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: doc. Ing. Robert Pergl, Ph.D.

June 4, 2020

Acknowledgements

I thank my thesis supervisor, doc. 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

In Prague on June 4, 2020 …………………

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

Tato práce se zabývá refaktoringem kódu s využitím projektivních technologií společnosti Codiscent. Teoretická část obsahuje analýzu současných přístupů k refaktoringu a nástrojů k udržování čistého kódu. Poskytuje úvod do refaktoringu a testování. Praktická část je věnována demonstraci využití projektivních technologií na konkrétních příkladech refaktoringu.

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

Abstract

This thesis focuses on code refactoring with the use of projective technologies developed by the company Codiscent. The theoretical part contains an analysis of current approaches and tools designed to keep the code clean. It also provides an introduction to refactoring and testing. The practical part shows how projective technologies could be used for refactoring on specific examples.

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

Contents

List of Figures

List of Tables

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.
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.
With refactoring, we save time in the long term, because adding new features to a well-maintained code base is faster and easier. Because refactoring takes time, which could be used for further development of the project, it is not so worth in the short term. 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 it because it is object-oriented and 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, testing and projective technologies. Then I will describe some types of refactorings with rules on how to perform them. In the following part, the usage of projective technologies for refactoring purposes is shown on specific examples alongside the explanation of the code. In the last part, I will ponder possible future work and formulate a conclusion.

Chapter 1

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 the 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 could result in 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 too, 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 write code that interacts with the obsolete parts every day. 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 - **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 also helps to specify the particular code requirements, which could prevent bad design.
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 developers themselves. The execution of these tests is often automated

with build tools or development environments.
There are many benefits of creating these tests. If the unit tests are properly written and are executed 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 cheaper than if they are caught later during the development cycle. The resulting 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. It 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 the 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[1, p. 46]:

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 targetedly 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, with the following benefits:

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 the definition of refactoring it seems it is mainly about improving the quality of the program 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 technology 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 part of the program needs to be changed. Those parts are called code smells.

Definition 1.3.2 (Code smell) *a piece od 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 founding 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 the refactoring should be 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 to 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, you 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 - the original author and a reviewer make a good pair. The reviewer suggests changes which are then discussed 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 other 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]
The *Red phase* 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, the test statistics typically glow red, hence the Red name.
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.

Chapter 2

Codiscent's projective technologies

Codiscent Ltd. is a consulting and software development company that specializes in software for code generation. They offer various levels of cooperation and tools and methods usage according to the situation and the estimated frequency of updates to the generated system. Codiscent claims that their software helps their customers to improve every measurable dimension of software development, also called The Triple constraint [7]. Codiscent managed to do this by identifying and removing the common sources of inefficiencies frequently encountered in the software development process:

- **Repetitive coding** - increases the defect rate
- **Writing more code** - introduces more defects and fixing them takes more time
- **More time spent on programming and debugging** - Because more time needs to be spent on programing (as opposed to code generation), there is less time for design and architecture, if we want to preserve the same cost of the solution

Their solution to these problems is *Agile Model-Driven Development.*

Definition 2.0.1 (Model-Driven Development) *Model-Driven Development (MDD) is a technique in which a problem space is modeled and code to perform a function is generated across the domain of the problem.*

Let's demonstrate the MDD on example. We are facing a problem where we need to read and import tables from the database. To solve this we need to conform to their structure - column names, data types, and so on. Because doing this manually is tedious and time-consuming, we apply the MDD. Firstly, we create a template containing the code to perform the read, define

the structure of the database as input and then generate the code needed to perform the task.
Agile Model-Driven Development in Codiscent's version employs their templating language and work-bench, which supports iterative generative programming. MDD combats previously mentioned inefficiencies with the ability to work at a raised level of problem abstraction, resulting in a smaller code base to build and manage, accelerated development and reduced cost and the ability to focus on business and software architecture instead of spending more time on programming the solution.

2.1 Projective Technologies

Projective technologies is a term invented by Codiscent to describe it's *Reverse Engineering Studio (RES)* and *Generative engineering studio (GES)*. Currently, these technologies are still under development. The main Codiscent product (for the scope of this thesis) is **Generative Engineering Studio**. This is an IDE created by Codiscent that employs the usage of the projective technologies. GES facilitates the building and managing of the assets associated with generative development projects. Working with GES is designed to be completely consistent with CodiScent's methodology and can reduce the development costs by as much as 60% [8].

2.1.1 Generative Engineering Studio

Many generative engineering tools support a limited range of languages. Because they use to be tightly integrated with specific IDEs, their usage for code generation is limited. If our requirements aren't satisfied, these tools tend to be of little value for us. Codiscent's generative engineering solutions are different. GES supports numerous specification data sources (XML, database tables...) and supports generating source code of any programming language or anything which can be written as text. All of the interactions between specification data and templates are managed by GES.
GES processes the transformation of objects, from one form to another using orchestration commands as well as generative (resp. reverse) engineering framework functionality. It manages all artifacts needed for the code generation - models, specification data sources, templates, and extensions. Related artifacts are then grouped into a solution.
This platform consists of several components:

- *Reverse engineering part* - which extracts information from the source text and creates a model representing the data.

- *Transformation commands* - which are used to modify the model created with RES through a series of transformations.

- *Forward engineering part* - generative engineering technologies that are used to generate code out of these models.

The core of the generative engineering technology is *Projector Template Generator (PTG).* It is a text generator that employs clear, intuitive and flexible templates, which can be used to generate output in any format - code, text, or data. PTG's output is independent oofn the rest of the CodiScent platform. These templates are written in language developed by Codicent, which I describe in the implementation chapter of this thesis 4.

2.1.2 Reverse engineering studio

Reverse Engineering involves reading the source code of an application and decomposing it and using the result to generate a new version of the application, possibly in a different language. The Codiscent *Reverse Engineering Studio* provides the ability to interactively configure, test and refine parsing templates, which uses similar syntax to the generative ones, and then pass the extracted data to the *Generative Engineering Studio.*
RES technology 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. This structure could be further passed to GES where it could be transformed and used to generate new output. The tokens produced by RES could range from something small as a variable to whole function bodies, this depends on how the user designs the reverse engineering templates.

Example transformation: the RES can create an object set from Java source code, identify common fields among the classes that were extended from a superclass, move those fields to the superclass and generate corresponding source code using the GES templates.

2.1.3 Other Codicent products

Apart from Generative engineering studio and Reverse engineering studio, other Codicent tools contains: Graphical toolkit used to define, create and use diagrams to represent model requirements, Solution Modeling and Integrity Support with the components that support defining, populating and managing solution models and Control Center Generator that generates solution-specific management applications that allow users to manage their own solutions. [9]

2.2 Benefits and drawbacks

The main benefit of projective technologies is obvious - programmers have to write less code, which results in uniform code, fewer bugs are introduced and more time could be spent on designing and architecting. When properly designed, the PT templates could also be reused with another set of specification data with almost no effort or little changes to them. Codiscent states that they were able to redeploy a second instance of a C# solution, which took a week to build, in under 30 minutes [10].
Another big benefit is reduced cost of maintainabality because when the specification changes, the only thing needed is to regenerate the solution.

The main drawback of projective technologies is that they are not suitable for every problem. A significant part of the solution needs to be generated to justify the time investment into design of the templates and transformations[11]

Chapter 3

Analysis and Design

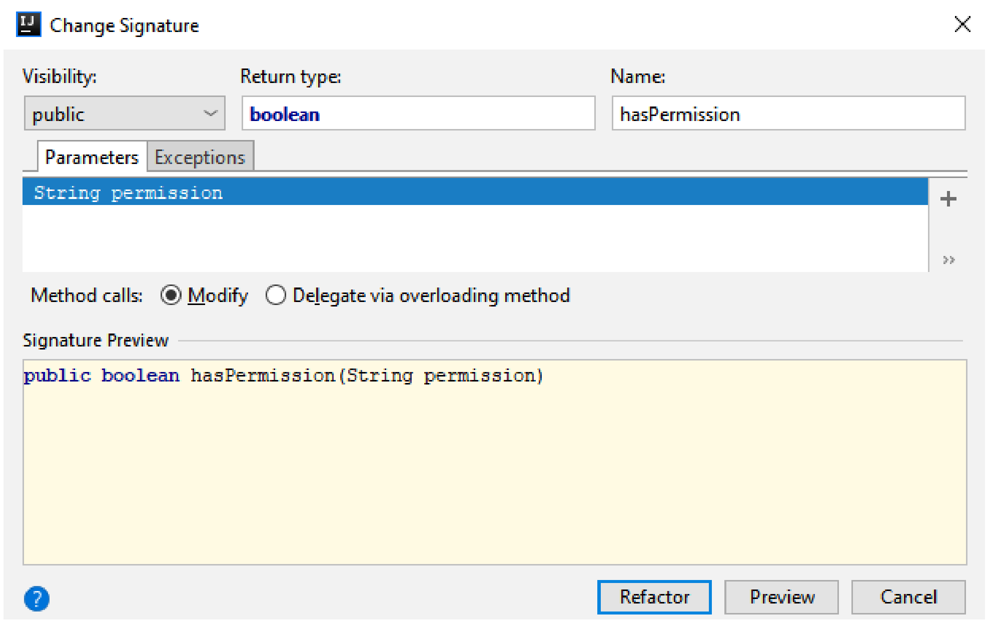
3.1 Refactoring using IDEs

Current IDEs offer numerous ways to refactor our code, varying from a simple renaming of objects to a complex restructuring of classes. The main benefit of using the IDE refactoring methods is their simple user interface, which allows performing desired operations easily, quickly, and with immediate feedback. 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 3.1. Other advantages of IDEs is the possibility to preview the output and simple rollbacking 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[12]. 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 them 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



Visibility: public
Return type: boolean
Name: hasPermission

```
String permission
```

Method calls: (● Modify) ○ Delegate via overloading method

```
public boolean hasPermission(String permission)
```

Figure 3.1: IntelliJ Idea refactoring example

to design the solution to be universal, 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 then 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 we face 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, these blocks of code need to be passed to 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 expressions. 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 to identify the code block by 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 3.1 lists the most popular refactorings used in IntelliJ IDEA [12]. As we can see, the IDEs are most frequetly 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 3.1: Top 10 refactorings used in IntelliJ Idea.

3.2 Types of refactoring and rules to implement them

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

3.2.0.1 Extract method

This is one of the most frequently used 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, which are used only within the code block that is extracted are extracted too. But if the variable is used further after the block,

the new method must return its value and then assign it back to the corresponding variable.

How to refactor:

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 those variables (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).

Let's see a practical example:

```
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);
}
```

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;
}
```

3.2.1 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 lies 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.
Let's see a practical example:

```
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;
}
```

3.2.2 Divide temporary variable

Many temporary variables serve to store the result of some calculations. If they are 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);
```

3.2.3 Replace method with an object

This type of refactoring is used when we have a long method with many local variables, which make using the Extract method impossible.
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 `compute()` is created, which contains the body of the original method.
Now all local variables are the new object's 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 its method `compute()`.

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

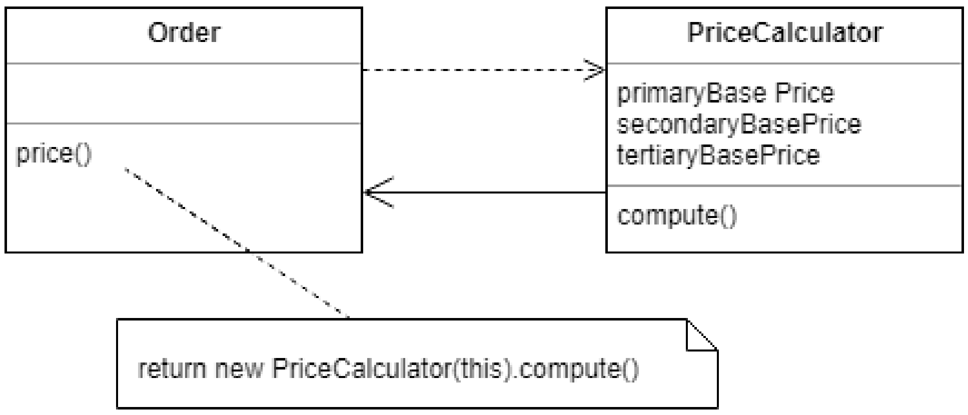


Figure 3.2: Replace method with an object

3.2.4 Move method

Moving a method is used when a method uses elements of a different object more than 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.

3.2.5 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.

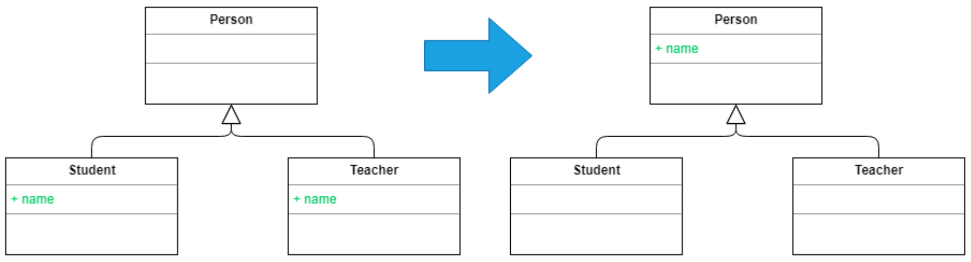


Figure 3.3: Pull up field

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

3.2.6 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. 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.

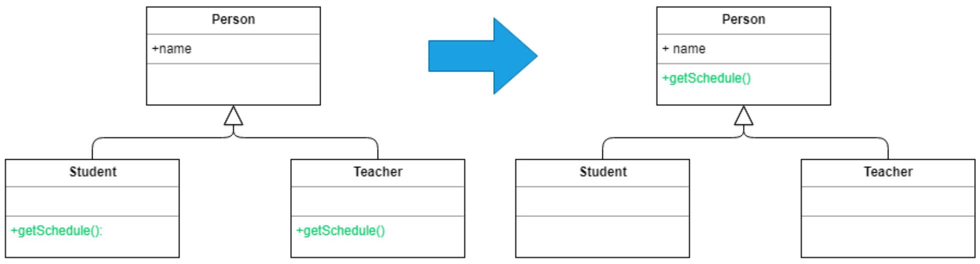


Figure 3.4: Pull up method

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.

3.2.7 Encapsulate field

This refactoring is used to change a public field to a private one and provide accessors to it.
Encapsulation, the ability to conceal object data, is one of the pillars of object-oriented programming. Without encapsulation, all objects would be able to get and modify the data of other objects without the owner's knowledge. This reduces the modularity of the program. When the data and behavior are clustered together, it is easier to modify the code - the changed code is in one place.

```
public String name;
```

Encapsulating the field `name` will result to the following code:

```java
private String name;
public String getName(){
    return name;
}
public void setName(String name){
    this.name = name;
}
```

How to refactor:

1. Create getter and setter methods for the field
2. Find all objects that reference this field. Use the setter method instead of assigning a value to this field and getter when accessing the field's value.
3. Declare the field as private

Chapter 4

Implementation

4.1 GES platform

GES is a generative software development platform used to create generative solutions like application generation, software conversion, and refactoring. Its strength lies in providing a platform to quickly reverse engineer and generate code while minimizing the amount of code written for these transformations. An example of how this IDE looks like can be seen in figure B.1.

In the first part, I list some of the frequently used commands used to work with GES and then show how the GES platform could be used on a representative subset of refactorings with code examples and explanation.

4.1.1 Working with GES

The transformations we are working with are called processes (or 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 they are executed by processing the compilation objects.

4.1.1.1 Useful editor keys

In the following table 4.1 are listed useful keys for working with GES. These keys are primarily used to run and compile the transformation scripts. It is very beneficial for the development process to get familiar with them, especially the one for compiling and running the process and the one presenting the object's data to the user.

Key	**Description**
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 the process and executes

Table 4.1: Useful keys

4.1.1.2 Controlling the execution

The execution can be controlled by the following commands 4.2. They are used primarily for the creation of new processes, for example, pausing the execution to look into the contents of the data structures to ensure they contain the desired information or during the debugging process.

Name	**Description**
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

Table 4.2: Commands used to control execution

4.1.2 Manipulation with objects

One of the essential parts of the GES transformations 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.

GES provides two main ways to create a new object. The first one is defining a new object and the second one is projecting it from an existing one. The main keyword for creating new objects is `Define`. Define is used to create an entirely new object with predefined attributes and values. 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` keyword. 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 object will have the new attribute.

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

4.1.3 RES and GES projective technologies

The RES functions are used to reverse engineer data from a given code. When called, 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 `@` or `$` sign and can be accessed by the name specified in the template. The templates work similarly like regular expressions, but with more features. The RES projective technology is used by calling the function ERES with the right arguments.

```
ERES(configuration,'@ParseConfiguration','','N');
```

The first argument of the RES function is the source - the object we will extract the data from. 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, or if it is more complex, a separate file containing the template can be referenced, like in this case.
The next attribute is the view name to extract out of RES if the default one is not right.
The last attribute is the edit flag, if it is set to true, RES editor will be opened at run time.

```
[@@CodeBlock
[$Accessor:(private)$ $DataType:([\w<>]+)$ $AttributeName$
   $Rest:[^;{}]*;$]
]
```

Let's take a look at the example of a RES template (the one in the example is used to extract a private field from a code fragment). The template itself can contain several types of variables (tokens). The first type is defined by the `@` sign. It is used to capture whole code blocks and the compiler will calculate meta information of this block, which could be used later for locating this certain code block in the code snippet (useful for editing it later).

The next type of variables is defined by the `$` sign. The name of the token can be followed by a colon sign and a regular expression that defines how the input text should be matched.

```
id~~~location:length:rel_location
example: I10000~~~292:39:292~~~0
```

Figure 4.1: The RES meta format

Using the `_meta` suffix with variable name captures the location of the code fragment extracted by RES in the original source code. This location is stored in the RES meta format. The RES meta format is in the following structure 4.1.

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

4.1.4 Interactive modes of PTs

Codiscent's studio provides another important tool for writing processes for code transformations which is called interactive mode for GES and RES. This feature must be invoked manually from the code when calling specific transformation. It will open a new window allowing to experiment with the chosen technology and providing great help with debugging the templates.

```
ERES(configuration,'@ParseConfiguration','','Y');
```

The code sample above shows how to call RES in the interactive mode, by choosing the value **Y** (value **N** will suppress the interactive mode) in the method call. When the compiler encounters the line with this code, a new window will be opened in the studio and the user will be able to debug the template. When the user is done with their work, the compiler will continue executing the code from the next line. In figure B.2 you can see how the interactive mode looks like,
In the first text area, there is a preview of the input text to which the template will be applied. In the text area below is the RES template. Both can be edited here and by clicking on the **RE** button, the studio will recompile the code and show the result in the table below.

Interactive GES mode could be invoked the same way as the RES one. The studio will open another new window B.3, which presents all the variables that are ready to be used in the template and two main text areas - the upper one for the template itself and the lower one for previewing the output. Clicking on the element will paste it into the template area, providing an easy way to construct the template.

4.2 Code samples

Before I was able to work on the refactorings, I needed to create code examples on which the refactorings will be performed. Depending on the type of refactoring, I wrote a code sample on which it will be easy to demonstrate the key aspects of each refactoring. For example, for the pull up members refactoring, I designed a structure of multiple subclasses that are using duplicate methods and fields. When the code samples were ready, the next step was writing tests - unit tests. As I mentioned before, writing tests is an important part of refactoring the code. For testing of my work, I chose to incorporate the jUnit framework because it is easy to work with and has good integration with current development environments.

4.3 Designing refactoring templates

One of the goals of the thesis is to define templates to configure the refactorings. I designed simple templates to configure the refactoring, so the user doesn't need to edit the code in the GES platform IDE. These templates help to reuse the implemented refactorings because it is easier and faster to edit the configuration template. The second benefit is that the user doesn't need to know the Codiscent's language, they only need to follow the instructions written in the description block. In the next sample 4.2 the general structure of the template could be seen.

```
/* This is a configuration template for ENCAPSULATE FIELD
*
*  It will generate getter and setter methods and make the
*  chosen field private
*
*  Insert /*encapsulate*/ comment before the declaration
*  of the field you want to encapsulate. Multiple fields
*  can be selected this way.
*
*  inputFile: specify the full path to the input file
*  outputFile: specify the full path to the output file
*/

inputFileName: "C:/Abosulte/Path/To/File.java"
outputFileName: "C:/Abosulte/Path/To/File.out.java"
```

Figure 4.2: Configuration Template for Encapsulate Field

The structure is always in the following format. It starts with the description block, which consists of **name of the refactoring**, followed by a description of the **expected result**, **special instuctions** - if there are any - in this case, it is the identification of the fields that will be encapsulated, and **format of the input fields**. The description block is ended by the **configuration fields** themselves.

The next step was to determine how to parse these configuration templates. Fortunately, because the template is structured, the RES technology is of great use here. All I needed to do was to create a RES template that would fit the configuration file 4.3. Because the configuration templates for each refactoring contains different fields, I needed to create a parsing template for each of them.

```
$Commentary:\/\*.+\*\/$
inputFileName: "$inputFileName:.+$"
outputFileName: "$outputFileName:.+$"
```

Figure 4.3: RES template for parsing of configuration

RES technology tries to match the template to the input object. The template is a combination of variables and plain text. The variables are enclosed in the dollar signs and they are the ones that will be extracted from the configuration file. You can notice that those variable names are followed by a colon and a regular expression. These tell us how to match the variables from the source file. If there is no regular expression, RES will try to match a simple string ending with whitespace.

With the source code samples and configuration files prepared, I could start working on implementing the refactoring transformations themselves.

4.4 Refactorings

In this section, I will describe a representative subset of refactorings implemented alongside the implementation process, which I believe are the most suitable ones to show how the projective technologies could be used for code transformations.

4.4.1 Extract method

The first refactoring explained is an extraction of a method from an expression. Because extracting a method belongs to the most commonly used transformations, it was my first choice of implementation. Unfortunately, as we will see from the implementation process, this is not an ideal one for the usage with projective technologies.

Firstly, we need to parse the input data from the configuration file and read the file containing the source code

```
/*Parse configuration template*/
Define CONFIG(configLocation)=[src-data/extract-method/
   configuration-template.txt];
Extend CONFIG configuration=ReadFile(configLocation);
Extend CONFIG [inputFileName,outputFileName]=
  ERES(configuration,'@ParseConfigurationEMRes','','N');
Project X from CONFIG extend content=
  ReadFile(inputFileName);
```

Firstly, we create a new object CONFIG with one field containing the location of the configuration template. Notice that the path to the file is unquoted and in the Unix format. The ReadFile method will then expand the path to an absolute one and read the contents of the file into the variable `configuration`. Then we make a call to the RES technology and parse the configuration file. The content of the source file is read in the same fashion.

With the source data ready, we can start the code transformation process. Again we use the reverse engineering technology to identify the relevant data that will be extracted, in this case, the relevant code block is identified with the token `/*extract*/`. Also, you can notice that in this case, we use an inline RES template instead of an external one, which was used to parse the configuration.

The variable `@ExpressionBlock` which starts with the `@` sign allows us to capture the whole code block matched with the template. When this syntax is used, RES will implicitly create a new variable `@ExpressionBlock_meta`, that could be used to locate the extracted code fragment in the input object.

```
Project Y from X extend
  [Type, Element,Expression,@ExpressionBlock_meta]=
  ERES(content,'[@ExpressionBlock /*extract*/$Type$
   $Element$ = $Expression:.+?$ ;] ','','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, which is called to further parse the expression we extracted earlier.

In the next step, we create the function body. We process the result of the 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. Finally, we can construct the function definition with GES using the previous variables and the function call.

```
/* Make a function call body  */
Extend Y ElementCapitalized=CapitalInitial(Element)^
[ExpressionVariable]=
    ERES(Expression,'$ExpressionVariable$','','N')^
FunctionBody=
    GES('@ExtractMethodGesFunctionBody','','N')^
FunctionCall=
    GES('$Type$ $Element$=get$ElementCapitalized$
    ([$ExpressionVariable$ ^,]);','','N')
present [-ExpressionVariable];
```

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

```
/* Call the newly created function */
Extend Y xcnt=RSBI(cnt,@ExpressionBlock_meta,
   FunctionCall)[];
```

The RSBI function(replace set by index location) replaces the original code block with the new one containing the call to the created function. Its arguments are - text to change, starting position - this could be an index or the RES meta format(here we are using the variable `@ExpressionBlock_meta`) - 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(updatedContent,
  'public class $ClassHeader$ {[ $ClassBody:.+$ ]}',
  '','N') ^
Output = GES('@ExtractMethodGesOutput','','Y') ^
outputFile = 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.

4.4.1.1 Extract method summary

The biggest benefit of the reverse engineering technology is that it is very general and the usage could be tailored to solve many specific problems, independent of the language of the source code. But this is also its biggest drawback in this refactoring. To create the function signature in this refactoring we need to know the correct data types of the variables that are found in the extracted expression. In this case, we can safely assume that the data

types are the same as the type of the result, but it always doesn't have to be that easy.
To correctly identify the types of the variables and whether the variables are local or not we would need something stronger than a reverse engineering template, for example, parser of the used language. This is the biggest advantage for refactoring purposes that specific development environments have over using projective technologies, especially for this type of refactoring where the context of the code is very important.

4.4.2 Pull up members

The following example recognizes common fields and functions from child classes and moves them to a parent class. As we will see from this example, the projective technologies are much better suited to perform this kind of refactoring.

We start the transformation in the same fashion as the previous example by parsing the configuration file. In this case, it is a little different, the child classes are defined in a field named subClass, which we need to process as a list. The processing is done using RES technology, which will extract the name of the classes and save it to a variable as a type List. In the next step, we create the path to those subclasses. The compiler correctly identifies our variable `fileList` as a list and iterates over its values to create the desired paths to the files, which contents are then read.

```
/*Read files using list*/
Extend A [fileList]=
  ERES(subClass,'[@ListBlock $fileList$; ^]','','N');
Project B from A extend files=
  inputFolder+'/'+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. The `present` directive tells the compiler which of the attributes we want in the projected object, or we want to omit (we use a minus sign to exclude them).

```
/*Extract elements from the sub classes*/
Project D from B 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<>'';
```

The RES method call references an external template. This template serves to parse the elements from the subclasses, so we can decide if they need to be pulled up or not. The elements can be of two types, either methods or fields. The part of the template to capture the methods is different - we need to extract their parameter sets and bodies too.

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

Then we find the common elements to be pulled up. We create two new objects, `D1` containing the unique names of all the elements from the child classes, and `D2` containing names of all the common fields and methods. This is done by joining the `D1` object with itself defined by the mapping rules. Because `D1` contains only two columns - the name of the element itself and the name of the class, where it is declared, this results in filtering the elements that are common.

```
/*filter common elements*/
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];
```

Once the common elements are identified, the next step is to remove them from the sub-classes by replacing each occurrence of the corresponding code block with an empty string. When this is finished, the updated classes are ready to be saved to a file.

```
Project D3 from a<D> join b<D2>
  link a.Element=b.Element map *:a.*;
/* Update the body of sub classes */
Extend D3 updatedContent=
  RSBI(content,@CodeBlock_meta,'')[fileList]
  present [fileList,updatedContent];
/* Write the updated files */
Project B1 from B present[outputFolder];
Project O1 from D3 cross B1;
Extend O1 outputFileName=
  outputFolder+'/'+fileList+'.out.java'^
  o = WriteFile(outputFileName,updatedContent);
```

When the sub-classes are updated we start working on changing the superclass. We need to add the extracted common fields and methods to the parent class. First, we parse the class headers and body using RES into the new object E.

```
/* Add the moved code to the superclass */
Extend B parentPath=inputFolder+'/'+parentClass+'.java';
Extend B parentContent=ReadFile(parentPath);
Project E from B extend[Imports,ClassHeader,ClassBody]=
  ERES(parentContent,
  '$Imports:.*$
  public class $ClassHeader$ {
    [ $ClassBody:.+$ ]
  }','','Y');
```

New object D4 is prepared for the purpose of generating the code block with the extracted elements.
But before the code could be generated, there needs to be performed one last important step. The extracted elements which were private in the subclasses could no longer be private and needs to be protected now. This is done with the `ReplaceByPattern` function. Afterward, the code block could be generated. We then generate the result class in the same way as in the Extract method example using the GES technology.

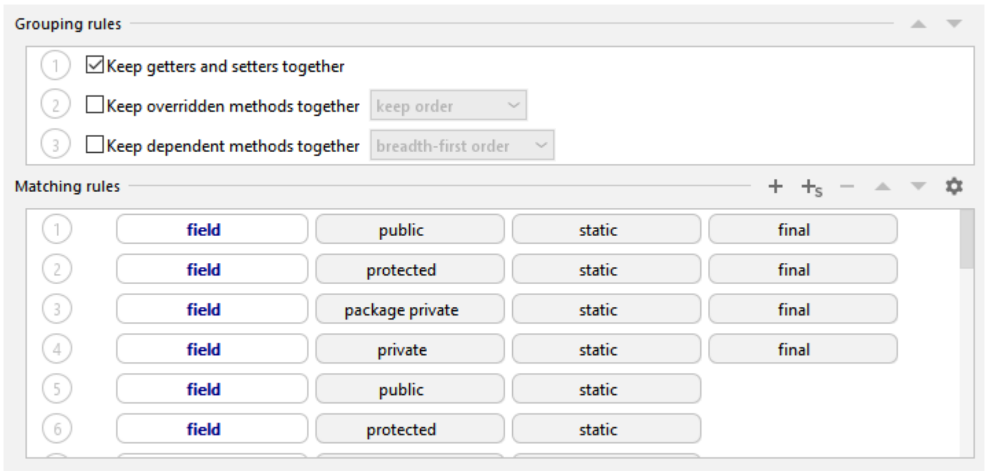
```
Project D4 from a<D> join b<D2>
  link a.Element=b.Element map *:a.*
  present [Element,CodeBlock];
Extend D4 CodeBlock=
  ReplaceByPattern(CodeBlock,P'private','protected');
Extend D4 FunctionBody=
  GES('$CodeBlock$','','N')
  present [FunctionBody];
```

4.4.2.1 Summary of pull up elements

As can be seen from the implementation process of this refactoring, the use of the projective technologies is much more beneficial here. This kind of transformation is ready to be automated and doesn't suffer from the same problems as the Extract Method. Another benefit is that it doesn't require much input from the user, all they need to do is specify the parent and child classes in the configuration template.

4.5 Rearanging the code

Predefined class structure helps us reaching what we are looking for more quickly. Because there is a consistent pattern, we know where to look for the 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 structured. On the picture 4.4 below you can see the default rules. The rules are applied to our code when the formatting is run.



Grouping rules

1. ☑ Keep getters and setters together
2. ☐ Keep overridden methods together — keep order
3. ☐ Keep dependent methods together — breadth-first order

Matching rules

1	field	public	static	final
2	field	protected	static	final
3	field	package private	static	final
4	field	private	static	final
5	field	public	static	
6	field	protected	static	

Figure 4.4: Definition of rearranging rules in IntelliJ IDEA

The rearranging of code could also be done using the projective technologies. I created a script that will find all class fields in a class and then regroup them above method declarations, protected fields first, private fields second, and public fields last. Although the IDE version offers more customization, I think this script is still very useful, because it could be easily automated and doesn't need much input from the user, except specifying the source file.

The user needs to specify an input and an output file in the configuration template and then the source class will be transformed.

Similarly as in the Pull up members refactoring, we extract the member fields using the RES technology. The RES template is the following:

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

The CodeBlock annotations define that the whole block will be saved to a

variable, with meta information like the location of the code block. Then follows the description of the element we are looking to extract.
When the elements are ready, all we need to do is to remove them from the class body, group them together, and then generate a new class file. The generation of the file is defined in the following GES template:

```
$*Imports$
public $ClassHeader$ {
    $*protectedBlock$
    $*privateBlock$
    $*publicBlock$
    $updatedClassBody$
}
```

In this example, the names of the respective code blocks are preceded by the asterisk sign. It serves to tell the GES compiler that these elements are optional and may be null. If there are any unhandled null elements in the template, the generation of the code will result in an empty string. The grouping of the elements can be easily changed by reordering them in the template.
Unfortunately, the output of this script is not formatted at all. I will address solving this issue in the next section.

4.6 Formatting the code

After the GES generates the desired source code, the result is rarely formatted. Although the formatting of the result could be solved with clever placement of whitespaces in the GES template when developing the scripts, it is not a good thing to do, and in some cases, it is impossible to do it right. It makes the templates unreadable and is very time consuming because the whitespaces need to be edited with trial end error.
This is a major drawback. The goal of code refactoring is to make code cleaner and more **readable**. As you can see from the output files of the refactorings, it takes much more time to read than when the code we look at is formatted. Fortunately, we could solve this issue far more easily and elegantly than editing whitespaces in the templates - with the use of code formatting programs, although we need to use another third party software.

The formatting of the code could be done manually, and although if could be quite fast when the target code is small enough, it defeats the purpose of the automated refactoring - allowing us to introduce new errors to the code.

There are a few ways it can be done. The first one is using an online tool like **Code Beautify** [13]. This is useful when we are working with small projects when copy-pasting the code into the browser is not a problem, but it is not optimal. Another solution is to use **plugin** for our favorite text editor or there is a possibility that the IDE supports it by itself. Then it could be configured to run the code formatter with every saving of the file or build of the application, which is very convenient. This practice helps us ensure the code formatting is the same across our application, it is faster when more than a few lines of code need to be changed and could be done just by pressing a keyboard shortcut.

4.6.1 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 uploading 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.[14]

Another option, if we don't want to run fully equipped IDE, is to use text editor with some sort of plugin to format the code. My personal choice of the text editor is *Sublime Text 3* and I had great success using the plugin *Formatter* [15], which is fully customizable and supports multiple programming languages. In this case, the reformatting is done when the source file is opened and could be configured to run on shortcuts too.

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 representative subset of refactorings. As can be seen in these samples, some of the refactorings are more suited to be used with the PTs than others. The ones that have the potential to be used in practice are the ones that didn't require frequent interaction with the user and aren't heavily dependent on the context of the code.

The greatest obstacle I was facing when working on this thesis is that the Codiscent's projective technologies are still under development and I was encountering bugs when working with them. Fortunately, I was able to report these bugs and Codiscent was happy to fix them or provide a workaround.

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(after all, this is one of the key reasons for using an IDE) than to take the time to write and test the refactoring scripts. Another point is, that the Java programming language is very widely used and is greatly supported in various development environments. Therefore it may be beneficial to use the projective technologies for refactoring in different languages with lesser IDE support.

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] Testing levels. Mar 2018, [Online; accessed 22-December-2019]. Available from: `http://softwaretestingfundamentals.com/software-testing-levels/`

[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`

[10] Codiscent Ltd. About Generative Software Engineering [online]. [Online; accessed 03-June-2020]. Available from: `http://codiscent.com/?page_id=296`

[11] 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.

[12] 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`

[13] Code Beautify. `http://https://codebeautify.org/`, accessed: 2010-06-01.

[14] Reformat and rearrange code. [Online; accessed 08-January-2019]. Available from: `https://www.jetbrains.com/help/idea/reformat-and-rearrange-code.html`

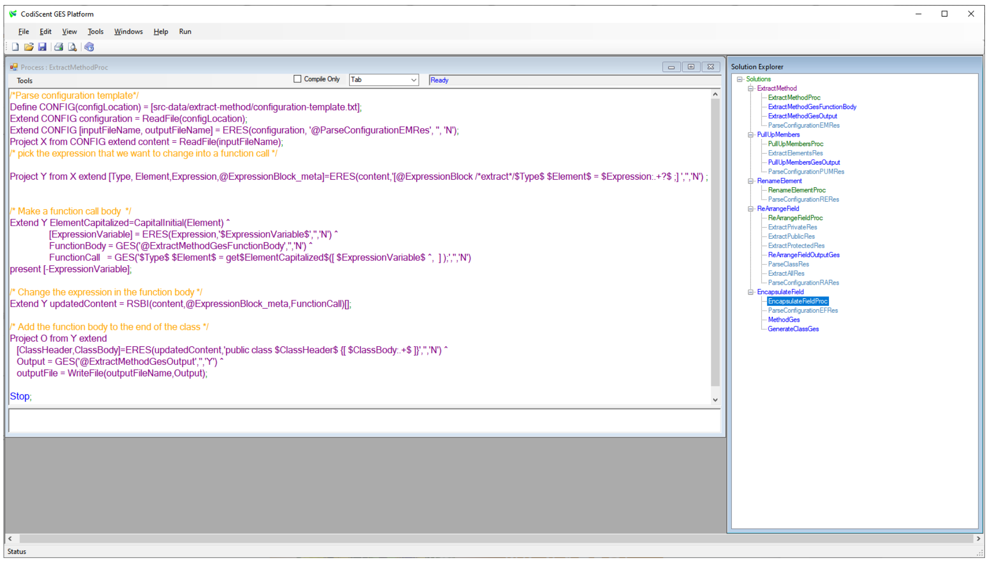
[15] Sublime Text 3, Formatter plugin. `https://packagecontrol.io/packages/Formatter`, accessed: 2010-06-01.

Appendix A
Acronyms

GES	Generative Engineering studio
IDE	Integrated development environment
MDD	Model-Driven development
PTG	Projector Template Generator
PTs	Projective technologies
RES	Reverse Engineering studio

APPENDIX B

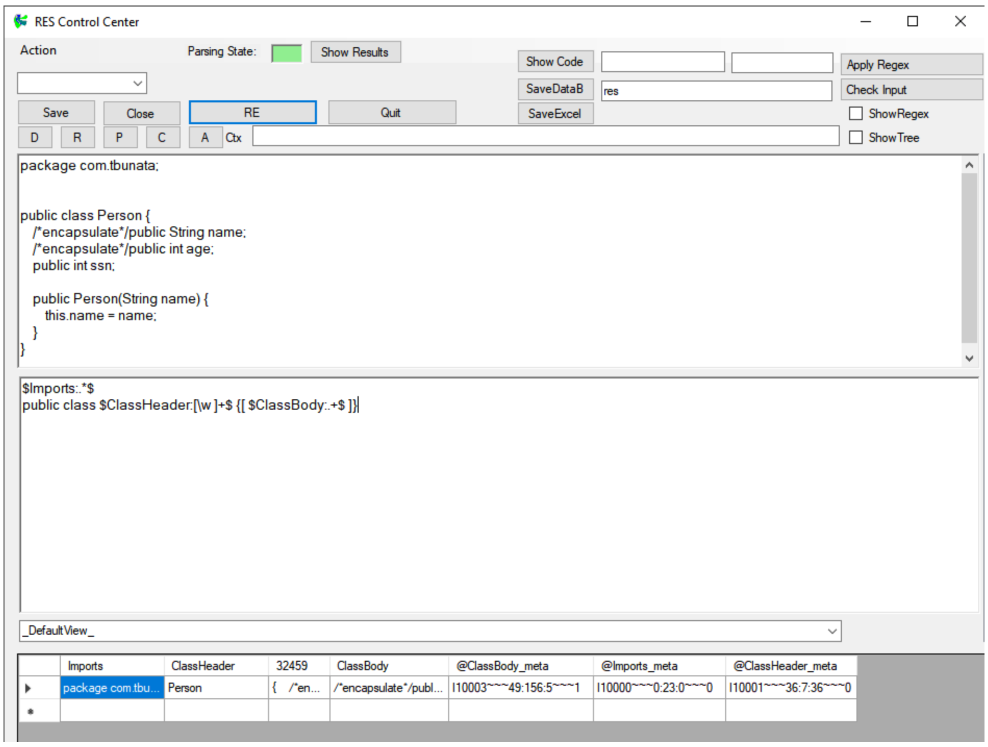
Screenshots of Codiscent's IDE



```
/*Parse configuration template*/
Define CONFIG(configLocation) = [src-data/extract-method/configuration-template.txt];
Extend CONFIG configuration = ReadFile(configLocation);
Extend CONFIG [inputFileName, outputFileName] = ERES(configuration, '@ParseConfigurationEMRes', '', 'N');
Project X from CONFIG extend content = ReadFile(inputFileName);
/* pick the expression that we want to change into a function call */

Project Y from X extend [Type, Element,Expression,@ExpressionBlock_meta]=ERES(content,'[@ExpressionBlock /*extract*/$Type$ $Element$ = $Expression:.+?$ ;] ','','N') ;


/* Make a function call body  */
Extend Y ElementCapitalized=CapitalInitial(Element) ^
            [ExpressionVariable] = ERES(Expression,'$ExpressionVariable$','','N') ^
            FunctionBody = GES('@ExtractMethodGesFunctionBody','','N') ^
            FunctionCall   = GES('$Type$ $Element$ = get$ElementCapitalized$([ $ExpressionVariable$ ^,  ] );','','N')
present [-ExpressionVariable];

/* Change the expression in the function body */
Extend Y updatedContent = RSBI(content,@ExpressionBlock_meta,FunctionCall)[];

/* Add the function body to the end of the class */
Project O from Y extend
  [ClassHeader,ClassBody]=ERES(updatedContent,'public class $ClassHeader$ {[ $ClassBody:.+$ ]}','','N') ^
  Output = GES('@ExtractMethodGesOutput','','Y') ^
  outputFile = WriteFile(outputFileName,Output);

Stop;
```

```
Solution Explorer
⊟ Solutions
  ⊟ ExtractMethod
      ExtractMethodProc
      ExtractMethodGesFunctionBody
      ExtractMethodGesOutput
      ParseConfigurationEMRes
  ⊟ PullUpMembers
      PullUpMembersProc
      ExtractElementsRes
      PullUpMembersGesOutput
      ParseConfigurationPUMRes
  ⊟ RenameElement
      RenameElementProc
      ParseConfigurationRERes
  ⊟ ReArrangeField
      ReArrangeFieldProc
      ExtractPrivateRes
      ExtractPublicRes
      ExtractProtectedRes
      ReArrangeFieldOutputGes
      ParseClassRes
      ExtractAllRes
      ParseConfigurationRARes
  ⊟ EncapsulateField
      EncapsulateFieldProc
      ParseConfigurationEFRes
      MethodGes
      GenerateClassGes
```

Figure B.1: Look at Codiscent's IDE



res

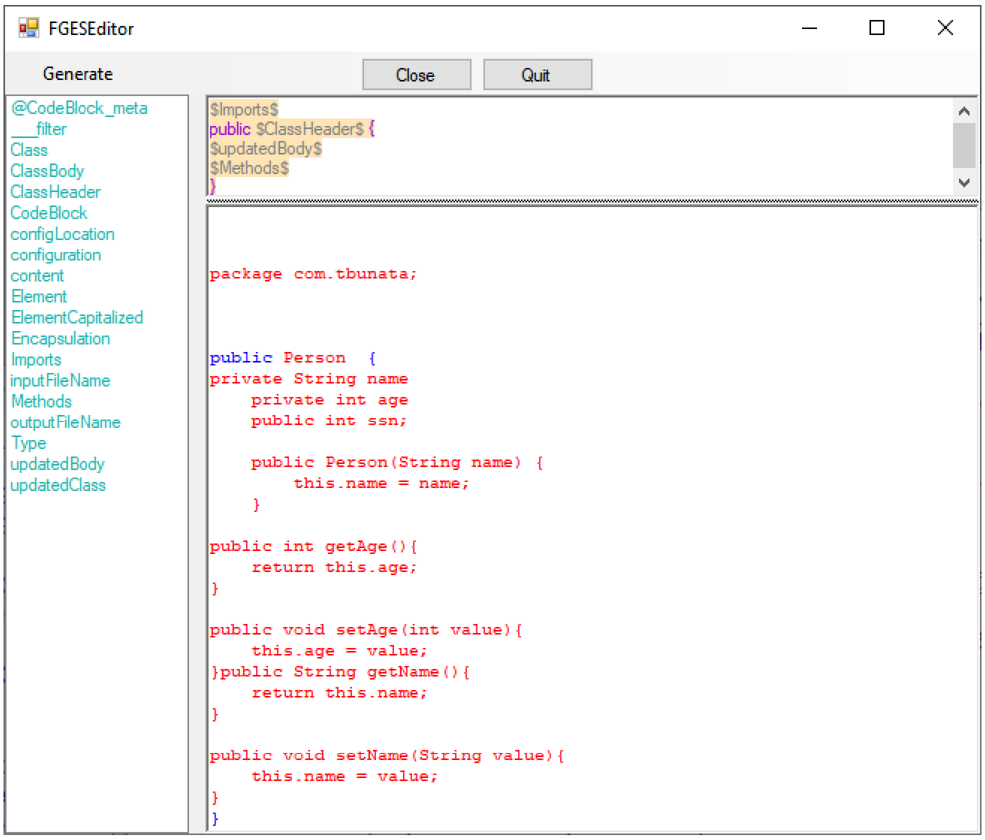
```
package com.tbunata;


public class Person {
   /*encapsulate*/public String name;
   /*encapsulate*/public int age;
   public int ssn;

   public Person(String name) {
      this.name = name;
   }
}
```

```
$Imports:.*$
public class $ClassHeader:[\w ]+$ {[ $ClassBody:.+$ ]}
```

DefaultView

	Imports	ClassHeader	32459	ClassBody	@ClassBody_meta	@Imports_meta	@ClassHeader_meta
▶	package com.tbu...	Person	{ /*en...	/*encapsulate*/publ...	I10003~~~49:156:5~~~1	I10000~~~0:23:0~~~0	I10001~~~36:7:36~~~0
*							

Figure B.2: RES in interactive mode



@CodeBlock_meta
___filter
Class
ClassBody
ClassHeader
CodeBlock
configLocation
configuration
content
Element
ElementCapitalized
Encapsulation
Imports
inputFileName
Methods
outputFileName
Type
updatedBody
updatedClass

```
$Imports$
public $ClassHeader$ {
$updatedBody$
$Methods$
}
```

```


package com.tbunata;



public Person  {
private String name
    private int age
    public int ssn;

    public Person(String name) {
        this.name = name;
    }

public int getAge(){
    return this.age;
}

public void setAge(int value){
    this.age = value;
}public String getName(){
    return this.name;
}

public void setName(String value){
    this.name = value;
}
}
```

Figure B.3: GES in interactive mode

Appendix **C**

Contents of enclosed CD

```
 readme.txt ........ the file with CD contents and installation description
 src ........................................ the directory of source codes
   thesis .............. the directory of LaTeX source codes of the thesis
   ReleaseCode ...................... implementation and GES platform
     src-data ............. refactoring configuration, input and output
     FileSet ....................................... refactoring scripts
     ObjectSet.xml ...................... GES settings and templates
     MetaObjectSet.xml .................................. GES data
     Cache ................................... vital GES platform data
     WorkFlowOrchestrator.exe .......... executable of GES platform
     OrchestratorDocumentation.doc ......... documentation of GES
 text ........................................ the thesis text directory
   thesis.pdf .......................... the thesis text in PDF format
```

Appendix D

Installation guide

D.1 List of refactorings

The attached CD contains the following refactoring examples:

- Encapsulate field
- Extract method
- Pull up members
- Rearrange fields
- Rename element

D.2 Requirements

- Windows OS 7 or later

D.3 Performing the refactoring

1. Start the GES platform by executing the file:
   `src/ReleaseCode/WorkFlowOrchestrator.exe`
2. The Solution Explorer (right column) contains an overview of available refactorings.
3. Choose the desired refactoring in the Solution Explorer and double click on its RefactoringNameProc file (colored in green)
4. The refactorings could be customized by editing the configuration template - `src/ReleaseCode/src-data/refactoring-name/configuration-template.txt`

5. Change the source and output file paths in the configuration template - the absolute path is needed
6. Return to the refactoring's process in the GES platform and click on **Run** → **Recompile and Run** or hit **F11** to execute the refactoring.
7. The result will be written to the output file specified in the configuration file.