvut.cz/handle/10467/63020>

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 7, 2019



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Bachelor's thesis

Code refactoring using Codiscent's projective technologies

Tomáš Buňata

Department of Computer science
Supervisor: Ing. Robert Pergl, Ph.D.

January 9, 2020

Acknowledgements

I thank my thesis supervisor, Ing. Robert Pergl, Ph.D., for guidance, optimism and patience during my work. I also thank my family for their support during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on January 9, 2020

.....

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Tomáš Buňata. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Buňata, Tomáš. *Code refactoring using Codiscent's projective technologies*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Práce se zabývá využitím projektivních technologií k refaktoringu a analýzou současných přístupů a nástrojů k udržování čistého kódu. Poskytuje úvod do refaktoringu, testování a srovnání využití možností refaktoringu v moderních vývojových prostředích oproti projektivním technologiím.

Klíčová slova refaktoring, projektivní technologie, generování kódu, testování softwaru, Java

Abstract

The thesis focuses on the use of projective technologies for refactoring and analysis of current approaches and tools helping to keep code clean. It also provides an introduction to refactoring, testing, and comparison of using development environments for refactoring versus using projective technologies.

Keywords refactoring, projective technologies, code generation, software testing, Java

Contents

Introduction	1
1 State-of-the-art	3
1.1 Technical debt	3
1.2 Importance of writing tests	4
1.3 Refactoring	5
1.4 Codiscent's projective technologies	8
2 Analysis and design	11
2.1 Refactoring using IDEs	11
2.2 Types of refactoring and rules to implement them	13
3 Realisation	19
3.1 Working with GES	19
3.2 Refactorings	21
3.3 Formatting the code	24
Conclusion	27
Bibliography	29
A Acronyms	31
B Contents of enclosed CD	33

List of Figures

2.1	IntelliJ Idea refactoring example	12
2.2	Replace method with an object	16
2.3	Pull up field	17
2.4	Pull up method	18
3.1	Definition of rearranging rules in IntelliJ IDEA	26

List of Tables

2.1	Top 10 refactorings used in IntelliJ Idea.	13
-----	--	----

Introduction

As many of us already know, developing new software is a complex process. Project managers push the developers to add new features as fast as possible, developers want to make the new code maintainable and bug-free and clients want to keep the price low. This represents something called *Triple constraint of project management*. It is composed of three elements - time, quality, price - of which we only can choose two.

With refactoring, we save time in the long term, because adding new features to a well-maintained code base is easier. Because refactoring takes time, which could be used for further development of the project, it is not so worth in the short term. Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet it improves its internal structure.

The goal of this thesis is to study different types of refactorings found in the Martin Fowler's book Refactoring[1] and then explore possibilities of using Codiscent's projective technologies to perform those refactorings and demonstrate it on examples.

Those technologies are Projector Template Generator (PTG), Reverse Engineering Studio (RES) and Generative Engineering studio (GES). We could think of this software like some kind of black box, which input is the source code and a template describing the desired refactoring, and output is the transformed code.

All code transformation will be shown in the Java programming language. I chose Java, it tends to be a language that everyone understands to some degree.

In the first part of the thesis, I will provide an introduction to refactoring, then describe some types of refactorings which will be suitable to use with projective technologies. In the following part, I will form the templates to perform these refactorings and show the solution. In the last part, I will ponder possible future work and form a conclusion.

State-of-the-art

1.1 Technical debt

This is a metaphor that equates software development to financial debt. Imagine taking a loan versus saving money to buy a car. You either have to wait several months to save enough money, or take a loan and buy the car instantly, but you have to also pay interest. Needless to say, the collected interest could be so high, that it makes the repayment impossible.

The same can happen when developing software. By taking shortcuts, not writing tests, delaying refactoring and so on, the development process could be temporarily sped up. But with the initial time savings comes the interest - the future work. And the same as the interest in the financial loan could be so high it cannot be repaid, the technical debt could make the project unprofitable or even fail. A little debt could speed up development, but must be planned thoughtfully and promptly repaid.

Definition 1.1.1 (Technical debt) *is a concept in programming that reflects the extra work that arises when shortcuts during development are taken instead of applying the best solution[2].*

For example, programmers skip writing tests to save time, but it will gradually slow the project down until the tests are written - the debt is repaid.

Technical debt can have many causes. One of them is business pressure - even the greatest programmers can create debt if they are working under unrealistic project constraints. Rolling out features before they are finished then results into many bugs and patches.

Another important cause is not understanding the technical debt. Sometimes the management doesn't understand, that with the technical debt comes the interest, which will slow down the development as the debt accumulates. This could make it hard to dedicate time for refactoring because management

doesn't see the value of it.

Lack of documentation is another cause. It slows down the introduction of new people to the new project and could pause the development if too many people leave.

There could be other causes like incompetence of the developers or lack of monitoring of the work, but the most important one (for the scope of this thesis) is delayed refactoring. Because the project requirements are constantly changing, some parts of the program may become obsolete, badly designed or bulky. On the other hand, programmers every day write code that interacts with the obsolete parts. The longer the refactoring is delayed, the more dependent code will have to be written in the future. Technical debt is often associated with refactoring, because it is one of the most common practices to repay it, along with writing tests.

1.2 Importance of writing tests

Before we dive into refactoring, let me mention one important part of it. It is writing tests, especially unit tests.

Because we want to develop code in the fastest way possible, we need to eliminate as many errors as possible. Tests are an easy way to check if the code we wrote is working correctly. Testing our code also could lead to better code design. Creating tests before the implementation helps to specify the particular code requirements, which could be followed during the development process

By compiling and running tests after every change to the code, many hours could be saved by identifying errors early. Also, well-designed tests tell us which functionality isn't working and where the error could be located. They need to be easy to run, otherwise, people will be discouraged from running them often. Some build tools (like Maven for example) run tests automatically with every build of the application if certain project structure is followed.

Because refactoring, by definition, changes the structure of already working code, we need to be sure that we didn't break anything.

1.2.1 Testing levels

There are four main levels of software testing: unit, integration, system, and acceptance. These are performed in different stages of the development life-cycle.

Unit testing is a level of software testing where individual components/units of software are tested. The purpose is to ensure that every unit of software is working as designed. A unit is the smallest testable part of any software, which only takes a few inputs and usually a single output. The unit might be a procedure or a function in procedural programming, or a method belonging to a class in an object-oriented one. Unit testing is often performed by the

developer themselves. The execution of unit tests is often automated with build tools. There are many benefits of creating these tests. If the unit tests are properly written and ran as often as the code is changed, they will be able to promptly catch any bugs introduced in the last software changes. Also if the bugs are caught early, the cost of fixing them is lesser than if they are caught later during the development cycle. The code is more reusable because to make unit testing possible, it needs to be modular [3].

Integration testing tests how individual software modules work together. Different software modules are combined together and are tested as a group. This kind of testing is performed by developers or testers.

System testing is performed on a completely integrated system. It allows checking the system's compliance with the requirements. It involves load, performance, reliability and security testing. It evaluates both functional and non-functional needs for the testing. This testing level is typically performed by testers.

Acceptance testing is the final level of testing and is performed before making the system ready for use. Internal acceptance testing is performed by the members of the same organization that created the project but not by the ones that were directly involved with it. External acceptance testing is performed by the customers or the end-users [4].

From the description of testing levels above, the most interesting levels of testing for refactoring purposes are unit testing and to some degree integration testing.

1.3 Refactoring

The main purpose of refactoring is to fight technical debt. M. Fowler defined refactoring as follows:

Definition 1.3.1 (Refactoring) *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*

It could be said that refactoring is just a process of cleaning the code. But there is more to it. Refactoring offers patterns on how to effectively and target fully clean the code. This patterns then could be partially automated by various IDEs or, in this case, Codiscent's software.

1.3.1 Benefits and drawbacks

Without refactoring, the design of the program will decay. As the program code is being changed. the code loses its structure. It starts to become hard to understand, more bugs appear and due to the poor structure, development is slower. The solution to these problems lies in refactoring.

The first benefit is **Readability**. Programming is not only about telling the computer what to do, but because code is constantly changing (eg. bug fixing, feature adding), the code needs to be easy to understand. It is common that multiple people are working on the same parts of the program. Or maybe I will want to change the code myself after a few months. The faster the code is understood, the faster the work can begin. These time savings are well worth the small price of refactoring. There is one other benefit to understandability - because the code is more clear, it is easier to spot new things about the design of the program, which could lead to further refactorings.

The next benefit is **Faster bug finding**. This goes hand in hand with the previous benefit. Because code is easier to read, bugs could be spotted faster. Another thing is, bugs could be found during the process of refactoring.

Another of the benefits is **Extensibility**. Due to the good design of the program, it's easier to extend its capabilities and the application could be more flexible towards future changes.

The last benefit of refactoring is **Faster development**. This point may seem somewhat counterintuitive because from previous points it seems refactoring is mainly about improving the quality of the program because we spend time to change it without adding any new features.

But the point of having a quality design is to allow rapid development. With poor design, we can progress faster for a while, but then the bad choices will start to slow the development process. More time is spent to find and fix bugs. New features need more coding and all changes take more time due to worse understanding of the code.

There is one main drawback to refactoring - **Resources**. Because by definition, refactoring doesn't change observable behavior of the code (no features are being added), it could be hard to justify spending resources on it. In the long term, refactoring pays itself off. In the first place, adding features to a well-maintained program takes much less time than to a badly maintained one. In the second place, developers are able to understand code more quickly, which leads to better productivity. And in the third place, it helps find bugs faster.

In the short term, refactoring doesn't offer that many benefits, because if a project has a tight deadline or is running out of money, the time is better spent on development.

Many of the refactorings can be automated. In this thesis I will use Codiscent's RES to load the original source code into an inner representation and the use the GES to modify that structure and generate refactored code.

1.3.2 Code Smells

Before we can refactor, we first need to identify which parts of the program needs it. Those parts are called code smells.

Definition 1.3.2 (Code smell) *a piece of code that violates fundamental design principles and negatively impacts design quality.*

Some examples of code smells are:

- Code duplicates
- Long methods
- Complex conditions
- Dead code
- Switch statements

Today's IDEs offer various tools to ease the refactoring process. They often offer real-time code analysis, which is useful for detecting code smells as soon as possible. IDEs are especially good at finding code duplicates or code that is never executed.

The sooner the code smell is discovered the better - it is much easier and cheaper to remove. Other refactoring tools range from renaming variables, extracting methods to generating code. It is as simple as selecting a piece of code and choosing the operation that will be performed and add a few arguments. The goal is to make the process as much automated as possible, so the refactoring is faster and fewer errors are made.

1.3.3 When to refactor?

After seeing all the refactoring benefits, you may start to wonder when should be the refactoring performed. In almost all cases, it should be done during code development in small bursts. When working on something, you notice some code smell and remove it, and in turn, the refactoring helps you do the original work. There are a few cases which can help you tell when you should refactor[5]:

- **Rule of three**

Rule of three is a simple guideline consisting of three steps. When you do something for the first time, you just do it. When the same thing is done for the second time, think about the duplication, but do it anyway. But when you encounter it for the third time, you refactor.

- **Adding a feature**

This is the most common time to refactor. As you think about the piece of code you are modifying, refactoring it will help you understand it better and you can make it more readable. This will also help some future developers working on the same piece of code.

Another thing is, it will be easier to add the new function to the refactored code.

- **Fixing a bug**

Most bugs are found in the dirtiest parts of the code. By refactoring them, you almost make them discover themselves. Some bugs could even be fixed by the refactoring itself.

- **Code review**

Code review is a nice way of checking and tidying up the code before it becomes available to the public. It is a good idea to perform a code review with a partner, original author and a reviewer make a good pair. The reviewer suggests changes and then they discuss them together. This way it is more effective to spot code smells and the consultation leads to a better solution.

- **When facing legacy code**

The launch of a project is no reason for development to stop. As the project grows more bugs are introduced, the code is slower or maybe we want to add new functionality. When we get to the legacy code it is important to not start refactoring right away and fix it. Firstly we need to get acquainted with the code and understand it because there might be dependencies we are unaware of.

There are many different techniques and approaches to code refactoring. One of them is the widely used Red-Green-Refactor approach used in Agile test-driven development. This way the refactoring is divided into three steps. [6]

Red is always the starting point of the cycle. In this phase, the tests are written to inform the implementation of a feature. These tests pass only when the feature's expectations are met. Because the new feature has not been implemented yet, it typically glows red, hence the Red phase.

The Green phase is about the implementation of the new feature and making the tests written in the red phase pass. The goal of this phase is to find a solution without worrying about the speed and optimization of the implementation. Once we are in the green (the typical color of passed tests), the work on optimizing the implementation can start.

In the Refactor phase, we can think about better implementations of our code and refactor the code to make it more readable.

1.4 Codiscent's projective technologies

Codiscent Ltd. is a consulting and software development company that specializes in software for code generation. Codiscent claims that their software helps its customers to improve every measurable dimension of software development, also called The Triple constraint [7]. Codiscent's business is a commercial software development and consulting services. They offer various

levels of cooperation and tool and methods usage according to the situation and the estimated frequency of updates to the generated system.

1.4.1 Codiscent's products

The main Codiscent product (for the scope of this thesis) is **Generative Engineering Studio**. This is an IDE facilitating the building and managing of the assets associated with generative development projects. CodiScent claims that working with GES can reduce development costs by as much as 60% [8]. It processes the transformation of objects, from one form to another using orchestration commands as well as generative engineering framework functionality like GES and RES. It manages all artifacts needed for the code generation — models, specification data sources, templates, and extensions. It groups related artifacts into a solution. This product consists of several components.

- The reverse engineering part - is used to tokenize source code, identify the linguistic patterns in it and create a token structure representing the elements of the source code and their relationship to one another, which could be then used as a base for code generation with GES for example. The tokens produced by RES could range from something small as a variable to whole function bodies, for example
- A set of **transformation commands**, which are used to modify the model created with RES through a series of transformations - eg. extending it or projecting it to a new one
- And the forward engineering part - **Projective technologies** that are used to generate code out of these models.

The heart of CodiScent's delivery system is the Projector Template Generator (PTG). It employs clear, intuitive and flexible templates, which can be then used to generate output in any format - code, text or data. PTG output is independent of the rest of the CodiScent platform. Example of transformation: the GES can create an object set from Java source code, identify common fields among classes that were extended from a superclass, move those fields to the superclass and generate corresponding source code. Apart from Generative engineering studio and Reverse engineering studio, other Codiscent tools contains: Graphical toolkit used to define, create and use diagrams to represent modes requirement, Solution Modeling and Integrity Support with the components that support defining, populating and managing solution models and Control Center Generator (CCG) Generates solution-specific management applications that allow users to manage their own solutions. [9]

1. STATE-OF-THE-ART

The main benefit of projective technologies is obvious - programmers have to write less code, the generated code is more uniform. With the right design, the templates could also be reused with another set of specification data with almost no effort. Another big benefit is reduced maintainability because when the specification changes, the only thing needed to be done is to regenerate the solution.

The main drawback of projective technologies is that they are not suitable for every problem. [10]

Analysis and design

2.1 Refactoring using IDEs

Current IDEs offer numerous ways to refactor our code, varying from a simple renaming of objects to complex restructuring of classes. The main benefit of using the IDE refactoring methods is their simple user interface, which allows performing desired operations easily and quickly. This is a very important feature because the key to refactoring frequently is the ease of use. Example of how IntelliJ IDEA handles refactoring can be seen in figure 2.1. Other advantages of IDEs is the possibility to preview the output and simple roll-backing when the refactoring is not successful. Some environments are clever enough to predict code conflicts before the changes are made and offering means to resolve them[11]. But the simple interface might not be able to provide the refactoring capabilities needed, and this is where we might find a use for Codiscent's projective technologies.

The Codiscent's technologies provide us with tools to produce a set of scripts, which can be executed and run any time we need it to transform the source code.

The main benefit of taking this approach is the ability to tailor it to the problem we are solving. But there is one big drawback of using this tool to solve the refactoring problem we are facing - the time commitment to create the process. There are several questions we should ask ourselves before working on the automation of our solution. Is the problem we are facing unique, or are there multiple occurrences, which could be solved with the same template? Would be the refactorings templates future proof? Are there any better other ways suited to solving our problem? With answers to these questions, we can consider if creating the scripts is worth it. When we are encountering the same code smells frequently, apart from reflecting on our code design choices, the automation of refactoring could be the solution we are looking for. We need to keep in mind that it is vital to design the solution to be universal,

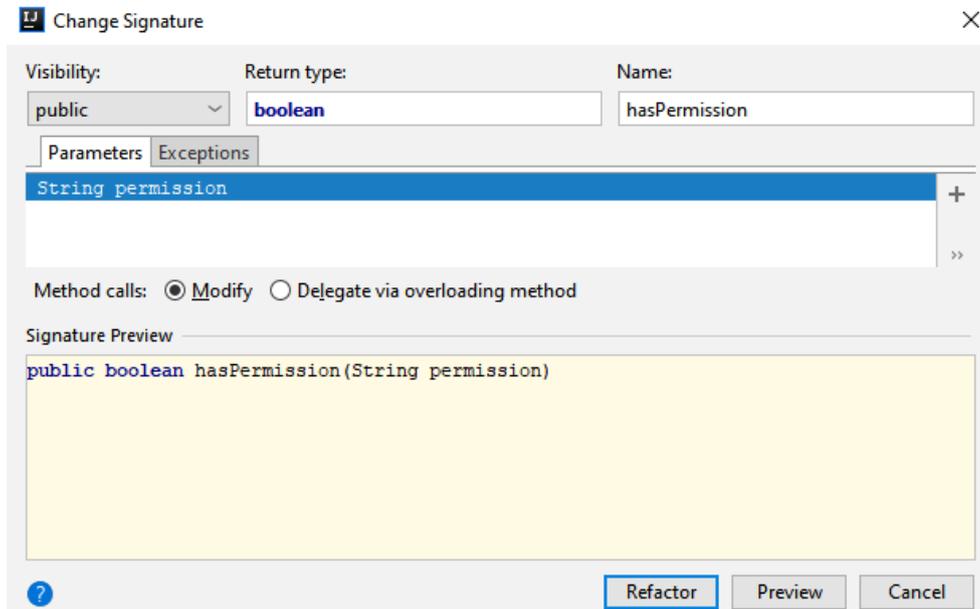


Figure 2.1: IntelliJ Idea refactoring example

with only needing small changes to be able to be used to solve our current problem. We don't want to spend more time automating the solution than what it would take to change the code manually. Another thing to keep in mind is we need to test these scripts thoroughly, as well as the transformed source code because we want an effective, easy to use solution, not introducing more bugs to the system.

There is one more challenge when we refactor via scripts and this is code selection (we need to identify which blocks of code are intended to be moved and so on). This is the area where integrated development environments truly shine. Because of their user interface, there are many ways to mark the code intended for refactoring. One of them is simply right-clicking the target object and selecting the desired operation, the other is selecting a block of code. When writing scripts, the blocks of code need to be passed GES, and this is possible in two ways. The first is to identify the code with the indexes of the start and end of the code block. This approach is faster and easier to implement, but requires more reading time, and is not very future proof - one small change to the source code could shift the refactored block and then the indexes are pointing to invalid positions.

The better way to handle this is with the usage of a regular expression. This approach is not as easy to implement as the first one, because we need to create the regular expression uniquely describing the target code block. The simplest way is if the code block contains some unique name or expression.

The other way to do this is to use some dummy tokens - for example, some unique comment in code describing the start and the end of the block.

The table 2.1 lists the most popular refactorings used in IntelliJ IDEA [11]. As we can see, the IDEs are used to do the simplest refactorings

Name	Keyboard shortcut
Safe delete	Alt+Delete
Copy/Move	F5 / F6
Extract Method	Ctrl+Alt+M
Extract Constant	Ctrl+Alt+C
Extract Field	Ctrl+Alt+F
Extract Parameter	Ctrl+Alt+P
Introduce Variable	Ctrl+Alt+V
Rename	Shift+F6
Inline	Ctrl+Alt+N
Change Signature	Ctrl+F6

Table 2.1: Top 10 refactorings used in IntelliJ Idea.

2.2 Types of refactoring and rules to implement them

In this section, I will describe several types of refactoring and explain why to do them and list examples as well as rules to use them. The rules follow refactoring patterns based in [1]

2.2.0.1 Extract method

This is one of the most frequently used types of refactorings. It is used for moving a part of code, which forms a logical group to a new method.

Typically it's used to divide long methods to more sub-methods to increase readability and prevent duplicities. Because long methods are harder to understand, we increase the readability by dividing the code into smaller parts. Also if the names of the new methods exactly describe their purpose, we may decrease the need for comments.

Unfortunately, there are some complications we can encounter - local variables. If we refactor without accessing local variables, the deed is trivial. If the variables are read-only, they are passed as a parameter to the new method. But if the local variables are changed, the refactoring couldn't be done in some cases. Temporary variables, that are used only within the part of the code that is extracted are extracted to. But if the variable is further, the new method must return its value and then it is assigned back to the corresponding variable.

How to refactor:

2. ANALYSIS AND DESIGN

1. Create a new method and name it in a way that makes its purpose self-evident.
2. Copy the relevant code fragment to the new method. Delete the fragment from its old location and put a call for the new method there instead.
3. If the variables declared before the extracted code are changed and used afterward, the result needs to be returned to that variable (for simplification, let's assume, that no more than one variable used later in the code is changed in the method, otherwise the problem could be solved by returning an object and assigning the values or pass addresses of those variables to the new method).

```
public void printMovieDetails(int pricePerTicket, int visitors){
    int profit = pricePerTicket * visitors;
    System.out.println("Name: " + this.name );
    System.out.println("Released: " + this.yearOfRelease);
    System.out.println("Profit: " + profit);
}
```

After the calculation of profit was moved to a new method:

```
public void printMovieDetails(int pricePerTicket, int visitors){
    int profit = getProfit(pricePerTicket, visitors);
    System.out.println("Name: " + this.name );
    System.out.println("Released: " + this.yearOfRelease);
    System.out.println("Profit: " + profit);
}

private int getProfit(int pricePerTicket, int visitors) {
    return pricePerTicket * visitors;
}
```

2.2.1 Replace temporary variable with function call

The problem with temporary variables is that they are temporary and local. They are only visible from the current scope, which leads to the writing of long methods. The solution to this is in moving the expression calculating the value of the variable to a new method and any time the old variable is used, we call the new method.

Another benefit of this refactoring is that the new method could be called from other parts of the program. Replacing variable with a function call is often a necessary step before Extracting a method

```
double calculateTotal() {
    double basePrice = quantity * itemPrice;
    if (basePrice > 1000) {
        return basePrice * 0.95;
    } else {
        return basePrice * 0.98;
    }
}
```

Now a function is created for the calculation of base price and it could be reused somewhere else in the code

```
double calculateTotal() {
    if (basePrice() > 1000) {
        return basePrice() * 0.95;
    } else {
        return basePrice() * 0.98;
    }
}

double basePrice() {
    return quantity * itemPrice;
}
```

2.2.2 Divide temporary variable

Many temporary variables serve to store the result of some calculations. If they're assigned value more than once, it could mean they are being used for multiple purposes. Because this is confusing for the reader, these variables should be replaced with multiple variables, each serving its dedicated purpose.

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

By splitting the variable "temp" into new self-explaining ones, the code is much more readable.

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

2.2.3 Replace method with an object

This type of refactoring is used when we have a long method with many local variables, that it is impossible to use the Extract method.

The principle is, we replace the whole method with an object - variables that the method was using are now passed as an argument to the constructor of the new object and a method "calculate" is created, which contains the body of the original method.

Now all local variables are the new objects attributes, we can use the Extract method as we like without worrying about passing parameters to the new method. In the place where the original calculation was located, we create a new object and call it's method "calculate"

```
class Order...{
    double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation;
        ...
    }
}
```

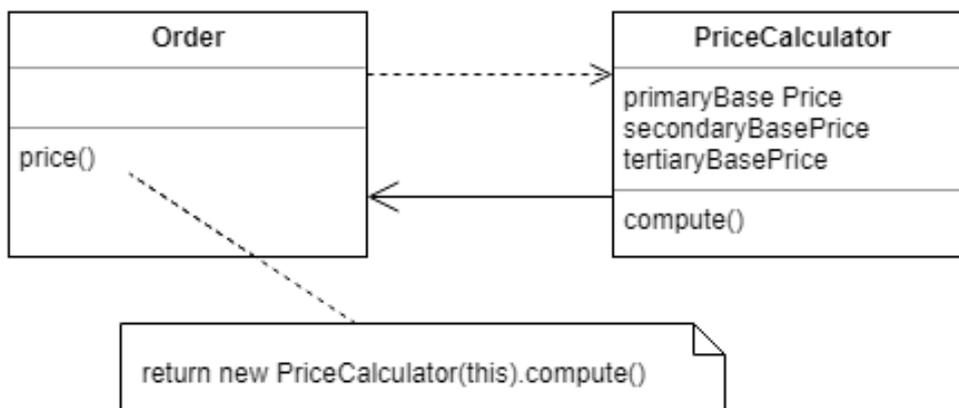


Figure 2.2: Replace method with an object

2.2.4 Move method

Moving method is used when a method uses elements of a different object more than that of its own. Along with the Extract method, this is one of the building blocks of refactoring.

When we design the program, we hardly get it right from the start. Then when we notice that a class does many more things than what it was designed for, the corresponding methods are moved to the appropriate class. Refactoring Move field is also used often with Move method because some of the original fields are used by the moved method and it makes sense to move them to the new class too, otherwise, they have to be accessed by other means.

2.2.5 Pull up field

When multiple people work on one project, they sometimes develop subclasses of one superclass separately. As those subclasses grow on their own, new fields appear, which can be duplicates in both classes. As soon as they are identified, they can be pulled up to their superclass.

In order to perform this refactoring, these fields need to serve identical purposes.

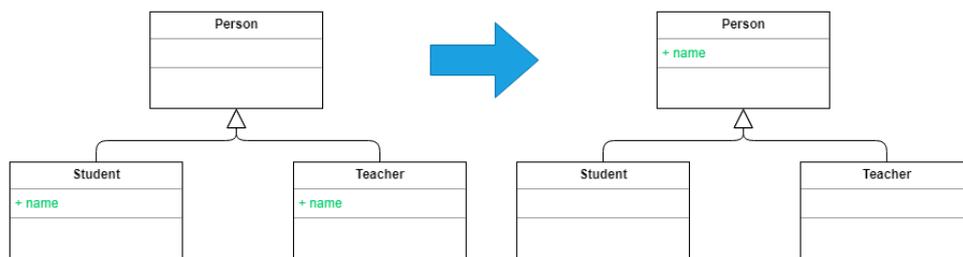


Figure 2.3: Pull up field

How to refactor:

1. If the fields have different names, choose a suitable name and rename them
2. Create the field in the superclass. If the original fields were private, now the field needs to be protected, so the subclasses can access it.
3. Remove the field from the subclasses

2.2.6 Pull up method

This refactoring is used in the same scenario as the previous one. When there are duplicate methods in subclasses, it is suitable to move them to their parent. But there could be few things that will block this refactoring.

Firstly, the method uses attributes located only in the subclasses - the solution is to pull up those attributes.

And secondly, the method uses other methods located only in the subclasses.

2. ANALYSIS AND DESIGN

We solve this by pulling this method up first, or the pulled up method could be defined abstract (this will make the superclass abstract if it isn't). It's important to consider which option is the best or if it is even possible, to perform the refactoring.

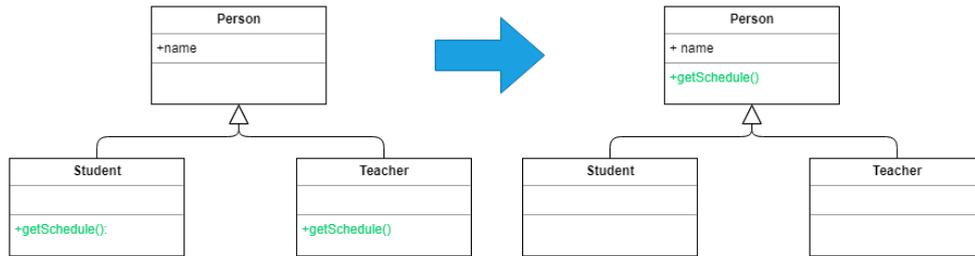


Figure 2.4: Pull up method

How to refactor:

1. Investigate similar methods in superclasses. If they aren't identical, format them to match each other.
2. Copy the method to the superclass. If this method uses attributes or methods from subclasses - solve this by pulling those attributes or methods from subclasses. If those methods can't be pulled up, the superclass could be made abstract.
3. Remove the methods from the subclasses.

Realisation

GES is a generative software development platform used to create generative solutions like application generation, software conversion, and refactoring. It is very flexible. A solution can be easily extended to perform refactoring on different files by altering the names or regular expressions, for example.

Firstly, we list some of the frequently used commands used to work with GES and then show how the GES platform could be used on some simple refactoring examples and then head into more complex ones. The commands could be divided into several categories

3.1 Working with GES

The transformations we are working with are called process-scripts. The processing of these scripts is divided into two phases - compilation and execution. The scripts are compiled into a set of compilation objects and then it is executed by processing the compilation objects

3.1.1 Useful editor keys

F2 Presents the object compilation space

F5 Starts a process compilation and execution

F6 Presents the current objects with their attributes. This is a very useful tool for debugging the scripts, especially with the *Stop* command.

F11 Recompiles a process and executes

3.1.2 Controlling the execution

The execution can be controlled by the following commands. They are used primarily for the creation of new processes, for example, pausing the execution

3. REALISATION

to look into the contents of the data structures to ensure they contain the desired information or during the debugging process.

Stop: Completely stops the execution of the process

Pause: Pauses the execution of the process. By pressing F5 the execution will proceed.

Show Prepares selected object to be shown after the show data command (F6). Otherwise all objects are shown.

Message Prints a message to the message pane

3.1.3 Manipulation with objects

One of the essential parts of the GES transformation is manipulation with objects (along with templates for working with the file structure and generating output). To be able to generate the desired source code, we transform and extend the original objects until we get the result object, from which we can generate the code we wanted.

There are several ways to create a new object. One is with the Define keyword and the other is Project. Define creates a completely new object with predefined attributes and values.

GES's objects could be created with the following commands: Define and Project. The main keyword for creating new objects is "Define". Because of its flexibility, we can create objects with as many fields as we would like and set their values all in one line of code.

```
Define(a, b, c) = (1, 2, 3)
```

The second way of creating new objects is by using the "Project" keyword. This command creates a new model by copying it from an existing one. The process of projecting a new object is very often modified with the extend command. What happens is that the GES takes the original object, adds new attributes specified with the extend command and then projects a new object from it. Using Extend as a standalone command will edit the original object, but if it is used in combination with Project, only the result will have the new attribute.

```
Project CONTENT from SOURCE extend sourceCode = ReadFile  
(fileName);
```

3.1.4 RES and GES template structure

The RES functions are used to reverse engineer data from a given code. When used, the function returns an array of variables containing the code fragments

that were defined in the RES template. The elements that will be extracted are marked with @ sign and can be accessed by the name specified in the template. The templates work similarly like regular expressions, but with more features.

The GES templates are very similar to the RES templates. But instead of content being extracted from elements marked with @, the data is inserted into them, before generating the output.

3.2 Refactorings

3.2.1 Extract method

The first refactoring explained will be the extraction of a method from a simple statement.

```
Define X(inputFileName ,outputFileName) = [C:\Users\Tom\
IdeaProjects\extract_method\src\Movie.java , C:\Users\
Tom\Documents\BP\ReleaseCode\Sample001.out.txt ];
```

```
Extend X content = ReadFile(inputFileName);
```

Firstly, we create a new object with two fields - inputFileName and outputFileName, both linking to the corresponding file locations. Then a new field - content - is added to the object X, with the content of the input file. The reading of input files is very convenient. We just use ReadFile function and the GES handles the rest.

```
Project Y from X extend [Element , Expression ,
@ExpressionBlock__meta]=ERES( content , '[
@ExpressionBlock int $Element$ = $Expression:..+?$ ;]
' , ' ' , 'N') ;
```

After that, we create new object Y from the X with several new attributes(the name of the attributes must match the one from the RES template), which values are taken from the function ERES.

The first argument of the RES function is the source - the input file in this case. The second one is the RES template that will be applied to the source. The RES template can be defined inline in a simple string, as you can see here, but if it is more complex, it could be moved to a separate file, where we have more options to work with it. The next attribute is the view name to extract out of RES in case not the default. The last attribute is the edit flag, if it is set to true, RES editor will be opened at run time. The __meta suffix captures the location of the code fragment extracted by RES in the original source code in the RES meta format. The RES meta format is in the following structure:

```
id~~~location:length:rel_location
```

example: I10000~~~292:39:292~~~0

In the next step we create the function body:

```
Extend Y ElementCapitalized=CapitalInitial(Element) ^
    [ExpressionVariable] = ERES(Expression, '
        $ExpressionVariable$ ', '', 'N') ^
    FunctionBody = GES('@EMFunctionBody', '', 'N') ^
    FunctionCall = GES('int $Element$ =
        get$ElementCapitalized$ ([
            $ExpressionVariable$ ^,]);', '', 'N')
present [-ExpressionVariable];
```

We process the result of reverse engineering. The name of the variable is capitalized, so it could be used to create the function name. The attribute "[ExpressionVariable]" is an array reverse-engineered from the original expression, which contains all the variables from it. Next, GES is used to generate the body of the function. The GES function signature is very similar to the RES signature. Here you can see the referencing of GES template in an external file. The @ sign tells the studio to look for a file with the name 'EMFunctionBody' instead. Finally, we can construct the function definition using the previous variables and the function call.

The next step changes the original expression to call the newly created function:

```
Extend Y xcnt = RSBI(cnt ,@ExpressionBlock_meta ,
    FunctionCall) [];
```

The RSBI function(replace set by index location) replaces the original expression with the created function call. The arguments are - text to change, starting position - this could be an index or RES meta format - and the new value. The final step is to append the function definition to the end of the original class:

```
Project O from Y extend
[ClassHeader ,ClassBody]=ERES(xcnt, 'public class
    $ClassHeader$ {[ $ClassBody:.$ ]}' , '', 'N') ^
Output = GES('@EMOutput', '', 'N') ^
o = WriteFile(outputFileName, Output);
```

Using RES we capture the class header and body from the updated class, which are used by GES to generate the new class file. The output is then written to the file.

3.2.2 Pull up members

The following example recognizes common fields and functions from child classes and moves them to a parent class.

We prepare object A containing the input file names. Then we create new object B which will manage file locations and file contents. Loading multiple files at once is shown here by creating a list of file names and then passing it to the ReadFile function.

```
Define A( fileList )=[Person | Student | Teacher ];
Project B from A extend files='src-data/pull-up-members/
  src/'+fileList+'.java';
Extend B content = ReadFile( files );
```

When we have the file contents ready, we create new object D containing the instances of the derived elements and parse out the fields and methods using @ExtractElementsRes RES template

```
Project D from B where fileList not in ( 'Person ' )
  present [ fileList , content ];
Extend D [ @CodeBlock_ meta , CodeBlock , Accessor , DataType ,
  MethodName , AttributeName , ParameterSet , Body , Rest ]=ERES
  ( content , '@ExtractElementsRes ' , ' ' , 'N' ) ^
Element=MethodName | MethodName <>' ' ^ Element=
  AttributeName | AttributeName <>' ' ;
```

Then we find the common elements to be pulled up. D1 contains unique names of all the elements from the children classes and D2 contains names of all the common fields and methods

```
Project D1 from D where Element <>' ' present [ fileList ,
  Element ];
Project D2 from a<D1> join b<D1> link a.Element=b.
  Element map *:a.* ^ bfl:b.fileList filter fileList <
  bfl and Element <>' toString ' present [ Element ];
```

The next step is to remove the common elements from the sub-classes by replacing each occurrence with an empty string and then save the updated classes to corresponding files.

```
Project D3 from a<D> join b<D2> link a.Element=b.Element
  map *:a.* ;
Extend D3 updatedContent = RSBI( content , @CodeBlock_ meta
  , ' ' ) [ fileList ] present [ fileList , updatedContent ];
Extend D3 outputFileName='src-data/pull-up-members/
  output/'+fileList+'.out.java ' ^ o = WriteFile(
  outputFileName , updatedContent );
```

When the sub-classes are updated we start working on changing the super-class. The new fields and methods are added to the class. We parse the class headers and body into new object E. New object D4 is created and contains the elements to be pulled up. then D4 is extended with generated code for

corresponding elements. Then we pass the methods and fields code to E by cross joining it with the D4 and generate the output the same way we did it in the last example.

```
Project E from C where fileList in ('Person') extend
[ClassHeader,ClassBody]=ERES(content,'public class
  $ClassHeader$ {[ $ClassBody:.$ }','','N');
Project D4 from a<D> join b<D2> link a.Element=b.Element
  map *:a.* present [Element,CodeBlock];
Extend D4 FunctionBody = GES(
  '// $Element$
  $CodeBlock$
  ','','N') present [FunctionBody];
Project E from E cross D4;
Project O from E extend outputFileName='src-data/pull-up
  -members/output/'+fileList+'.out.java' ^
Output = GES('@PullUpMembersGesOutput','','N') ^
o = WriteFile(outputFileName,Output);
```

At the beginning of the script, we used an external RES template. The script served to parse the elements before it was decided if they would be pulled up. The elements can be of two types, either methods or fields

```
[@@CodeBlock
[$Accessor:(private|public)$ $DataType:[\w<>]+$
  $MethodName$ ([ $ParameterSet:.$ ] ) {[ $Body:.$ }
  ]} ]
[$Accessor:(private|public)$ $DataType:[\w<>]+$
  $AttributeName$ $Rest:[^;]+;$ ]
]
```

3.3 Formatting the code

After the GES generates the desired source code, the result is rarely formatted. Although the result formatting could be solved with clever placement of whitespaces to the GES template when developing the scripts, it is not a good thing to do. It makes the templates unreadable and is very time consuming because the whitespaces need to be added by trial and error.

Fortunately, we could solve this issue far more easily and elegantly- with the use of code formatting programs. Although it could be done manually, and it is much simpler and easier when the code sample is small enough, it is better to use an application specifically developed for this purpose. This practice helps us ensure the code formatting is the same across our application, is faster when more than a few lines of code need to be changed and could be done just by pressing a keyboard shortcut. Many development environments have

this functionality built-in or accessible via external plugins. The advantage of using tools to beautify our code is quicker

3.3.1 Formatting code using IDE

In IntelliJ IDEA we can use the "CTRL + ALT + L" keyboard shortcut format the selected text, or "CTRL + SHIFT + ALT + L" to format the whole file that is being edited. IDEA also supports the reformatting of whole directories or modules.

IDEA offers many ways to customize the reformatter settings. We can use marker comments to prevent the reformatter from formatting the code fragment encapsulated in them. Another useful setting is the preferred line endings and indentation. If we are not careful, developers may be committing files with different line endings and different indentations. After the reformatter is set to our liking, it could be exported to a file. The export feature is very valuable because it allows us to simply share it with other coworkers, enforcing the same code style across the project we are working on.[12] Many text editors have this functionality too, so it shouldn't be hard for anyone to find the right tool for the job.

3.3.2 Rearranging the code

Predefined class structure helps us reaching what we are looking for more quickly because based on the pattern, we know where to look for code fragments we need. Code rearranging is a practice that can provide great help when used right. All we need to do is define a structure of how elements should be organized. The general idea is to keep similar code fragments together - to have one group with setter and getter, another with overridden methods and so on. Another set of rules could be created for setting the order of fields and methods. For example, public fields will take the first place, protected fields will be in second place... IDEs also support the definition of rearranging rules, that keeps the code. On the picture bellow 3.1 you can see the default rules. The rules are applied to our code when the formatting is run

3. REALISATION

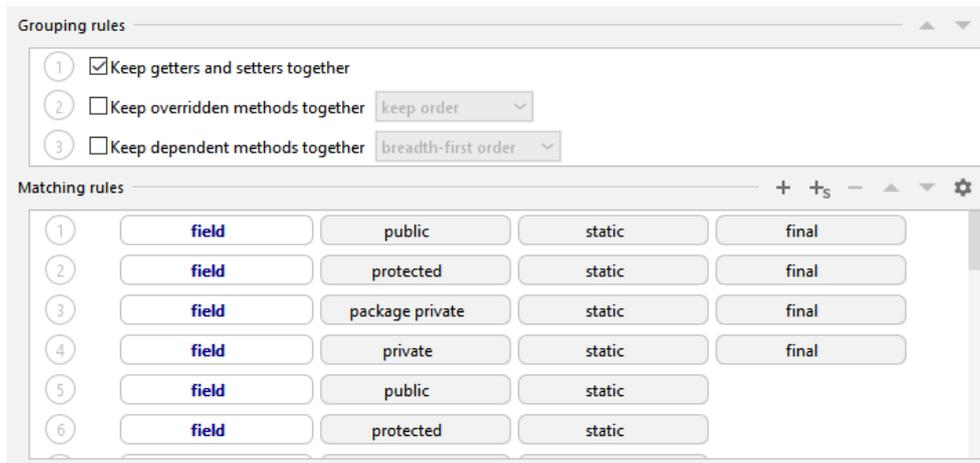


Figure 3.1: Definition of rearranging rules in IntelliJ IDEA

The rearranging of code could also be done using the GES. As in the Pull up members example, we can pick up the corresponding fields or methods and then generate update code with those members together. The code fragments will then be grouped together by the expression defined in the RES template. To create more categories, we need to write more specific expressions to describe the elements we are rearranging. In the following example, we can see GES code that would be used to identify public and private fields

```
[ $Accessor:( public $DataType:[\w<>]+$ $AttributeName$
  $Rest:[^;]+;$ ]
[ $Accessor:( private)$ $DataType:[\w<>]+$ $AttributeName$
  $Rest:[^;]+;$ ]
```

After the elements are extracted, the elements will be sorted based on the specification in the GES template

```
public class $ClassHeader$ {
    $PublicFields$
    $PrivateFields$
    $OtherClassMembers$
}
```

Conclusion

The goals of this thesis were to perform a review on current approaches and tools for refactoring, define rules for these refactorings and demonstrate how CodiScent's projective technologies could be used to perform these refactorings.

All of these goals were reached and with the help from CodiScent I was able to demonstrate the use of projective technologies on a subset of refactorings.

In the future, it would be possible to make the refactoring scripts more universal, to better suit practical use. Currently, it is much more efficient to perform refactorings directly through IDE than to take the time to write the scripts and test them. Although the refactoring is probably not the best application of projective technologies, it is a good way to introduce them to people who may be interested in this topic.

Bibliography

- [1] Fowler, M. *Refactoring: improving the design of existing code*. Addison-Wesley signature series, Boston: Addison-Wesley, second edition edition, 2019, ISBN 978-0-13-475759-9, oCLC: on1064139838.
- [2] Technical debt. [Online; accessed 27-April-2019]. Available from: <https://refactoring.guru/refactoring/technical-debt>
- [3] Unit Testing. Mar 2018, [Online; accessed 22-December-2019]. Available from: <http://softwaretestingfundamentals.com/unit-testing/>
- [4] Unit Testing. Mar 2018, [Online; accessed 22-December-2019]. Available from: <http://softwaretestingfundamentals.com/unit-testing/>
- [5] When to refactor. [Online; accessed 27-April-2019]. Available from: <https://refactoring.guru/refactoring/when>
- [6] Code Refactoring Best Practices: When (and When Not) to Do It. [Online; accessed 12-December-2019]. Available from: <https://www.altexsoft.com/blog/engineering/code-refactoring-best-practices-when-and-when-not-to-do-it/>
- [7] Codiscent.com – Better, Cheaper, Faster Technology from Codiscent. [Online; accessed 21-April-2019]. Available from: <http://codiscent.com/>
- [8] Codiscent Ltd. Application Development Using Codiscent Generative Technology and Methodology [online]. 2013, [Online; accessed 12-December-2019]. Available from: <http://ccm.fit.cvut.cz/wp-content/uploads/2013/10/CodiScent-Technology-and-Methodology.pdf>
- [9] Codiscent Ltd. Codiscent Tools [online]. [Online; accessed 12-December-2019]. Available from: http://codiscent.com/?page_id=296

BIBLIOGRAPHY

- [10] Červenka, J. *Utilising projective technologies for object-oriented development of WEB UI*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2015, an optional note.
- [11] The most popular refactorings supported in IntelliJ IDEA. [Online; accessed 06-January-2020]. Available from: <https://www.jetbrains.com/help/idea/refactoring-source-code.html>
- [12] Reformat and rearrange code. [Online; accessed 08-January-2019]. Available from: <https://www.jetbrains.com/help/idea/reformat-and-rearrange-code.html>

Acronyms

GES Generative Engineering studio

RES Reverse Engineering studio

PTG Projector Template Generator

IDE Integrated development environment

Contents of enclosed CD

readme.txt.....	the file with CD contents description
src.....	the directory of source codes
├ thesis.....	the directory of L ^A T _E X source codes of the thesis
├ ReleaseCode.....	implementation and GES files
│ └ src-data.....	refactoring input and output files
│ └ FileSet.....	refactoring templates
│ └ WorkflowOrchestrator.exe.....	executable of GES
text.....	the thesis text directory
└ thesis.pdf.....	the thesis text in PDF format