Utilization of Code Clones in Productization Process to Detect Customer-Specific Differences

Master’s Thesis
Department of Built Environment
School of Engineering
Aalto University

Espoo, 16 October 2017

Bachelor of Science in Technology Erika Salmivaara

Supervisor: Professor Lauri Malmi
Advisor: M.Sc. Mikko Ristikangas

Aalto University
School of Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi

Abstract of master's thesis

Author Erika Salmivaara

Title of thesis Utilization of Code Clones in Productization Process to Detect Customer-Specific Differences

Degree programme Degree Programme in Geomatics

Minor Software Technology **Code** T3001

Thesis supervisor Professor Lauri Malmi

Thesis advisor(s) M.Sc. Mikko Ristikangas

Date 16.10.2017 **Number of pages** 79 **Language** English

Abstract

The topic for this thesis was inspired by two case studies. The case studies are applications that are conceptually but not technically products. Their code bases contain customer-specific branches. The development strategy with the case studies has been forking an existing branch and customizing it to the needs of the new client. Code reuse and forking can be an efficient or even a necessary development strategy due to time pressure. However, code duplication may result in harder maintainability of the code base which in turn increases the maintenance costs.

Finding similar code fragments is researched in the field of code clone detection. Code clones are code fragments that are either the same or similar. The similarity can be categorized into 4 types. Type I clones are exact matches that differ only in layout, whitespace or comments. In addition to type I changes, type II clones can differ in identifier names and types or literal values. Furthermore, type III clones can have statements added, deleted or modified within the code fragments under comparison. Type IV clones are functionally similar clones. There are different kinds of techniques and tools for both detecting and visualizing clones. Different techniques find different sets of clone types. Code clone visualizations present both the overview of the cloning situation, and the details in the source code level. The branches of the same product of the case studies can be considered as clones of each other. They are expected to remind type III clones. They essentially originate from the same code base, but each one has added, deleted and modified statements within the corresponding files between the other branches. Identifying these changes facilitate forming an overall picture of how much the branches truly differ.

The transformation process from development of customer-specific software to product software is called productization. In order to productize, the differences in the branches must be determined. Each customization needs to be considered in the productization process to avoid reducing the value of the product. We defined a process how to utilize code clone visualizations to explore differences between customer-specific branches. Conclusion of this thesis is that utilization of code clones clearly expedites the productization process. The visualizations aid to locate the differences much faster than manually. Code clone detection is applied to fade out the uninteresting differences between the branches. Hence, the method aids to navigate to the truly interesting customizations that require manual inspection. The method also provides a general view of the cloning situation, which eases the task of estimating the workload. The process is applicable in situations, where the diverged code bases are expected to remind each other structurally, yet contain so many changes that a manual comparison of the branches with file comparison tools would be too time-consuming.

Keywords code similarity, code clone, duplicated code, productization, code smells, code reuse, legacy software, technical debt, program comprehension

Aalto-yliopisto
Insinööritieteiden
korkeakoulu

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi

Diplomityön tiivistelmä

Tekijä Erika Salmivaara

Työn nimi Koodikloonien hyödyntäminen asiakaskohtaisten erojen havaitsemiseksi tuotteistus-prosessissa

Koulutusohjelma Geomatiikka

Sivuaine Ohjelmistotekniikka **Koodi** T3001

Työn valvoja Professori Lauri Malmi

Työn ohjaaja(t) DI Mikko Ristikangas

Päivämäärä 16.10.2017 **Sivumäärä** 79 **Kieli** englanti

Tiivistelmä

Motivaatio diplomityön tekemiselle syntyi kahden tapaustutkimuksen johdosta. Ne käsittelevät sovelluksia, jotka ovat käsitteellisellä tasolla tuotteita, mutta eivät teknisesti. Niiden lähdekoodit sisältävät asiakaskohtaisia haaroja. Kehitysstrategia sovellusten kohdalla on ollut haarauttaa koodipohja asiakaskohtaiseksi koodipohjaksi ja muokata se asiakastoiveiden mukaiseksi. Koodin uusiokäyttö voi olla tehokas tai jopa tarvittava kehitysstrategia aikataulupaineiden johdosta. Toisteinen koodi voi kuitenkin hankaloittaa sovellusten ylläpitoa ja täten nostaa ylläpitokustannuksia.

Samankaltaisten koodin osien etsimistä on tutkittu koodikloonien tutkimusalalla. Koodikloonit ovat koodin osia, jotka ovat joko samoja tai samankaltaisia. Samankaltaisuus voidaan luokitella neljään tyyppiin. Tyypin I kloonit eroavat vain ulkoasun, tyhjätilamerkkien tai kommenttien osalta. Tyypin II kloonit voivat erota myös muuttujien nimien tai tyyppien osalta tai literaalien arvoissa. Tyypin III klooneissa voi olla lisättyjä, poistettuja tai muuttuneita lauseita välissä. Tyypin IV kloonit ovat toiminnaltaan samankaltaisia. Koodikloonien tunnistamiseen ja visualisointiin on erilaisia menetelmiä. Eri tekniikat löytävät eri tyyppisiä klooneja. Koodiklooneista voidaan visualisoida sekä kokonaiskuva kloonaustilanteesta että yksityiskohdat lähdekooditasolla. Saman tuotteen haarat tapaustutkimuksissamme voidaan ajatella olevan tyypin III klooneja toisistaan. Ne periytyvät alun perin samasta koodipohjasta, mutta jokaisessa on lisättyjä, poistettuja ja muutettuja lauseita toisiaan vastaavien tiedostojen välillä. Nämä muutokset halutaan havaita, jotta voimme saada kokonaiskuvan siitä, kuinka paljon haarat todellisuudessa eroavat toisistaan.

Tutkimuksen kohteena oli tuotteistusprosessi, jossa asiakaskohtaisesti räätälöidyt koodipohjat pyrittiin muuntamaan yhdeksi tuotteeksi. Tavoitteena oli selvittää kaikkien koodipohjien asiakaskohtaisesti räätälöidyt osat, jotta ne tulisivat huomioitua tuotteistusprosessissa. Jokainen räätälöinti voi olla tuotteen arvoa nostava tekijä. Kehitimme prosessin, jonka mukaisesti kloonien visualisointeja voidaan käyttää tuotteistusprosessissa. Tutkimuksessa havaittiin, että koodikloonien hyödyntäminen nopeutti selkeästi tutkimuskohteiden tuotteistusprosessia. Visualiointien avulla erot löydetään huomattavasti nopeammin kuin manuaalisesti. Kloonien tunnistusmenetelmiä käytetään tässä yhteydessä häivyttämään koodipohjasta epäkiinnostavat erot. Täten menetelmä ohjaa niiden erojen äärelle, joiden tarkastelu oikeasti vaatii manuaalista tulkintaa. Menetelmä antaa myös kokonaiskuvan tilanteesta, mikä helpottaa tuotteistamiseen tarvittavien työmääräarvioiden tekemistä. Menetelmä sopii tilanteisiin, jossa toisistaan erkaantuneet koodipohjat muistuttavat vielä rakenteeltaan toisiaan, mutta sisältävät niin paljon muutoksia, että käsin tehtävä koodihaarojen vertailu tiedostojen vertailuun tarkoitetulla työkalulla olisi liian aikaa vievää.

Avainsanat koodin samankaltaisuus, koodiklooni, toisteinen koodi, tuotteistus, koodin hajut, koodin uusiokäyttö, perinneohjelmisto, tekninen velka, ohjelman ymmärtäminen

Acknowledgements

Firstly, I would like to thank Professor Lauri Malmi for not only supervising this thesis but also his support and guidance throughout this process. I am also grateful for my advisor Mikko Ristikangas for being supportive and providing useful comments.

Most of all, I would like to state that my friends are the best. Thank you for everything.

Espoo, October 16, 2017

Erika Salmivaara

Contents

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Addressing the Problem with Code Clones 4
1.3 Focus of the Thesis 6
1.3.1 Purpose Statement 6
1.3.2 Structure of the Thesis 6

2 Code Clones **8**
2.1 Definition 8
2.1.1 What Is a Code Clone 8
2.1.2 Origins of Code Clones 11
2.1.3 Harmfulness of Code Clones 13
2.2 Clone Types 14
2.2.1 Type I Clones 15
2.2.2 Type II Clones 15
2.2.3 Type III Clones 16
2.2.4 Type IV Clones 17
2.3 Clone Detection 18
2.3.1 Clone Detection Process 18
2.3.2 Clone Detection Techniques 21
2.4 Clone Visualization 24
2.4.1 Getting an Overview of the Cloning Situation 24
2.4.2 Viewing the Differences in the Source Code 29

3 Used Techniques **30**
3.1 Tool Evaluation 30
3.2 Chosen Tools 35

4 Analysis **37**
4.1 Introduction 37
4.2 Preparation of the Source Code 39

4.3 Exploration of the Code Clone Visualizations 39
4.3.1 Interpretation of the Scatter Plot 40
4.3.2 Getting an Overview with the Scatter Plot 46
4.3.3 The Impact of Altering the Threshold Values 56
4.4 Results 64
4.4.1 Processing of the Found Customizations 64
4.4.2 Evaluation 66
4.4.3 Proposition of the Process 67

5 Conclusions **70**

Chapter 1

Introduction

1.1 Background and Motivation

The motivation for this thesis emerged from two case studies. Both of the case studies are applications that can be conceptually considered as products. However, neither of them is technically a product. The development strategy of the both systems has been forking an existing source code branch to a new customer-specific branch. The customizations made to branches over time have not been systematically documented. Thus, we do not have a general impression of how the branches differ.

The first case study is a larger code base, which has spread to 10 different branches. It has approximately 20K lines of code (LOC)[1] per branch. The other case study is smaller. It has spread to four branches, each having approximately 3K LOC.

Code reuse can be an efficient way to produce reliable code (Kim et al., 2004). Existing functionalities have most probably been tested before. Thus, time is saved in both implementation and testing of the software. Furthermore, forking the code base is justified, when the original and the new code base are expected to differ significantly (Kapser and Godfrey, 2006). However, if the code bases are not diverging, one ends up maintaining two similar code bases separately. This inevitably increases the maintenance costs, since bug fixes and improvements need to be implemented in two code bases instead of one. Logically, if there are four, or even ten code bases to maintain instead

[1]Number of lines in the code calculated with Visual Studio 2015 Code Metrics analysis tool. `https://msdn.microsoft.com/en-us/library/bb385914.aspx` referred 5.10.2017

of the single, the maintenance costs multiply accordingly.

Time-to-market pressure and high competition tend to result in application development without a formalized process (Di Lucca et al., 2002). This encourages programmers to reuse existing code. Furthermore, this results in code duplication. The code duplication may increase program complexity, which in addition to the lack of documentation might lead to hard maintainability of the code base.

Fowler et al. (1999) discuss *code smells*. Code smells are indicators of possible design issues in software. Code duplication is a recognized form of a code smell. However, sometimes code reuse is necessary due to time pressure of the project. Implementing abstractions that eliminate duplication is time consuming and often difficult. Especially, if the developer does not know the system well, implementing complicated abstractions is error-prone. Copying an existing, tested functionality and adapting it to the new environment is a secure way of not breaking the existing code. (Roy and Cordy, 2007)

Customizing the code base directly to the needs of the new client is an easy solution. Implementing the customization to the existing code base without breaking or removing any old features is a harder task. However, since it would result in a single maintainable code base, it would also be a better solution. Choosing the easier solution instead of the harder and better one results in taking *technical debt* (Yli-Huumo et al., 2017). The technical debt needs to be repaid in order to lower the maintenance costs.

Since the applications of the case studies are conceptually still products, their division to customer-specific branches has not been a cost-effective solution in the long-term. In order to gain control of the situation, we should aim towards the situation where we would have a single code base per product. Artz et al. (2010) discuss the process of transforming a customer-specific software to a product software, in other words, productization (Figure 1.1).

In order to productize, the differences between customer-specific branches should be detected. Since the customer-specific branches origin from the same original code base, they look similar. However, when one tries to compare the files with a file comparison tool, it may show that almost every line has changed, even though the contents of the files would appear to be similar. The traditional file comparison tools rather report possibly uninteresting

differences than miss a difference[2]. For example, the tools recognize changes in whitespace, variable names or layout even though the logic has not been changed. We want to generalize the common (standard) features, so that the focus may be shifted to the more significant customizations.

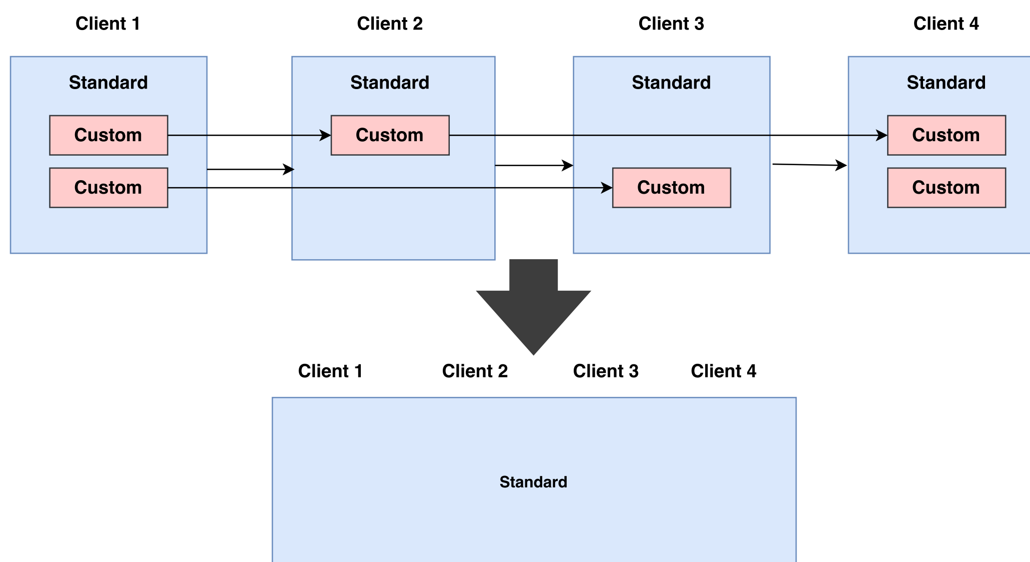

Figure 1.1: The objective of the productization process is to maintain a single code base for a product (below). The initial situation is presented above. Each client has own branch as a code base. However, they are essentially forked from the same code base. Hence, they include similar standard features. Nevertheless, the standard code base inevitably has slight modifications between branches, since the branches have been maintained separately for years. For example, the standard code base could vary in whitespace or variable names. Customizations are more significant changes. A customization may even be a feature that does not exist in every other branch. These are the changes that are found to be interesting in the productization perspective.

Another issue with file comparison tools, is that they do not provide an overall picture of the entire similarity situation. They may offer a directory comparison, which state which files are the same, similar or different between directories[3]. The number of differing lines between files does not really tell how similar the files are. Furthermore, comparing all files to each other line

[2] For example, Pretty Diff
`http://prettydiff.com/guide/unrelated_diff.xhtml` or

[3] WinMerge `http://winmerge.org/` referred 5.10.2017

by line is slow. All in all, manual comparison of files with a file comparison tool would be far too inefficient.

In summary, we need to find or define *a systematic way to productize software code that has diverged to customer-specific tailored branches.* Hence, research questions are:

1. Are there any tools that would aid comprehending undocumented legacy software?
2. Are there techniques to evaluate code similarity, when the code has been modified?

1.2 Addressing the Problem with Code Clones

In order to gain a general impression of the duplication situation of the branches, we would rather overlook some minor differences than find too many. The comparison of the branches could be done to code, which would have been normalized to some level. Finding similar code fragments that are not necessarily exactly the same, is researched in the field of *code clone detection.*

Code fragments that either consist of duplicated or similar code, are called code clones. Code clones can be textually or semantically similar or identical. There are four types of clones. The first three types are syntactically similar clone types. Type I clones are generally known as *exact matches.* Type II clones may have identifiers renamed, or types or literal values changed. Type III clones may have statements added, modified or deleted. Type IV clones are semantic clones. That is, they are functionally similar, but the textual representation might be completely different. (Roy and Cordy, 2007)

There are different kinds of techniques for detecting code clones. Different techniques find different sets of clone types. The techniques have limitations depending on how they examine the code. None of them finds all types of clones nor is suitable for every programming language. Thus, the chosen technique(s) strongly depend on the initial situation.

Clone detection tools usually report the source coordinates as the output of the detection, but they can also provide visualizations of the cloning situation (Roy et al., 2009). The detected clones can be processed after they are found. For example, they can be refactored away, or their visualizations

can be used for further examination. Code clones have been visualized with multiple techniques. The visualization techniques provide both overview of the cloning situation and source code views of the clones.

In our case studies, the branches of the same product can be considered as clones of each other. They are expected to remind type III clones, since they essentially origin from the same code base. However, each one has added, deleted and modified statements within the corresponding files between the other branches. Hence, the cloned parts can be considered to be the standard features between branches, and the type III differences the customizations in the code base.

Code clones have been used for software merging (Godfrey and Zou, 2005) and linked editing (Toomim et al., 2004) of duplicated code. Hence, we could try to merge the branches to one branch, or study whether the duplicated fragments of the branches could be modified simultaneously. However, we do not want to waste resources for maintaining the old code bases.

We want to have the product version of the software implemented with new technologies. Actually, the specifications and implementations of the new product versions of the case studies have already been started. Hence, defining the product base for the software is outside the scope of this thesis. Instead, we want to separate the customizations of each branch from the code base. Even though we want to use a single code base per product, we cannot just discard the custom features that have been implemented to different clients. Hence, we need to find out what customizations each code base contains, and make decisions whether to include them in the productized version, or is there a justified reason for discarding them.

Koleilat and Shaft (2007) explored clone visualizations as reverse engineering technique to locate frequently reused code fragments. Their research aimed to recognize reusable assets in the field of software product line analysis. Our research, on the other hand, aims to reduce reuse. We want to create a single product of the reused code base.

Code clones have been used in the field of program understanding by highlighting redundant code (Johnson, 1993). Similarly, highlighting the duplication between the branches of our case studies could make the customizations easier to perceive from the code base. Additionally, visualizing code duplication facilitates forming a mental model of the program (Rieger et al., 2004).

1.3 Focus of the Thesis

1.3.1 Purpose Statement

The purpose of this thesis is to determine, whether identifying code clones can be utilized to productize software from the technical perspective. We explore, what kind of information can be extracted from the code clone visualizations of the smaller application of the case studies. We conclude, whether the visualizations provide a general impression of the differences between branches, and can the customizations in branches be distinguished from them. Finally, we define a process, of how code clones can be utilized to explore the differences between branches of a code base that is conceptually a product. This process can be then used to explore the larger application of the case studies, or any other code base in similar situation.

We aim to expedite the process of productization by defining a method for overviewing and extracting the differences from the diverged branches. In order to productize software without reducing the value of the product, the customizations done to each client need to be considered in the process. In the long term, reducing the amount of duplication in the code base will lower the maintenance costs of the product.

1.3.2 Structure of the Thesis

The outline of this thesis is as follows. Chapter 2 is the theoretical background of the thesis, which is about code clones. First, the term code clone is defined, and their origins and harmfulness is discussed. Second, the different types of code clones are presented. Third, code clone detection process and techniques are explained. Fourth, the visualization techniques used in the analysis phase are represented. Other visualization techniques are discussed briefly.

Chapter 3 discusses about the tool evaluation. There are a lot of different tools for both detecting and visualizing code clones. We need to choose the most suitable tools for our code bases. The evaluation process is explained, and justifications for the chosen tools are presented.

Chapter 4 describes the performed analysis. The first section is a brief introduction to the analysis. The second explains the preprocessing stage. The third section explains how to interpret the overview gained with the scatter plot and the results of the explorative analysis. Different configuration val-

ues are tested in order to observe, how they impact the results. The fourth section presents the results of the analysis. First, there is a discussion of how the found customizations could be processed. Then, the results of the explorative analysis are evaluated. Last, the process of using code clones to detect differences between branches of a source code is presented as the outcome of this thesis.

Finally, chapter 5 summarizes the conclusions of the thesis. Suggestions for future work are also discussed.

Chapter 2

Code Clones

2.1 Definition

2.1.1 What Is a Code Clone

A sequence of lines of code is called a *code fragment* (CF). A code fragment can contain comments and whitespace characters. It is defined by triple

$$CF\ (\ file,\ begin,\ end\ ),$$

where *file* is the name of the file containing the code fragment, *begin* is the number of the starting line of the code fragment in the original code base, and *end* is the number of the ending line of the code fragment in the original code base. A code fragment can consist of any code block. For example, it can be formed of an entire definition of a function, or a sequence of statements. In other words, a code fragment is granularity free by definition. (Roy et al., 2009)

Code Fragment 1	Code Fragment 2
`1 int sum(int n) {`	`1 int sum(int n) {`
`2   int sum = 0;`	`2   int sum = 0;`
`3   for (int i=1; i<=n; i++) {`	`3   for (int i=1; i<=n; i++) {`
`4     sum = sum + i;`	`4     sum = sum + i;`
`5   }`	`5   }`
`6   return sum;`	`6   return sum;`
`7 }`	`7 }`

Table 2.1: Two code fragments that are clones of each other.

Code clones are code fragments that are either the same or similar to each other (Table 2.1). The relation is denoted by

$$f\ (\ CF_1\ ) = f\ (\ CF_2\ ),$$

where f is the similarity function, and CF_1 and CF_2 are code fragments (Roy et al., 2009). The relation of code clones is an equivalence relation (Kamiya et al., 2002). A code fragment is a clone to itself. In other words, reflexive relation holds. Naturally, clone detection tools (see section 2.3 on page 18) only report distinct clone fragments as clones. If CF_1 is a clone of CF_2, CF_2 is a clone of CF_1 also. Thus, the relation is symmetric. If CF_1 and CF_2 are clones of each other, they form a *clone pair* (Figure 2.1).

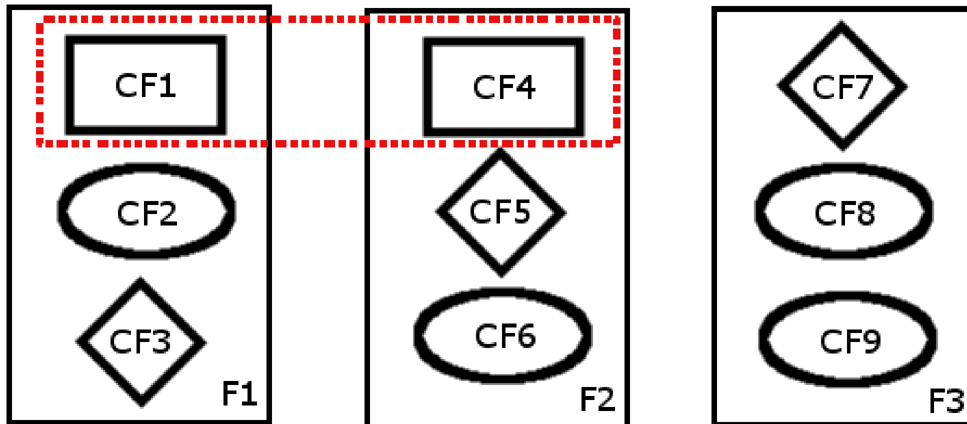

Figure 2.1: Code fragments CF1 and CF4, form a clone pair.

A group of code fragments that are clones with each other, form a *clone class* (Figure 2.2). Let CF_1 be a clone of CF_2, and CF_2 a clone of CF_3. If CF_1 is also a clone of CF_3, the code fragments $<CF_1, CF_2, CF_3>$ belong to the same clone class. In that case, transitive relation holds. However, similarity between code fragments can be measured in different levels. If the similarity between CF_1 and CF_3 would be below the defined threshold value, then CF_1, CF_2 and CF_3 would not form a clone class. In that case, their relation would not be transitive either. (Bellon et al., 2007)

If the same clone classes coexist between the same regions, the group of the clone classes is called a *clone class family* (Rieger et al., 2004) or *super clone* (Jiang et al., 2006). For example, if one clone class is defined CC_1 $<CF_1, CF_2, CF_3>$, and another CC_2 $<CF_4, CF_5, CF_6>$. For example, if

CF_1 and CF_4 are in the same file, CF_2 and CF_5 are with each other in another file, and CF_3 and CF_6 are yet in another file, the two clone classes CC_1 and CC_2 form a clone class family (Figure 2.3).

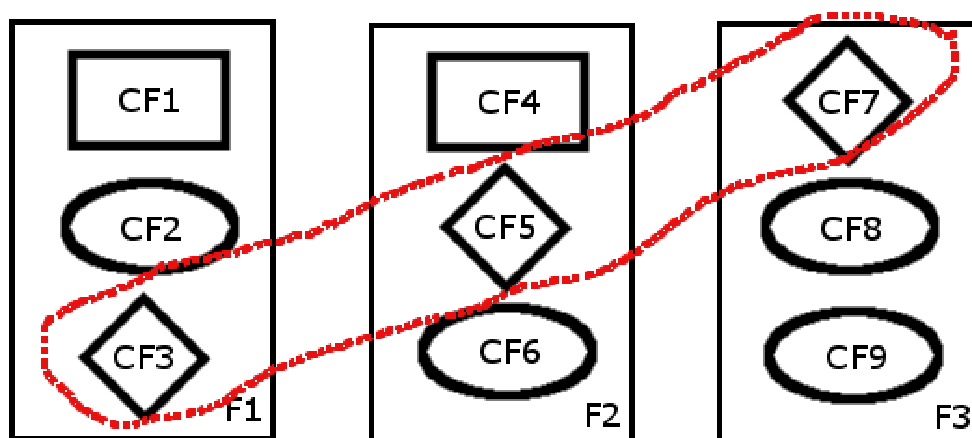

Figure 2.2: Code fragments CF3, CF5 and CF7, form a clone class.

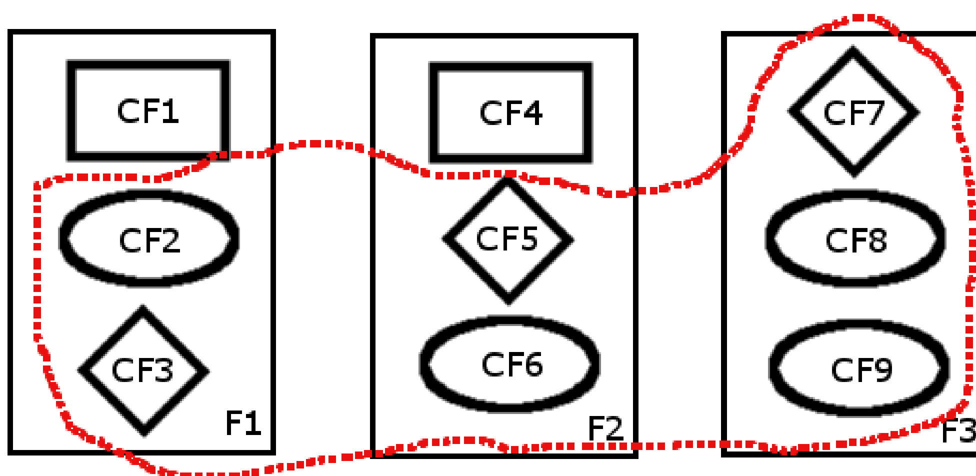

Figure 2.3: Code fragments CF2, CF3, CF5, CF6, CF7, CF8 and CF9, form a clone class family in the region including files F1, F2 and F3.

2.1.2 Origins of Code Clones

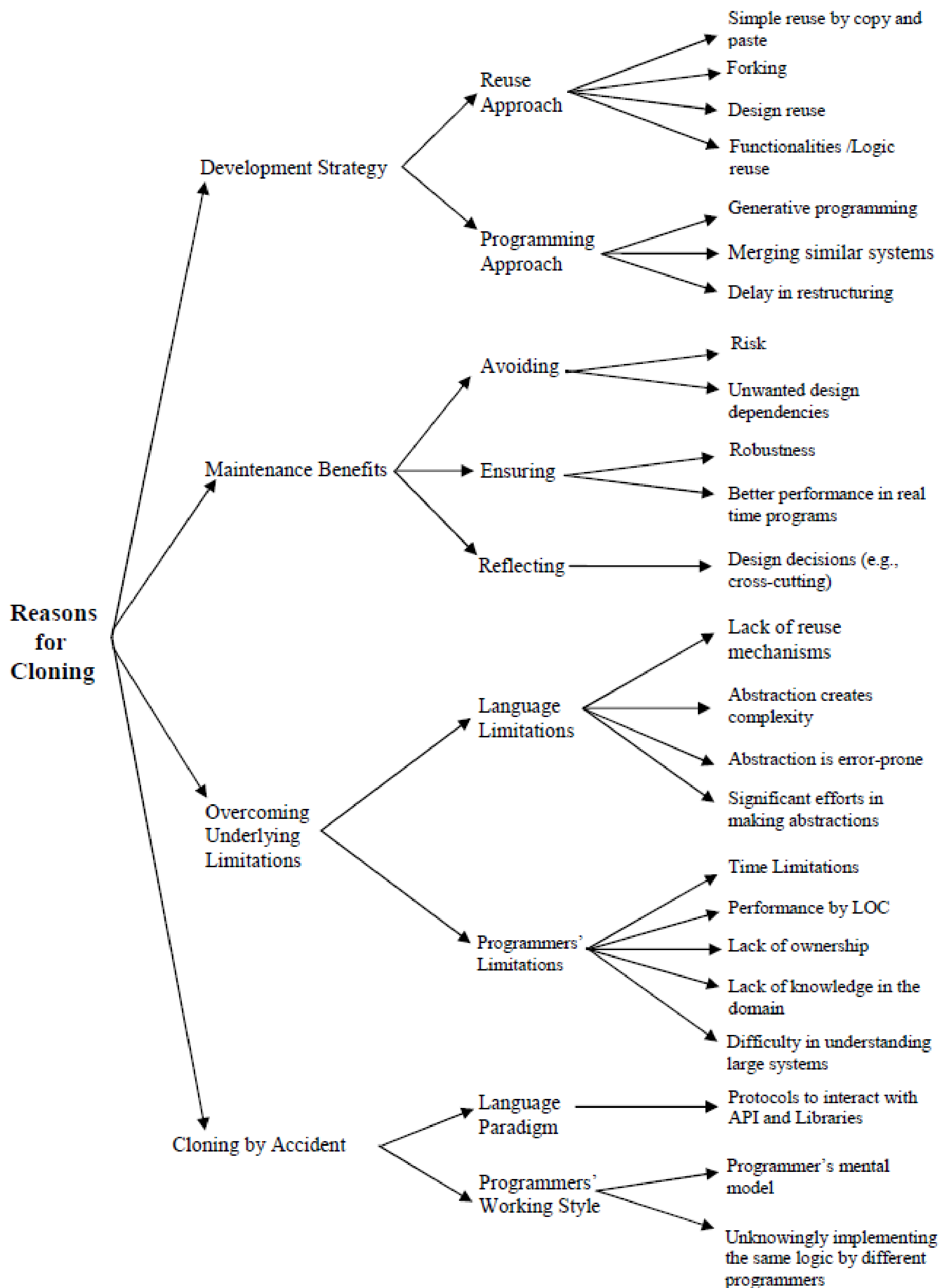

Figure 2.4: Reasons for cloning grouped by Roy and Cordy (2007).

There are several reasons for code duplication. Developers might be forced or tempted to clone code. Alternatively, clones can be formed by accident. Roy and Cordy (2007) have reviewed different factors causing code clones (Figure 2.4). This section summarizes their review.

The most obvious reason for code clones is reusing existing software. Code reuse strategies can be categorized into four groups. The first approach is code reuse by copying and pasting. Copied code fragment is still considered a code clone even if minor modifications have been made. Copy pasting existing code is a fast way to produce reliable functionalities, because the code most probably has been tested before. (Kim et al., 2004)

Second, forking is the act of reusing a large part, or an entire solution, as a base for a new solution. Forking is a justified action if the duplicated code bases are expected to diverge significantly during their evolution. (Kapser and Godfrey, 2006)

Third, design patterns can be reused. For example, a developer might recognize that he/she needs a similar structure that exists in another system. Gamma et al. (2002) have researched the reuse of design patterns in object-oriented systems. The fourth reuse strategy is logic reuse. Similar to design reuse, the developer might want to reuse some logic known to exist in a similar solution.

In addition to the reuse approach, another development strategy according to Roy and Cordy (2007) is programming approach. This can be divided to merging two similar systems, system development with generative programming approach and delay in restructuring. Merging of two systems may introduce code clones because there might exist similar functionality in both systems, which are being merged. Code generation often produces duplicates, because generating tools tend to have a template for the task at hand, and there most probably is repetition in the logic between the templates. Programmers may also delay the restructuring of their code, even though more coherent structure would reduce the amount of duplicate code in the system.

Furthermore, maintenance benefits may be gained by code cloning. A developer may be asked to reuse existing code, because if the existing code is well-tested, the risk of developing new code is higher than using the existing code. Sometimes the software architecture remains more understandable, if there is duplication. Abstractions needed for extracting code duplication

might be complicated (Kapser and Godfrey, 2006). If the cloned code fragments are isolated from each other, maintenance can be faster due to no need for considering effects of the changes to the clone.

Moreover, there might be limitations concerning the used programming language, the skills of the developer or the circumstantial factors. Programming language used might lack abstraction mechanisms. The effort of writing a general solution might be too significant compared with the option to maintain two cloned fragments. Furthermore, refactoring a piece of code to be more general increases the risk of introducing new bugs to the system. On the other hand, the challenge of writing more general solutions might be caused by the limitations of the programmer. If the programmer finds it difficult to understand the system or the problem at hand, he/she might be forced to copy paste similar existing solutions and modify them to his/her needs. Naturally, this is easier than generalizing existing solutions. The developer might also have time limits, which force to take the easier way out. Sometimes the productivity of a developer is measured by number of lines disregarding the quality of the code, which induces the developer to repeat rather than unify the code. Additionally, the reused code might not just be modifiable by the developer. Thus, the developer needs to copy and adapt it to his/her own system.

In addition to intentional code clones, they can be introduced by accident. Interacting with APIs or libraries usually happens with series of commands, and these commands might be repeated every time the same or a similar interface is used. Lack of interaction between the developers of the same system might lead to similar functionality programmed by different developers individually. On the other hand, the developer might repeat himself/herself, if he/she does not recall implementing such similar code earlier.

2.1.3 Harmfulness of Code Clones

As stated in previous section, the act of cloning is not always harmful. Forking and reusing code are fast ways to start with existing and tested code. For example, there may be a justified reason for the duplication, related to design clarity, stability or code ownership. However, it is still widely agreed that code clones have negative impact on quality and maintainability of the software (Fowler et al. (1999); Roy and Cordy, 2007; Tufano et al., 2015). (Rahman et al., 2012; Kapser and Godfrey, 2006)

Code duplication might lead to update anomalies (Rajapakse and Jarzabek,

2005). That is, if the cloned code is modified in one or more fragments, but not all of its clones are. For example, if the piece of code to be cloned contains a bug, that bug will be repeated along with the cloned fragment. Vice versa, if a bug is found in a code fragment that has been cloned, all clones should be inspected in case the bug exists. Not only bugfixes but also other modifications need special attention if duplicated code exists. Thus, maintenance costs are increased due to code duplication (Roy and Cordy, 2007).

Although the original cloned code fragment would not contain a bug, the reuse in itself might introduce a new bug to the system (Roy and Cordy, 2007). It is at the responsibility of the developer to adapt the cloned code fragment successfully to its new environment.

Duplicated code inevitably increases the overall amount of code, which results in two drawbacks. First, more redundant code equals more code to understand (Johnson, 1993). This especially complicates maintenance of legacy systems, because the developers, who inherited the software, have more code to look through. Thus, modifications and improvements in the system become more difficult (Roy and Cordy, 2007). Second, increase of the amount of code increases the overall system size. While the size increase is usually not a significant drawback, it is a problem for example for compact devices (Roy and Cordy, 2007).

Overall, detecting code clones aids the process of refactoring. Fowler et al. (1999) argue that code duplication yields bad design. That is due to lack of abstraction, for example inheritance structure. In other words, reducing the amount of duplicated code improves the design of the software. Furthermore, widely reused code fragments can be recognized to be good library candidates. Then, one would still have well tested code of which can be use widely. However, there would only be one place where to fix bugs, if they are found. (Roy and Cordy, 2007)

2.2 Clone Types

A clone relation exists between the code fragments, if the code fragments are similar. However, the concept "similar" is vague in the context of code clones. The terms and the categorization of types depend on the algorithms used, given threshold values and the format or visualization of the output. Nevertheless, there is a consensus on the general type categorization and the

most common terms of code clones (Roy and Cordy, 2007; Roy et al., 2009; Asaduzzaman, 2012):
Types I-III are textually similar code clones, type IV functionally similar:

- Type I clones are also generally called *exact matches*. The code fragments of type I clones may only differ in layout, whitespace or comments. (Baxter et al., 1998)
- Type II clones are known as *parametrized clones* or *near-miss clones* (Roy, 2009). In addition to possible type I differences between the code fragments, type II clones may differ in identifier names or types, or literal values.
- Type III clones are also known as *near-miss clones* (Roy, 2009). In addition to both type I and II differences, type III may have additions, modifications and deletions within the code fragment.
- Type IV clones are semantically similar code clones. In other words, the code fragments of type IV clone are functionally similar, but may not syntactically remind one another. (Krinke, 2001)

2.2.1 Type I Clones

Type I clones are exact clones (Baxter et al., 1998). That is, a type I clone fragment is either identical with the original code fragment or differs only in whitespace, comments or layout. In Table 2.2 there is an example of a type I clone pair. Consider the code fragment 1 as the original function. Code fragment 2 is otherwise exactly the same as the original, but from the line 2 some whitespace have been removed, layout between lines 3 and 4 have been changed, and a comment has been added to the returning statement.

For clone detection tools, type I clones are easiest to find from all clone types. Since they are usually a result of a simple copy and paste duplication, only minor preprocessing of the code base is needed to find them. Whitespace and comments are easily removable. However, changes in layout might be harder to detect for techniques that compare code line-by-line. (Roy and Cordy, 2007)

2.2.2 Type II Clones

In addition to type I variations of cloned code fragments, type II clones might vary in names of user-defined identifiers, types of identifiers or literal values

Code Fragment 1	Code Fragment 2
1 `int sum(int n) {`	1 `int sum(int n) {`
2 `int sum = 0;`	2 `int sum=0;`
3 `for (int i=1; i<=n; i++) {`	3 `for (int i=1; i<=n; i++)`
	4 `{`
4 `sum = sum + i;`	5 `sum = sum + i;`
5 `}`	6 `}`
6 `return sum;`	7 `return sum; //return`
7 `}`	8 `}`

Table 2.2: Type I clone pair. Code fragments only differ in layout, commenting and whitespace.

Code Fragment 1	Code Fragment 2
1 `int sum(int n) {`	1 `float sum(int n) {`
2 `int sum = 0;`	2 `float sum = 1.0;`
3 `for (int i=1; i<=n; i++) {`	3 `for (int j=1; j<=n; j++) {`
4 `sum = sum + i;`	4 `sum = sum + j;`
5 `}`	5 `}`
6 `return sum;`	6 `return sum;`
7 `}`	7 `}`

Table 2.3: Type II clone pair. Identifier names and types, and literal values can be changed.

(Roy, 2009). Thus, type II clones are near-miss clones. In the example in Table 2.3, type of identifier *sum* has been changed, variable i is renamed to j and *sum* is initialized to 1.0 instead of 0.

Type II clones are also called *renamed clones*. The renaming of identifiers does not need to be consistent. However, if the renaming is consistent, the duplicated code fragments are called *parameterized* or *p-match clones*. In Table 2.4 the code fragments are renamed clones, but not p-match clones. (Roy and Cordy, 2007)

2.2.3 Type III Clones

In addition to possible type I and II changes, type III clones can have statements changed, added or deleted (Table 2.5). Thus, type III clones are also near-miss clones. (Roy, 2009)

Code Fragment 1	Code Fragment 2
1 **if** (i < j) {	1 **if** (i < j) {
2 i = 0;	2 j = 0;
3 j++;	3 i++;
4 }	4 }

Table 2.4: Code fragments 1 and 2 form a type II clone pair. However, if only p-match clones are searched, they are not considered as clones. The logic between the parameters is inconsistent.

Code Fragment 1	Code Fragment 2
1 **int** sum(**int** n) {	1 **int** sum(**int** n) {
2 **int** sum = 0;	2 **int** sum = 0;
3 **for** (**int** i=1; i<=n; i++) {	3 **for** (**int** i=1; i<=n; i++) {
4 sum = sum + i;	4 sum = sum + i;
	5 sum = sum + 10;
5 }	6 }
6 **return** sum;	7 **return** sum;
7 }	8 }

Table 2.5: Type III clone pair. A statement has been added to the line 5.

Type III clones are also called *gapped clones*, since they can be formed of two or more type I or II clones that have diverging statements between them. Therefore, type III clones do not need to be detected directly, but can be aggregated from the detection results of nearby clone pairs. However, clone detection tools usually have a threshold value for minimum amount of lines or tokens for a code fragment to be detected as a clone. In consequence, if a gapped clone is not detected as clone as it is, too short type I or II clone with a gap could be missed by the tool. (Ueda et al., 2002b)

Finding type III clones is interesting from the maintenance perspective. Usually they are a result of a copy paste duplication, where new functionality is made out of old one with slight changes. This begs the question whether there was a real need for the duplication.

2.2.4 Type IV Clones

Type IV clones are functionally similar code blocks. Thus, if two code fragments do the same computation but with different syntax, they are type IV clones. For example, two functions which produce the same output with

Code Fragment 1	Code Fragment 2
1 **int** sum(**int** n) {	1 **int** sum(**int** n) {
2 **int** sum = 0;	2 **int** sum = 0;
	3 **int** i = 1;
3 **for** (**int** i=1; i<=n; i++) {	4 **while** (i<=n) {
4 sum = sum + i;	5 sum = sum + i;
	6 i++;
5 }	7 }
6 **return** sum;	8 **return** sum;
7 }	9 }

Table 2.6: Type IV clone pair. The for loop has been changed to a while loop.

the same input can be recognized as type IV clone (Table 2.6). (Krinke, 2001)

Reordered clones are also of type IV, where statements of copied code fragment are reordered but still result in the same output (Komondoor and Horwitz, 2001). Type IV clones need more advanced methods for detecting them, since data flow is harder to compare than textual similarity. However, type IV clones are interesting to find from maintenance perspective, since they can be accidental duplications.

2.3 Clone Detection

2.3.1 Clone Detection Process

The clone detection process aims to find code clones with adequate precision and recall. High precision indicates that most of the reported clone candidates are relevant clones. That is, there are not too many false positives. Furthermore, high recall indicates that most of the relevant clones are found. In other words, there are not too many false negatives. (Bellon et al., 2007)

Since it is not known beforehand, which code fragments are duplicated and where, the clone detection tool inevitably needs to compare all code fragments to every other code fragment. Thus, the space and time requirements may impact to the selection of the clone detection technique with large code bases. (Roy et al., 2009)

The steps of the clone detection process are presented in Figure 2.5.

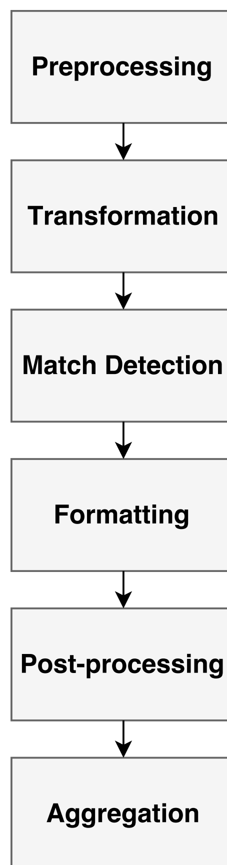

Figure 2.5: The generalization of the clone detection process.

Preprocessing
Preprocessing the code base reduces the amount of comparable code and prevents false positives (Roy et al., 2009). Uninteresting parts are removed in order to exclude them from clone detection. For example, if the used framework generates code, the generated files and parts of code can be removed in preprocessing stage.

Another preprocessing method to prevent false positives is partitioning the code before analysis. Code clones can be of different level of granularities. The granularity of a clone can be fixed or free. The fixed granularity means that the clones are detected based on some syntactic boundary of the programming language (Roy and Cordy, 2007). A syntactic boundary can be, for example, a *begin* - *end* block or a function. If the clone detection technique is granularity free, the syntactic boundaries are ignored in the detection process. Thus, *spurious clones* might be detected. Spurious clones are clone

Code Fragment 1	Code Fragment 2
1 **return** result;	1 **return** x;
2 }	2 }
3 **int** sum() {	3 **int** prod() {
4 **int** sum = 0;	4 **int** prod = 0;

Table 2.7: A spurious clone. The code fragments can be detected as clones, even though they are not interesting from the maintenance perspective (Roy and Cordy, 2007).

fragments that are detected as matches, but are not really clones from the maintenance perspective (Roy and Cordy, 2007). For example, a granularity free token-based approach may consider the code fragments in Table 2.7 clones, since their token sequences match.

A source unit is a code fragment that is the largest fragment that is compared with other source units. The source units can be any level of granularity, for example the functions or the *begin - end* blocks. The source units are further partitioned into comparison unit, the actual units that are compared with one another. The comparison units of the source units can be, for example, lines of statements or tokens.

Transformation
The amount of false negatives is reduced by normalizing the code in the transforming phase. Pretty printing the source code is transforming it into a standard form. In consequence, cloned code fragments with differing layout would be recognized (Roy and Cordy, 2007). Other measures to prevent false negatives are removing whitespace and comments. These transformations are sufficient to find type I clones.

In order to find all type II clones, the source code may need normalizing identifier names, types or values (Kamiya et al., 2002). Some clone detection approaches need the source code to be transformed into other intermediate internal representation form: the code may require tokenization, parsing or calculation of control and data flow analysis. The next section describes each detection approach and their transformation in more detail.

Match Detection
In match detection phase, the transformed code is given as input to the algorithm that does the actual clone detection. The clones are searched and

adjacent clones are aggregated by the terms of clone granularity. The output is the clone pairs. (Roy et al., 2009)

Formatting
The clone pair list formed in the match detection phase is mapped to the original source code. The source coordinates of the clone pairs are stored through the entire clone detection process. (Kamiya et al., 2002)

Post-processing
Post-processing includes either automatically or manually filtering false positives. Visual interpretation of clone detection results is discussed in section 2.4 in more detail.

Aggregation
Last, clone classes are aggregated from the detected clone pairs (Roy et al., 2009).

2.3.2 Clone Detection Techniques

The clone detection techniques can be categorized into six different types of approaches: text-, token-, tree-, PDG-, metrics-based or hybrid. This chapter summarizes the basic principles of each detection technique.

Text-based approaches
Text-based approaches usually compare raw source code to detect similarity between the code fragments. Little or no transformation is used. Some normalization of the layout can be done in order to detect type I clones more accurately. For example, whitespace and comments might be removed. Thus, text-based approaches are usually more language independent, since the code is considered only as a string. (Ducasse et al., 1999)

Different string matching algorithms are used in order to improve the performance of the clone search. For example, Johnson (1993)) uses Karp-Rabin string searching algorithm. It calculates hashes for substrings, called *fingerprints* that consist of a fixed number of lines of the source code. These fingerprints are compared instead of string characters. Equal fingerprints indicate that the original code fragments are clones. (Bellon et al., 2007; Roy, 2009)

Token-based approaches
In token based approaches the comparison is done between sequences of to-

kens. A lexer or parser is used to transform the source program to a comparable form. For example, CCFinder is a token-based tool for clone detection (Kamiya et al., 2002) (figure 2.6). It first performs a lexical analysis, where the program code is converted into token sequence based on the lexical rules of the used programming language. Second, transformation rules are applied. For example, the identifier names and types are replaced with a special token in order to detect type II clones. Third, the transformed token sequence is scanned for equivalent substrings. These matched substrings are the detected clone pairs. Fourth, the positions of the substrings in the token sequence is converted into the original line numbers in the original source files. The mapping information about the original formatting is stored through the entire process.

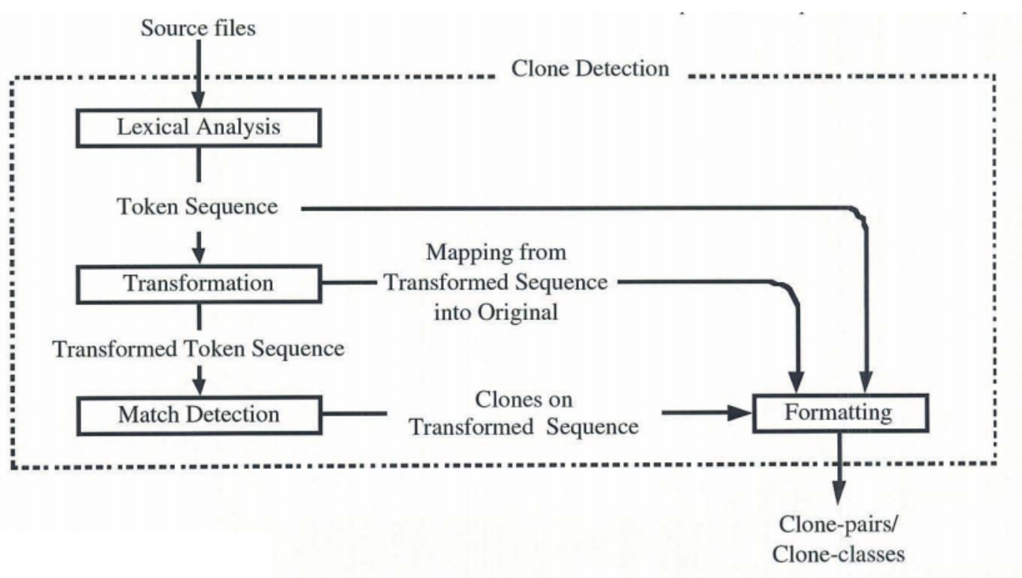

Figure 2.6: Clone detection process of CCFinder (Kamiya et al., 2002).

The need for lexical analysis make token-based techniques more language dependent than text-based approaches, but enables them to detect also near-miss clones in addition to exact clones. In other words, language dependency increases the need for preprocessing and transformations, but enables more advanced clone detection. (Roy et al., 2009)

Tree-based approaches
An abstract syntax tree (AST) or a parse tree is produced from a parsed source code. It is formed with a parser of the programming language used in the source code (Roy and Cordy, 2007). The tree contains all needed

information from the source code (Wahler et al., 2004). It can be processed for detecting clones using tree matching algorithms. (Roy et al., 2009)

Baxter et al. (1998) are pioneers of AST based techniques with their tool CloneDr. They use alternative tree representations to improve performance. Additionally, they apply hashing for the subtrees in order to reduce the number of needed comparisons. (Asaduzzaman, 2012)

PDG-based approaches
Some approaches use program dependence graphs (PDGs) to consider both syntactic and semantic similarities. The vertices of the graph represent the statements of the program code. The data and control flows are described as edges. Thus, the dependencies between graph vertices can reveal the semantic similarities in the source code. The clone detection is done by searching isomorphic subgraphs. Even though PDG-based techniques in general find more different types of clones compared to other techniques, they have scalability issues (Roy et al., 2009). (Krinke, 2001)

Metrics-based approaches
Instead of comparing the source code directly, different software metrics can be calculated and compared (Roy and Cordy, 2007). For example, Mayrand et al. (1996) calculate function metrics for layout, expressions and control flow graphs. These can include, for example, number of lines of source code, number of calls to other functions and number of arcs in the control flow graph. Similar metrics between functions indicate they are clones. Di Lucca et al. (2002) use software metrics to detect duplicated web pages. Calefato et al. (2004) first use the calculated metrics of a web application to offer potential function clones. After that, the user needs to then visually analyze whether the candidates are really clones of each other.

Hybrid approaches
Each technique has its own limitations. To overcome limitations, some tools combine different techniques. For example, Gabel et al. (Gabel et al., 2008) utilize the tree-based technique of Jiang et al. (2007) to improve scalability with their PDG-based approach. Another hybrid approach is used with a tool Nicad. Nicad utilizes both text-based approaches and ASTs to detect both exact and near-miss clones (Roy, 2009). Due to the use of the AST, the source code needs more preprocessing, but the technique is more robust in finding type III clones.

In summary, text- and token-based approaches need less preprocessing than

tree-, PDG- and metrics-based approaches. Some of the tree-, PDG- and metrics-based approaches may even need a full parser for preprocessing. Thus, they are very language dependent. However, text- and token-based approaches generally find mainly exact and type II near-miss clones, while other approches are more robust to reordering, adding and deleting of statements. PDG- and metrics-based approaches find the most semantic clones compared with the other approaches. (Roy et al., 2009)

The approaches for clone detection only return the clone pairs or classes. Visualization of the data helps inspection of the cloning situation in the source program. Different visualization techniques for code clones have been proposed. The most common visualization techniques are presented in the chapter 2.4.

2.4 Clone Visualization

Code clone detection tools may produce large amounts of data (Rieger et al., 2004). Due to the overwhelming results, the comprehension of the duplication situation may be difficult. Thus, it is useful to adopt the *Visual Information Seeking Mantra* of Shneiderman (1996): overview first, zoom and filter, then details-on-demand (Asaduzzaman, 2012).

Asaduzzaman (2012) and Zibran (2015) have summarized the used visualization techniques in the field of code clone detection. There are a lot of different visualizations used, but describing all of them is outside the scope of this thesis. The visualization techniques of the clone detection results that are used in the analysis phase, are described in this chapter. Other common clone visualization techniques are described briefly. The first section presents the techniques that gives the user an overview of the cloning situation of the source code. The second section describes, how the results can be visualized in more detail.

2.4.1 Getting an Overview of the Cloning Situation

A scatter plot is a common way to visualize clone pairs (Asaduzzaman, 2012; Church and Helfman, 1993; Ueda et al., 2002c). A scatter plot, of which Church and Helfman (1993) discussed as a dot plot, gives an overview of the clone data by revealing patterns. In Figure 2.7 there is a scatter plot of a source code. Letters from A to H represent files of which boundaries are drawn. Both the vertical and horizontal axes represent the source code under

comparison. The unit of the axes can be token sequences or line numbers of the source files. Each dot represents a clone between the comparable units.

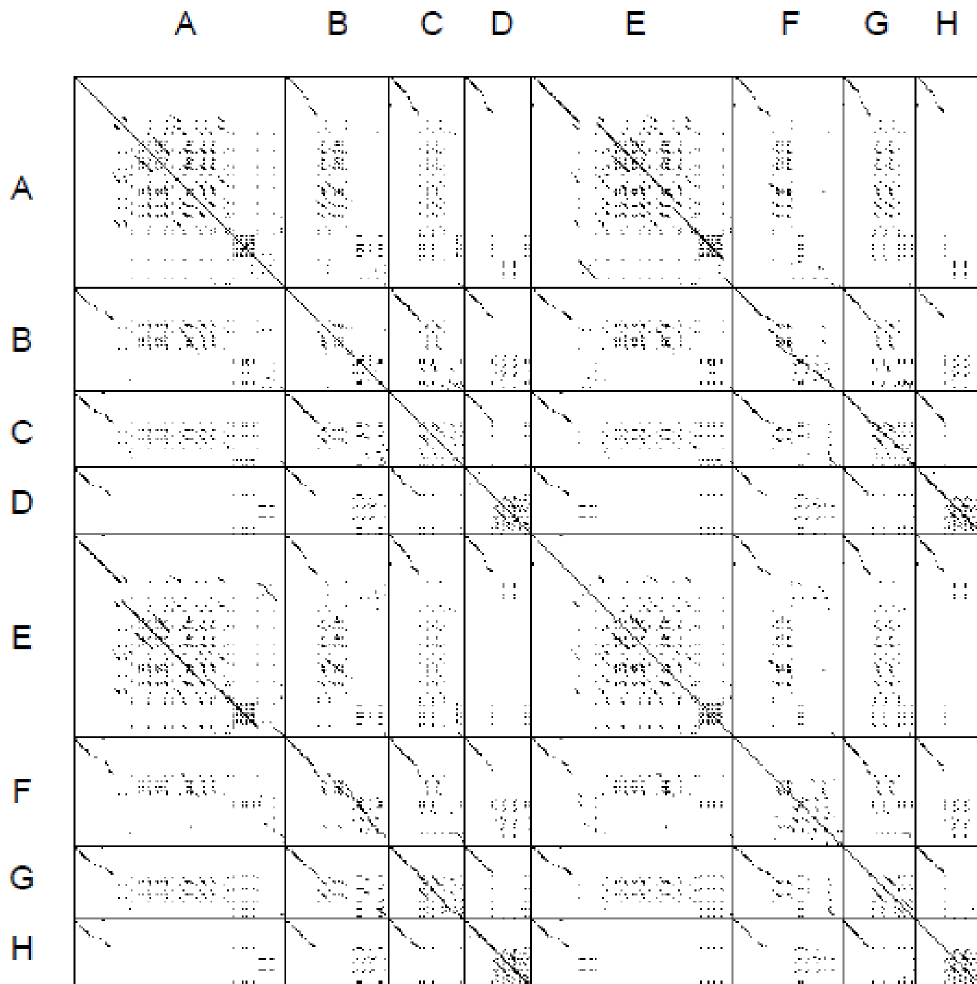

Figure 2.7: A scatter plot. The letters from A to H represent files. The dot textures represent clone pairs between the corresponding source code lines. (Church and Helfman, 1993).

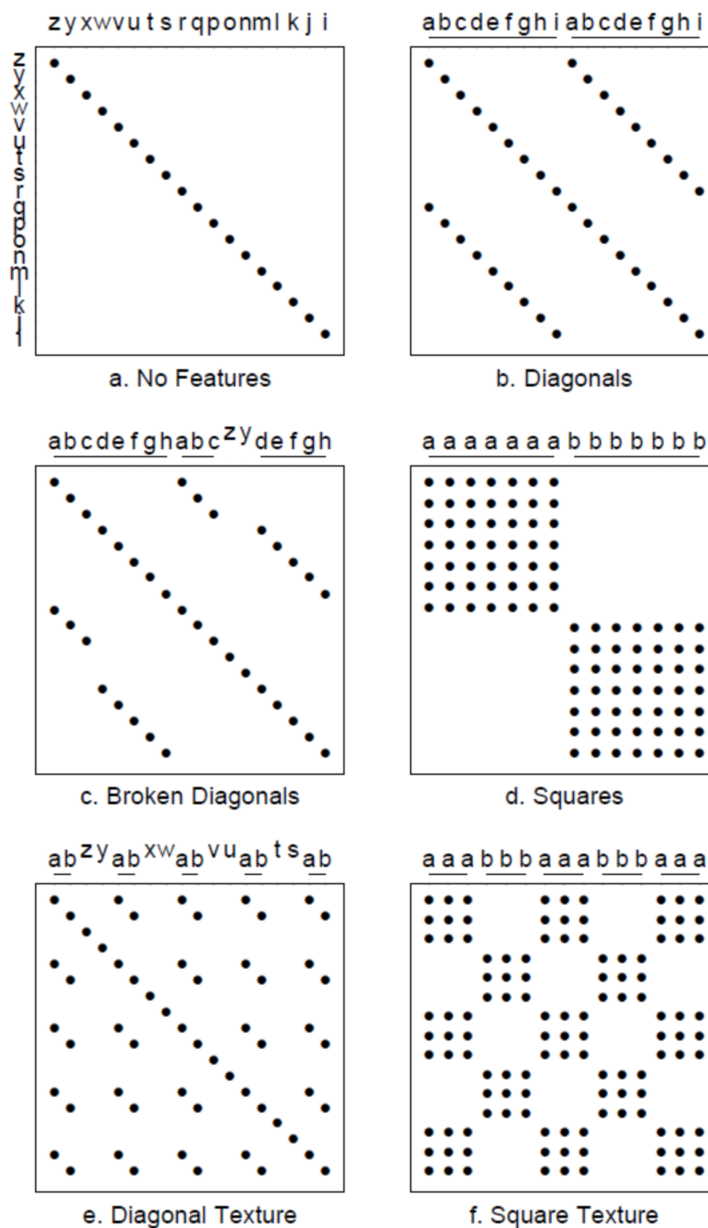

Figure 2.8: Examples of different kind of patterns in a scatter plot. (Church and Helfman, 1993).

The Figure 2.8 shows the interpretations of different kind of patterns occuring in scatter plots. A letter in the example can be considered a source code line. Since each source code line is an exact copy of itself, there lies a descending diagonal row of dots from the top left to the bottom right

corner (Figure 2.8a). The chart is symmetric with the relation to this diagonal. The diagonals elsewhere represent repetition of code fragments, in other words, code clones (Figure 2.8b). The broken diagonals reveal type III clones, where statements are either added or removed (Figure 2.8c). The squares can indicate the detection technique has found false positives, for example, a consecutive variable initializations (Figure 2.8d). Moreover, the regular texture patterns (Figure 2.8e and f) may yield false positives but also some interesting cloning patterns.

Different kinds of graphs are widely used in code clone visualizations (Rieger et al., 2004; Lanza and Ducasse, 2003). The source code files are usually presented as the nodes and the cloning relationships as the edges. Different visual variables can be used to gain a better overview of the cloning situation. For example, size of the graph node often describes amount of internally copied code, and width of the edge externally copied code (Rieger et al., 2004).

The graph can also be circular (Hauptmann et al., 2012; Rieger et al., 2004). For example, the Figure 2.9 is a chord diagram from the tool AtomiQ[1]. The containing folders are drawn as arcs (such as SourceFolder1). Nodes on the circle represent source files (SourceFilex.aspx). Edges between the source file nodes represent clone relationships. The width of the edge correlates with the size of the clone. The chord diagram only includes the cloned lines of the files containing clones. That is, the lines that have no clones are hidden, and the files containing no clones are not drawn at all. This may reduce visual cluttering, but does not give an overview of the entire situation. Additionally, the location of the cloned code fragments cannot be interpreted from the chord diagram.

[1] AtomiQ `http://www.getatomiq.com/overview` referred 16.10.2017

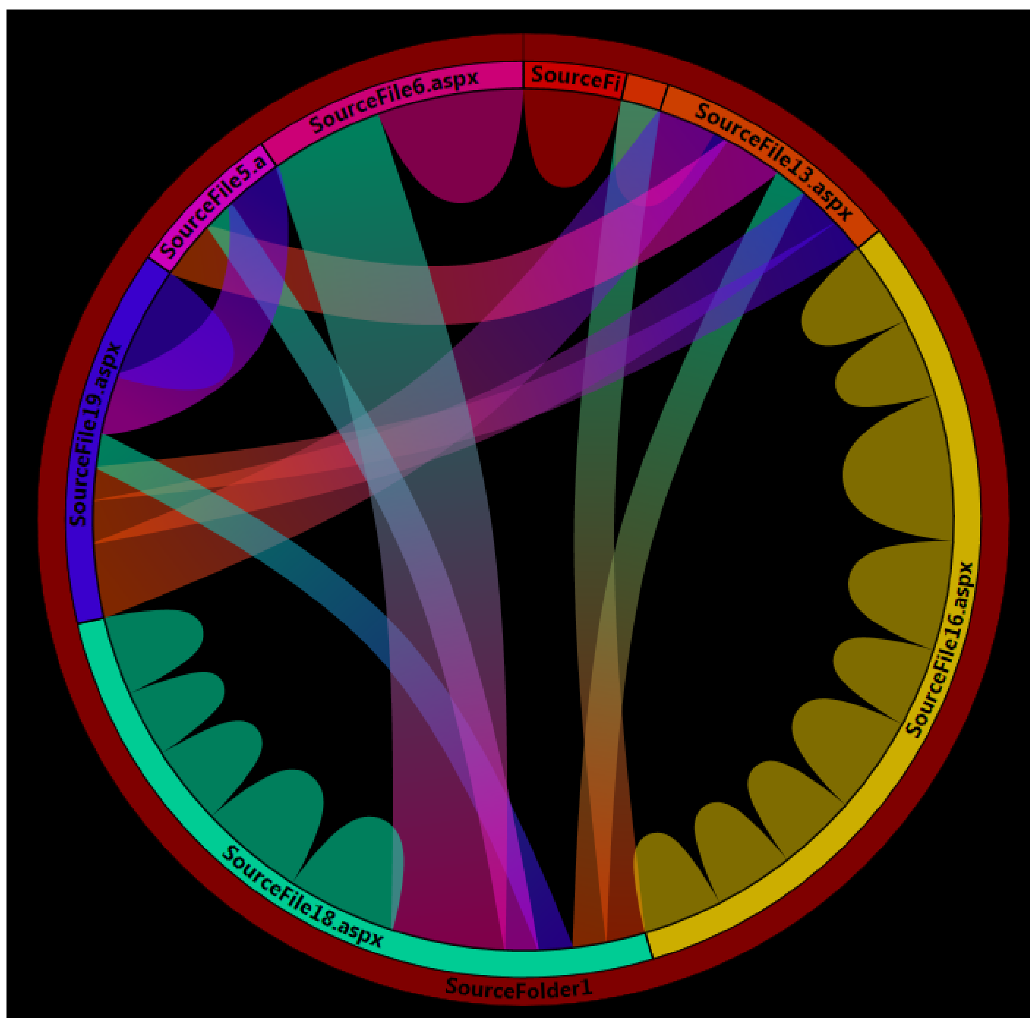

Figure 2.9: A chord diagram of the tool Atomiq. The containing folders are drawn as arcs, source files as nodes and cloning relationships as edges.

Visual cluttering and overplotting can also be controlled with good user-interaction properties in the clone visualization tool. This can be achieved by giving the possibility to filter out candidates and zooming to interesting views. For example, clone candidates could be filtered out based on the level of similarity or the number of source code lines. The overview aids the user to choose which files are worthwhile to compare in detail. The most detailed view in a code clone visualization tool is naturally some sort of source code view. (Asaduzzaman, 2012)

2.4.2 Viewing the Differences in the Source Code

Many of the tools use simple source code comparison view as the most detailed level in the tool (Ueda et al., 2002c; Asaduzzaman, 2012). Usually, the cloned fragments are highlighted (Figure 2.10). The tools generally provide a difference view for a clone pair to be investigated at a time.

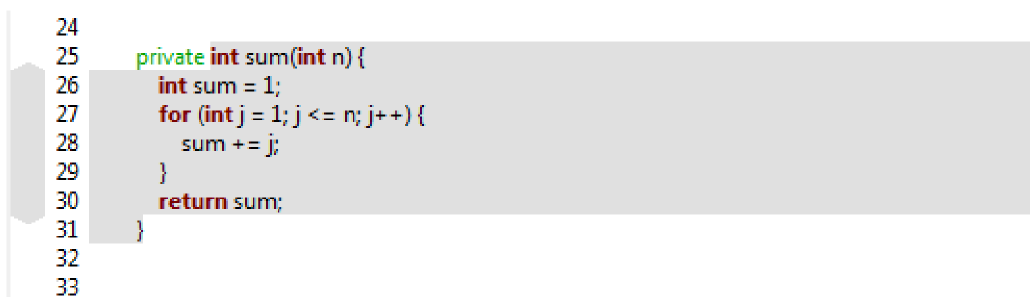

Figure 2.10: Source view of GemX. The cloned code fragment is highlighted with gray color. The gray highlight on the left side of the line numbers indicates the code fragment has been cloned externally to another file.

The Figure 2.10 is the source code view from the tool GemX[2]. The gray color indicates a cloned code fragment, which in this case is the entire *sum* function. The gray bar on the left side of the line numbers indicates the clone pair of the cloned code fragment is located in another file. A gray bar on top of the line numbers would mean its clone pair is located within the same file.

Again, the user-interaction properties of the tool play a key role in the usability of the source code view of the tool. Many of the tools provide navigation between the files of the cloned code fragments (Johnson, 1996). In addition, many of the tools enable comparing the cloned code fragments or clone classes side by side (Tairas et al., 2006; Zhang et al., 2008).

[2]GemX `http://www.ccfinder.net/doc/10.2/en/tutorial-gemx.html` referred 16.10.2017

Chapter 3

Used Techniques

3.1 Tool Evaluation

In 2009, Roy et al. conducted a comprehensive comparison of the different state-of-the-art clone detection tools and techniques of that time. They put together an evaluation, of which clone types (of I, II, III and IV), and in which scenarios each technique or tool would recognize, and how well. In addition, they organized the properties of the techniques and tools into 10 different facets:

1. *Usage*
   The examined usage facets were platform, external dependencies and availability. The platform indicated whether the tool was platform independent, or it could be run on Windows or Linux/Unix environment. In the external dependencies, the additional tools that the technique would require were listed, such as another tool for transformations. The availability of the tool specified if the tool was commercial, open source or otherwise available.

2. *Interaction*
   The interaction facet is categorized into three different sections. First, what kind of user interface it offers: is it only a command line tool, or does it offer graphical user interface (GUI), or both. Second, the nature of the output is described. The tool offers source coordinates of the cloned fragments, or graphical or abstract visualization of the output, or both. Third category is about the integrated development environment (IDE, for example Eclipse or Visual Studio) support. The tool could have no IDE support, or it could be a plug-in or integrated/dependent of an IDE.

3. *Language support*
   Language support describes whether the technique is applied with procedural or object oriented languages. Additionally, the languages that the tool is experimented with are reported.

4. *Clone Relation*
   Clone relation describes, does the technique yield clone pairs or classes, or does it group clone pairs in post-processing phase. Clone granularity can be fixed or free. Typical fixed granularities are, for example, function/method or begin-end block.

5. *Techniques*
   The algorithms that the tool uses are listed. The comparison granularity can be of different levels. For example the algorithm might compare single tokens, or entire AST subtrees, or even only identifiers and comments. Worst case computational complexity is given for those techniques for which it is known.

6. *Adjustments*
   The tool might need pre- and/or post-processing. Additionally, some techniques use different kinds of thresholds or heuristics. For example, a technique might only recognize code fragments as clones if the fragments are at least 15 lines long.

7. *Transformations and Normalizations*
   The needed or used transformations and normalizations are listed. For example, comments and whitespaces might need to be removed for the tool, or the tool might need language dependent transformation rules.

8. *Internal Representation*
   The internal representation of the code after the applied normalization/transformation is decribed. For example, the code might be pretty-printed and comments removed, or the representation might be an abstract syntax tree (AST).

9. *Program Analysis*
   Program analysis indicates what the technique needs for presenting the intermediate representation. For example, the tool might need regular expressions to remove comments or whitespace. It might even require a full parser, if the technique is very language independent.

10. *Evaluations*
    There might have been studies that have empirically validated the tool.

Whetherthe results of the validations are available or not is another category. The most common subject systems that have been used in the validations are provided.

The study of Roy et al. (2009) is used as a base for the tool selection process. However, since the study is from 2009, some of the information might be out of date. For example, availability and language support of the tools might have changed. Nonetheless, based on the facets listed by Roy et al. (2009), we define our own criteria. Hence, we are searching for a tool that:

- is free and available (including possible external dependencies)
- runs preferably on windows
- finds as many different clone types as possible
- either presents the output visually or the clone detection output is adaptable to another tool that can visualize them
- preferably yields clone classes or groups clone pairs in post-processing phase. A clone in our case studies can potentially be found in several branches.
- has adjustable threshold values of similarity. User-interaction properties aid to explore the data.
- is preferably granularity free. Spurious clones would not be false positives with our case studies. We are more interested in the uncloned parts.
- is either language independent or already adapted to analyze Visual Basic or C# code. Since our case studies are .Net code, tools intended for analysing .Net applications are preferred.

However, there are visualization tools that did not exists at the time of the research of Roy et al. (2009). The tool *VisCad*[1] (Zibran, 2015; Asaduzzaman, 2012) can visualize output of several clone detection tools, and results of other detection tools can be converted to form of VisCad input file. In addition, the tool AtomiQ[2] can detect type I clones itself and offers a different kind of source view.

[1] VisCad `http://homepage.usask.ca/~mua237/viscad/styled-3/index.html`

[2] AtomiQ `http://www.getatomiq.com/` referred 7.10.2017

Possible clone detection tools for visualizing with VisCad could be *Nicad* or *Clone Detective*. They satisfied other criteria but did not offer visualization. Nicad is currently known as Nicad4[3]. C# support has been added to it. Clone Detective was intended to use with .Net code. However, Clone Detective seems deprecated.

Nonetheless, there are currently other tools for .Net applications. The IDE that we have in use, Visual Studio, has integrated tool for clone detection[4]. However, it does not export results outside Visual Studio. There is also a tool *DupFinder*[5]. DupFinder has extensive number of different options. For example, the similarity cost can be influenced. It can be chosen, which kind of type II clones are detected. For example, whether to consider a changed value of a variable a clone or not. Since DupFinder is intended to use with .Net applications, and it has extensive adjustability options, we choose to test it with VisCad.

Other qualified tools are a clone detection tool *iClones* with a visualization tool *Cyclone*[6], and *CCFinderX* with *GemX*[7]. Neither of them supported Visual Basic or C# at the time of the study of Roy et al. (2009), but both support C# currently. Other tools did not satisfy the criteria, or could not be found anymore. For example, CloneDr[8] was discarded, because it is commercial.

In Table 3.1, the visualization techniques and types of the potential tool combinations are summarized. Each tool includes a source view and list view. List views group the clones to classes and usually provide interaction with the other views. For example, GemX highlights the selected clone classes. The problem with nearly all the source views is that they are intended for comparison of two files only. That is also the case with the source view of the tools GemX, Cyclone and VisCad. However, the source view of AtomiQ shows how many external clones the code fragment has and where. Hence, AtomiQ is chosen for the analysis. At the most detailed level, it does not matter that it only shows exact clones. This just needs to be considered during the analysis. As stated in chapter 2.4.1, the chord diagram hides the

[3]Nicad4 `https://www.txl.ca/nicaddownload.html`
[4]`https://msdn.microsoft.com/en-us/library/hh205279.aspx`
[5]DupFinder `https://www.jetbrains.com/help/resharper/dupFinder.html`
[6]iClones with Cyclone `http://www.softwareclones.org/cyclone.php`
[7]CCFinderX with GemX `http://www.ccfinder.net/ccfinderxos.html`
[8]CloneDr `http://www.semdesigns.com/Products/Clone/` referred 7.10.2017

Detection Tool	*Visualization Tool*	*Visualization Techniques*	*Clone types*
AtomiQ	AtomiQ	Chord diagram, list view, source view	I
CCFinderX	GemX	Scatter plot, list view, source view	I, II
DupFinder	VisCad	Scatter plot, treemap, list view, hierarchical dependency graph, source view	I, II
iClones	Cyclone	Treemap, list view, evolution view, plotview, source view	I, II

Table 3.1: Tools chosen that were chosen for further inspection. The tools CCFinderX with GemX and AtomiQ were chosen for the analysis.

files and code fragments that do not include clones, so it does not provide an overview of the cloning situation. Hence, it is not useful with our case studies.

Instead, scatter plot provides an overview of the entire cloning situation between all files and directories. In our case studies, the branches can be considered to be the directories at the highest level of hierachy. It is known that the branches are technically clones of each other. Basically, we want to find the fragments that are not clones. Since normally clones are the ones being searched, all visualization techniques will not suite our purposes. However, the tool Gemini with scatter plot has previously been used for detecting the gaps from the gapped clones (Ueda et al., 2002b). Hence, a tool that visualizes with scatter plot is wanted for the analysis. From the possible tools, GemX and VisCad provide a scatter plot. However, the scatter plot of VisCad is too generalized for our purpose. It does not provide detailed information about the locations of the clones in the files. Thus, GemX, which is actually a newer version of Gemini, is chosen for the analysis.

The other visualization techniques with the possible tools are not expected to provide better overview than scatter plot. Treemap and hierarchical dependency graph would provide valuable information about the distribution of clones. However, we already know the branches of the case studies are expected to be clones of each other. Thus, VisCad is discarded. The evolution and plotviews of Cyclone are intended for studying the evolution of clones, which is not relevant from the perspective of our research. Hence, iClones

with Cyclone is also discarded.

All in all, the tools GemX and AtomiQ together seem to provide the information we need for the explorative analysis. Both of them support C#. Additionally, both tools can also visualize front-end files of our case studies. GemX can interpret them as plain text, whereas AtomiQ can directly read .Net aspx extension, javascript and XML files. Naturally, only type I clones are interpreted from front-end code with GemX also. The following section gives overviews of the both tools.

3.2 Chosen Tools

CCFinderX with GemX
CCFinderX is a rewritten version of CCFinder (Kamiya et al., 2002). Similarly, Gemini (Ueda et al., 2002c; Ueda et al., 2002a) is a predecessor of GemX. The tools have several publications of their use and evaluation (Svajlenko and Roy, 2014; Bellon et al., 2007; Kamiya et al., 2000). GemX uses CCFinderX internally. CCFinderX uses a token-based approach to detect clones. CCFinderX finds clones of types I and II. The following parameters can be adjusted with GemX:

- minimum clone length in tokens. For example, if minimum clone length is set to 50, only clones with at least 50 tokens are recognized as clones.
- minimum number of different kinds of tokens in code fragments (TKS). If minimum TKS is set to 15, only code fragments with 15 different kinds of tokens are recognized as clones. For example, if there are only consecutive variable assigments in two different code fragments. Then, there are not many different kind of tokens in the code fragment. Thus, the two code fragments are not considered as clones.
- sharper level with enumeration from 0 to 3 is an option. The documentation does not really specify what it does, but some empirical testing indicated it would affect the comparison granularity.
- p-matches can be searched
- pre-screening of clones can be used to avoid overplotting
- different metrics can be calculated. For example, the clone coverage of a file or count of code fragments of a clone. Metrics can also be used for filtering the results.

CCFinderX could be used separately from command line with various other options[9]. However, the extra options are not needed for the intentions of this thesis.

AtomiQ
AtomiQ has been evaluated to work in cross-project context (de Oliveira et al., 2015). We intend to analyze the branches of the case studies simultaneously, which is one kind of a cross-projects analysis. AtomiQ uses a text-based approach for detecting code clones. AtomiQ recognizes only type I clones. The possible configurations of AtomiQ are:

- minimum similarity length can be altered. Minimum similarity length indicates the minimum number of lines to be considered as a potential clone. For example, if the value is set to 10, only code fragments with a minimum of 10 matching lines are considered to be clones of each other.

- the files to ignore in the analysis can be configured. For example, a regular expression to ignore files with .generated file extension could be configured.

The relevant views of the tools from the perspective of this research are explained in detail in the analysis phase of the following chapter 4.

[9] `http://www.ccfinder.net/doc/10.2/en/tutorial-ccfx.html` referred 7.10.2017

Chapter 4

Analysis

4.1 Introduction

The research objectives of the analysis are to explore, what kind of information code clone visualizations can offer in the situation, where the code base has uncontrollably spread to multiple branches. In this case, the ideal situation would be that there would be a single maintainable code base. In other words, the application would be a product. In order to productize the application, the customizations of each client need to be determined. With each customization, a decision needs to be made for what to do with it. A customization:

- can be made configurable to the product
- can be decided to be part of the standard product
- or be discarded altogether.

The analysis phase of this thesis aims to determine whether the customizations of the product branches can be discovered by exploring the code clone visualizations. The research question is, *can code clones be used to create a technical approach to productization?*

The two programs in the need of productization are .Net web applications. The other application is smaller. It has spread to 4 branches, each branch consisting of approximately 3K LOC. This application will be from now on called as the test application. This test application will be used to validate whether the exploration of the code clone visualizations reveal any interesting customizations. If it does, the process of how to carry through and interpret the results of such analysis, will be the outcome of this thesis. The

other application is bigger, approximately 20K LOC, spread to 10 branches. The scope of this thesis is to define a process that can be used to analyze it similarly to the test application, based on the results of the explorative analysis.

The code clone detection tools are normally used to find single clones from the code base. However, in this research, we aim to find the modified fragments, by treating the different branches as clones of each other. Since the tools are not made for this purpose, the visualizations of the tools are not optimal for this purpose.

Here, the standard product base could be considered to be the code base that is similar to every other branch. That is, the part of the code base that is found to be cloned in each branch. However, since the used code clone detection tool, CCFinderX, finds type II clones also, the product base could contain some trivial modifications that still need to be taken into account in the product version. For example, if a variable value was hardcoded and changed between branches, CCFinderX would find it to be a clone and thus the modification would not be revealed. However, these kind of changes can be easily made configurable. Anyway, the point is to get an overview of how much the branches really differ, and collect especially the non-trivial differences between the branches. These changes then can be documented in order to later decide how to proceed with them.

The architectural decisions of how to implement the productized version and determining the product base are outside the scope of this thesis. Both the test application and the application for which the process will be applied, are legacy code and thus implemented with already outdated technologies. The productized versions of them are implemented with new technologies, and the decisions of which features will be included in the productized version will include more specifications than just examining what the old versions included. This does not reduce the need to still determine all customizations of each client. To be able to start using the same code base and the productized version of the application with a client, the productized version should include all customizations that the client has previously had or there needs to be a justified reason for the discarding of each removed customization.

4.2 Preparation of the Source Code

This is the preprocessing stage of the code clone detection process. The uninteresting files need to be removed from the directories in order to prevent the clone detector including them into the analysis. The selected clone detector tools, CCFinderX with GemX and AtomiQ, are language dependent. Thus, the back- and front-end code needs to be analyzed separately.

Here, the back-end is C# code. The .Net framework autogenerates a lot of files that have the same file extension as the normal C# files (.cs), so they have to be removed or otherwise discarded from the analysis. With our case studies, the files have .generated.cs extension, so they can be removed based on that. AtomiQ could be configured so that the generated files would not be analyzed, but the GUI in GemX does not offer an option for that. Thus, the generated files are removed manually.

The front-end code must be treated as plain text with CCFinderX, since it does not support any languages used there. In the .Net web forms based projects, the front-end files can be, for example, files with extensions such as .aspx. In addition to these .Net specific files, the projects can contain javascript, CSS and XML files. All the interesting front-end files need to be renamed to plain text files, so that the clone detector can perform any sort of analysis on them. That is, all the different extensions should be changed to .txt. AtomiQ recognizes all the relevant file types related to these .Net projects.

4.3 Exploration of the Code Clone Visualizations

Accordingly to the visualization mantra (Shneiderman, 1996), we start the analysis by getting an overview of the cloning situation. After that, we filter the results to appropriate partitions of the application code base. Finally, we examine the interesting details that the scatter plots produce, and make suggestions how to interpret the results. The scatter plot visualizations are drawn with the tool GemX and the source views are screen captures of either AtomiQ or GemX.

4.3.1 Interpretation of the Scatter Plot

In Figure 4.1 there is the scatter plot of one branch of the test application. In this image, as in all of the following scatter plot visualizations, the following applies:

- the light gray horizontal and vertical lines are file boundaries
- the light gray diagonal line represents the self-similarity within each file
- the brown and white indicates areas without clones
- The black patterns represent clones.

The cloning patterns appear in different shapes and sizes, but essentially all form of diagonals. A diagonal represents a sequence of statements that has been duplicated, in other words, a code clone. In the next two following pictures, the code clone patterns are explained in detail. Three files are highlighted in Figure 4.1. The file highlighted with red border is examined closer in Figure 4.2. Code fragments that are cloned within a single file are called *internal clones* (Rieger et al., 2004). The files highlighted with green borders are inspected in Figure 4.4. They are compared with each other. Code clones that are located in different files are called *external clones* (Rieger et al., 2004).

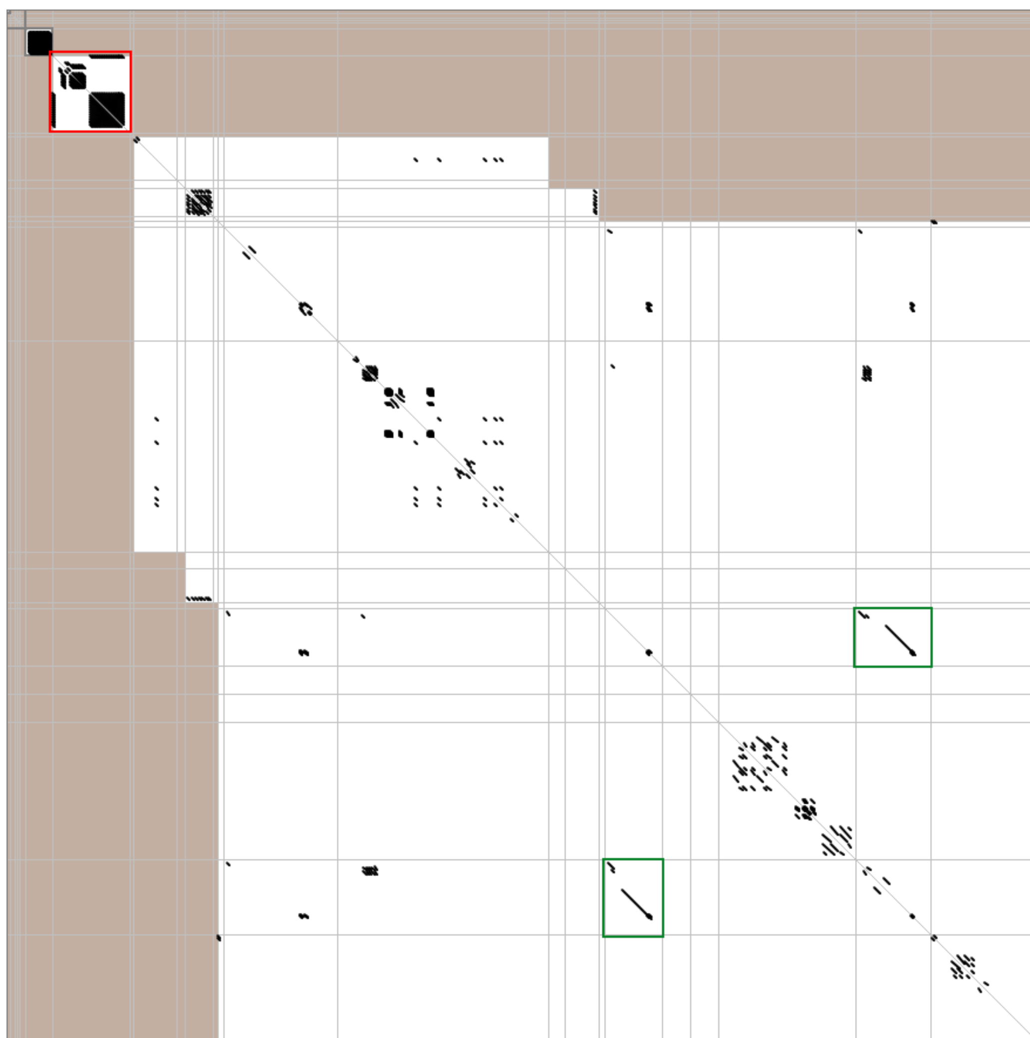

Figure 4.1: A scatter plot overview of a single branch of the test application.

Internal Duplication
In Figure 4.2 there can be seen square texture and diagonal patterns. Each diagonal represents a code clone that consists of consecutive similar code fragments. The square textures indicate a repetition of some code blocks. Here, the file consists of mainly *get-set* blocks. A source view of the code fragments highlighted with dashed lines is presented in Figure 4.3. CCFinderX seems to recognize type II clones with differing variable names and values, but not variable types. Each of the highlighted code fragments represent a boolean type property. The first property in the file is a boolean type, but the next boolean type properties start with line 286. Hence the horizontal and vertical rows of short diagonals. From the line 286 starts consecutive get-set blocks

of boolean properties. Each of them are clones with one another. Hence the square pattern.

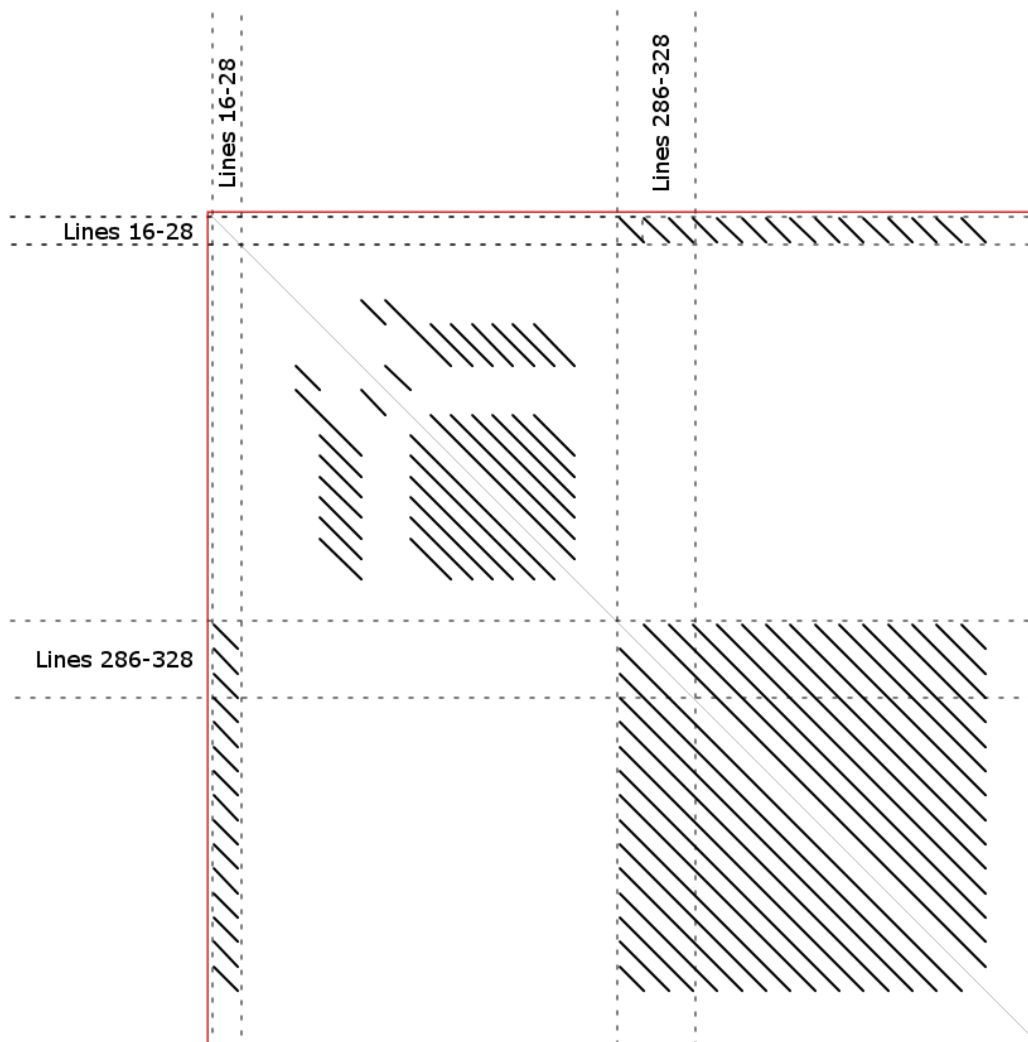

Figure 4.2: A scatter plot of the single file from Figure 4.1. Each diagonal represents *internal duplication*. The source view of the clones highlighted with dashed lines is presented in Figure 4.3.

```
16  public bool Regional
17  {
18    get
19    {
20      if (this["Regional"] == null)
21        return false;
22      return Conversions.ToBoolean(this["Regional"]);
23    }
24    set
25    {
26      this["Regional"] = Conversions.ToString(value);
27    }
28  }
29
30  [Display(Name = "City ID")]
31  public int CityId
32  {
33    get
34    {
35      if (this["CityId"] == null || Conversions.ToBoolean(this["Re
36        return 0;
37      return int.Parse(this["CityId"]);
38    }
39    set
40    {
41      this["CityId"] = value.ToString();
42    }
43  }
44
45  [Display(Name = "Min resolution for images")]
46  public string MinResolution
47  {
48    get
49    {
50      if (this["MinResolution"] == null)
51        return "1";
52      return this["MinResolution"];
53    }
54    set
55    {
56      this["MinResolution"] = value.ToString();
57    }
58  }
59
60  [Display(Name = "CSF File")]
61  public string CSF_
62  {
63    get
64    {
65      if (this["CSF"] == null)
66        return "";
67      return RelativePath.ToAbsolutePath(this["CSF"]);
68    }
69    set
70    {
71      this["CSF"] = value;
72    }
73  }
74
```

```
271  public string GeometryDB_
272  {
273    get
274    {
275      if (this["GeometryDB"] == null)
276        return "";
277      return this["GeometryDB"];
278    }
279    set
280    {
281      this["GeometryDB"] = value;
282    }
283  }
284
285  [Display(Name = "Draw Highlight Geometry")]
286  public bool HighlightGeometry
287  {
288    get
289    {
290      if (this["HighlightGeometry"] == null)
291        return true;
292      return Conversions.ToBoolean(this["HighlightGeometry"]);
293    }
294    set
295    {
296      this["HighlightGeometry"] = Conversions.ToString(value);
297    }
298  }
299
300  [Display(Name = "Hide daterange from results list")]
301  public bool HideDaterange
302  {
303    get
304    {
305      if (this["HideDaterange"] == null)
306        return true;
307      return Conversions.ToBoolean(this["HideDaterange"]);
308    }
309    set
310    {
311      this["HideDaterange"] = Conversions.ToString(value);
312    }
313  }
314
315  [Display(Name = "Order results alphabetically")]
316  public bool ShowAplhabeticalOrder
317  {
318    get
319    {
320      if (this["ShowAplhabeticalOrder"] == null)
321        return true;
322      return Conversions.ToBoolean(this["ShowAplhabeticalOrder"]);
323    }
324    set
325    {
326      this["ShowAplhabeticalOrder"] = Conversions.ToString(value);
327    }
328  }
329
```

Figure 4.3: A source view of the Figure 4.3. Screenshot is from GemX. Gray and turquoise highlighting of the code indicates that the fragment is a clone. The code fragments with turquoise background belong to the clone class that is currently selected in the tool. The vertical bar on top of the line numbers indicates a clone within the file. Tokens discarded from the analysis (such as comments, keywords for access modifiers) are colored in green. Reserved words have red coloring.

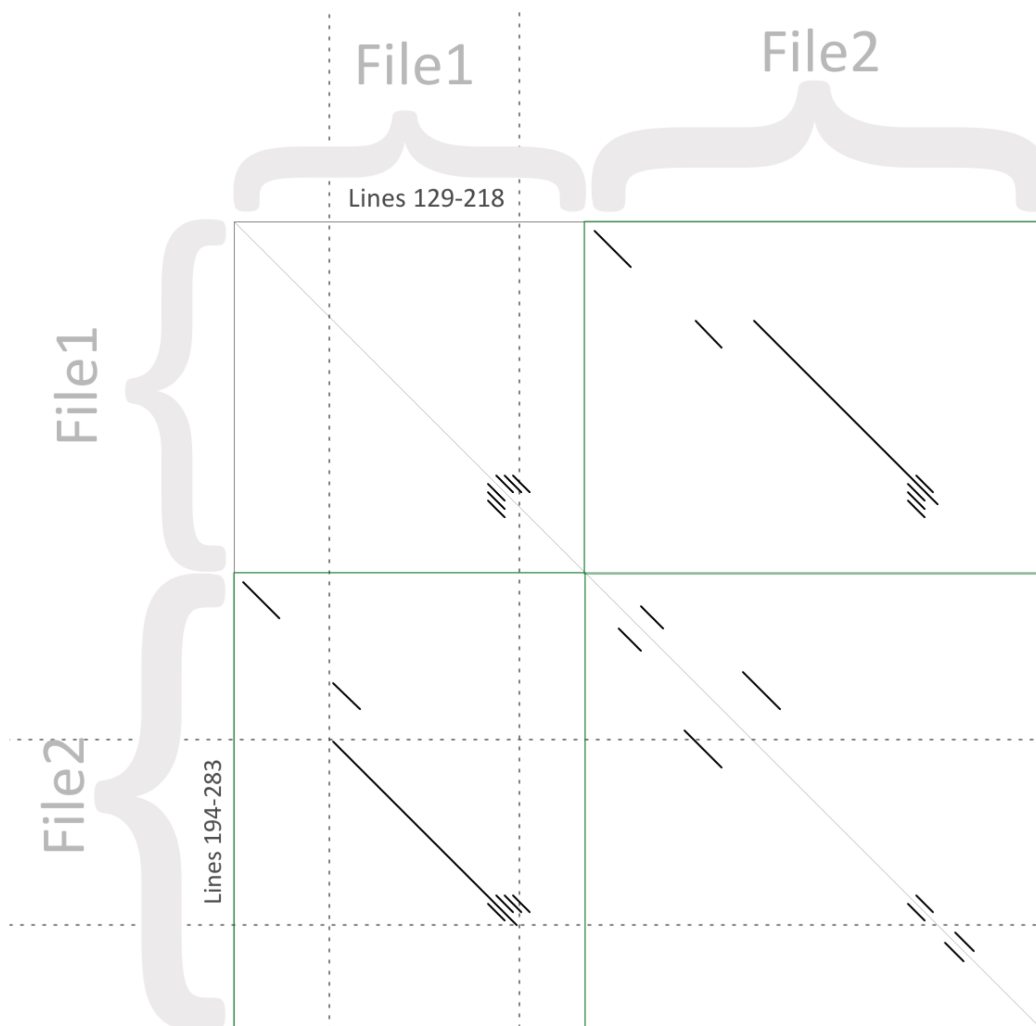

Figure 4.4: A scatter plot of the two files highlighted with green borders from Figure 4.1. The clone pair, from lines 129 to 218 of File1 and from lines 194 to 283 of File2, is marked with dashed line. Since a scatter plot is symmetric, the upper right corner shows the same pattern.

External Duplication within A Branch

In Figure 4.4, the two files from Figure 4.1 are drawn. The files have cloned fragments between them. There is some internal duplication, as can be seen from the short diagonals within the files. Lines from 129 to 218 of File1 form a clone pair with lines from 194 to 283 of File2. Additionally, the files seem to begin similarly, which is indicated by the shorter diagonals in the beginning of the files.

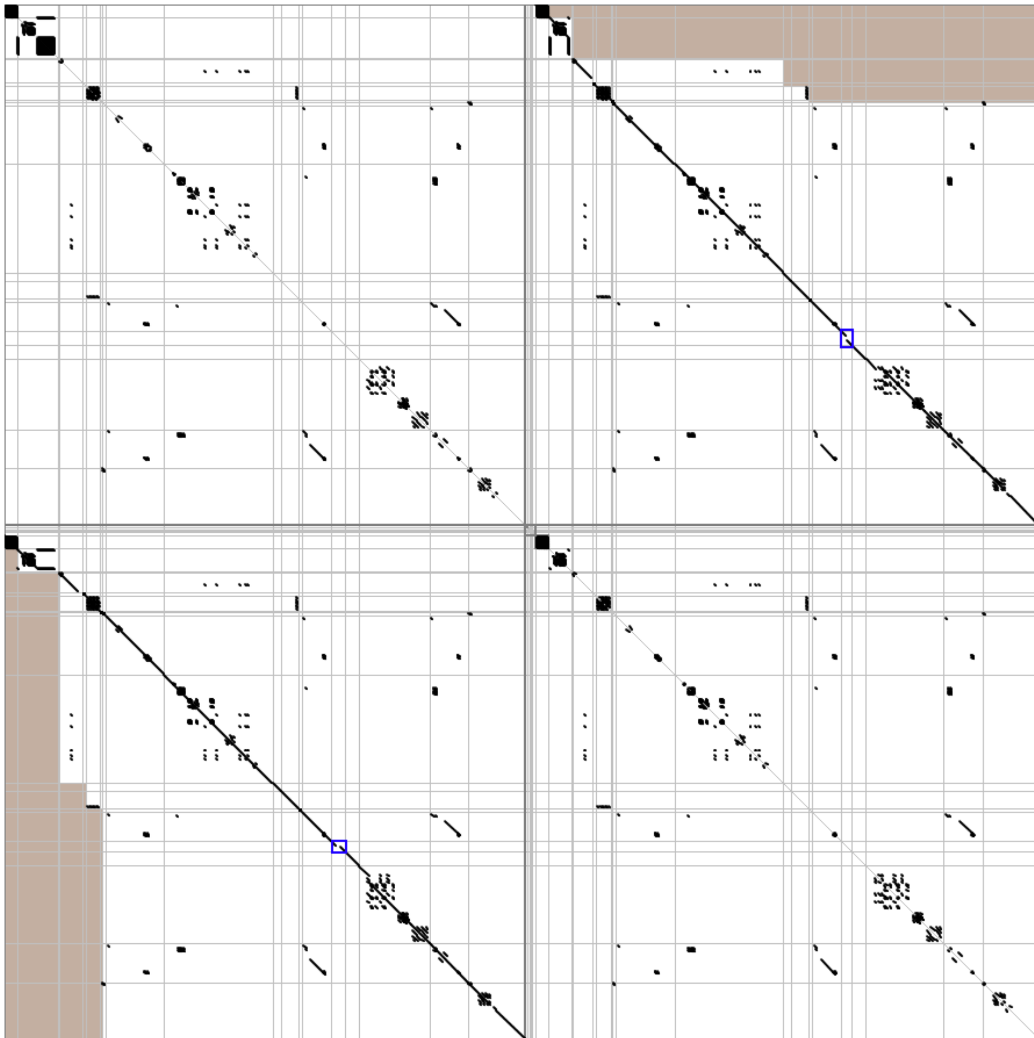

Figure 4.5: A scatter plot overview of two branches of the test application.

External Duplication between Branches
In Figure 4.5 there are two branches. The branch boundaries are drawn with dark gray. Since the branches are forked from the same origin, it is expected that the scatter plot between the branches reminds of the scatter plot of the branch itself. That is, the internal duplication patterns should be seen in the file comparison between branches also. Evidently, branches that have not differed significantly should have long diagonals between their corresponding files. The interesting parts are the broken parts of the diagonals that have the changes between the branches. This kind of file pair is highlighted with blue borders in Figure 4.5. In Figure 4.6 there is the corresponding file pair from the two branches. Here, the files are otherwise similar in the terms of

type II similarity, but the Branch1 has 4 lines (91-94) added between the lines 90 and 91 of the Branch2.

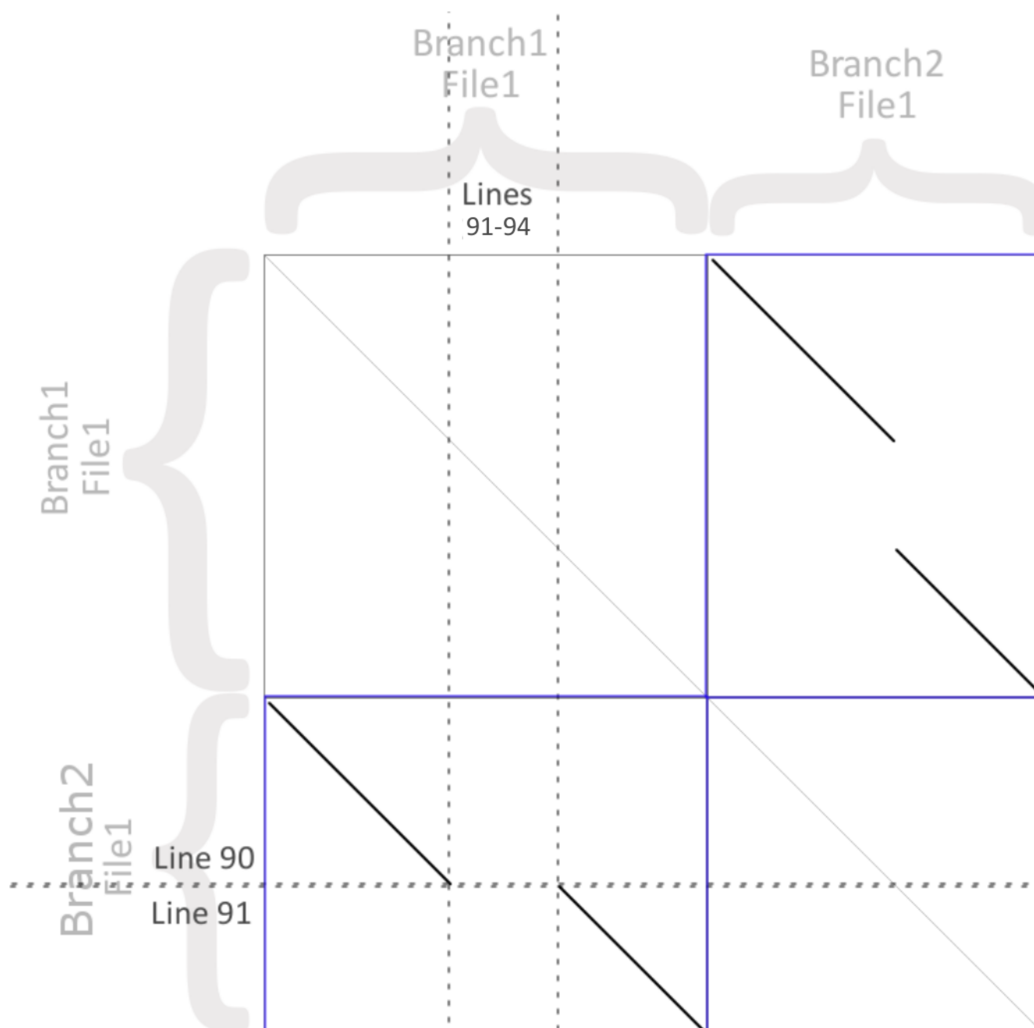

Figure 4.6: A scatter plot of the two files highlighted with blue color from Figure 4.5.

4.3.2 Getting an Overview with the Scatter Plot

Here, the settings used with the tools are the following: GemX has minimum clone length of 50 tokens, minimum TKS of 12 and sharper level 2. These are the also default settings of the tool. Additionally, p-match clones are not searched. Searching of the p-match clones did not seem to effect the results with our test application. That is expected, since the logic between the corre-

sponding variables between branches most likely has not been changed. The minimum similarity length is set to 5 with AtomiQ. This seemed to produce more relevant results in our case studies, yet not presenting too many false positives either. The impact of altering the variables is analyzed in section 4.3.3.

The overview of the scatter plot gives an impression of how similar the different branches are. In Figure 4.7, the front-end code of the test application is visualized. The directories of the different source code branches are drawn with darker border. The thick lighter gray borders represent the subdirectories, and the thinner light gray borders represent files. There are diagonal patterns between the branches 1 and 2, and between the branches 3 and 4. The branch pair consisting of branches 1 and 2 are almost identical. The branches 3 and 4 are similar, but clearly have some differences, since the diagonals between them are broken at some parts. However, the differences between the branch pairs <1, 2> and <3, 4> are significant.

Viewing all the files simultaneously may reveal surprising patterns between branches. For example, a code fragment may appear to be a customization in a file in comparison to other branches. Yet, the same code fragment might be found from the other branches, but in another file. However, fitting all the files to the same overview may be overwhelming. Thus, partitioning the code base to logical units is advisable. Partitioning to front- and back-end files is already been made in consequence of language dependency of the tools. Since we are familiar with the project structure of the test application, we know it consists of three file groups: pages for editing the data, pages for viewing the data, and files that relate to integration with another application. If it is known which files are interesting to compare with one another, zooming to these specified file groups facilitates detecting even small breaks in the scatter plot diagonals.

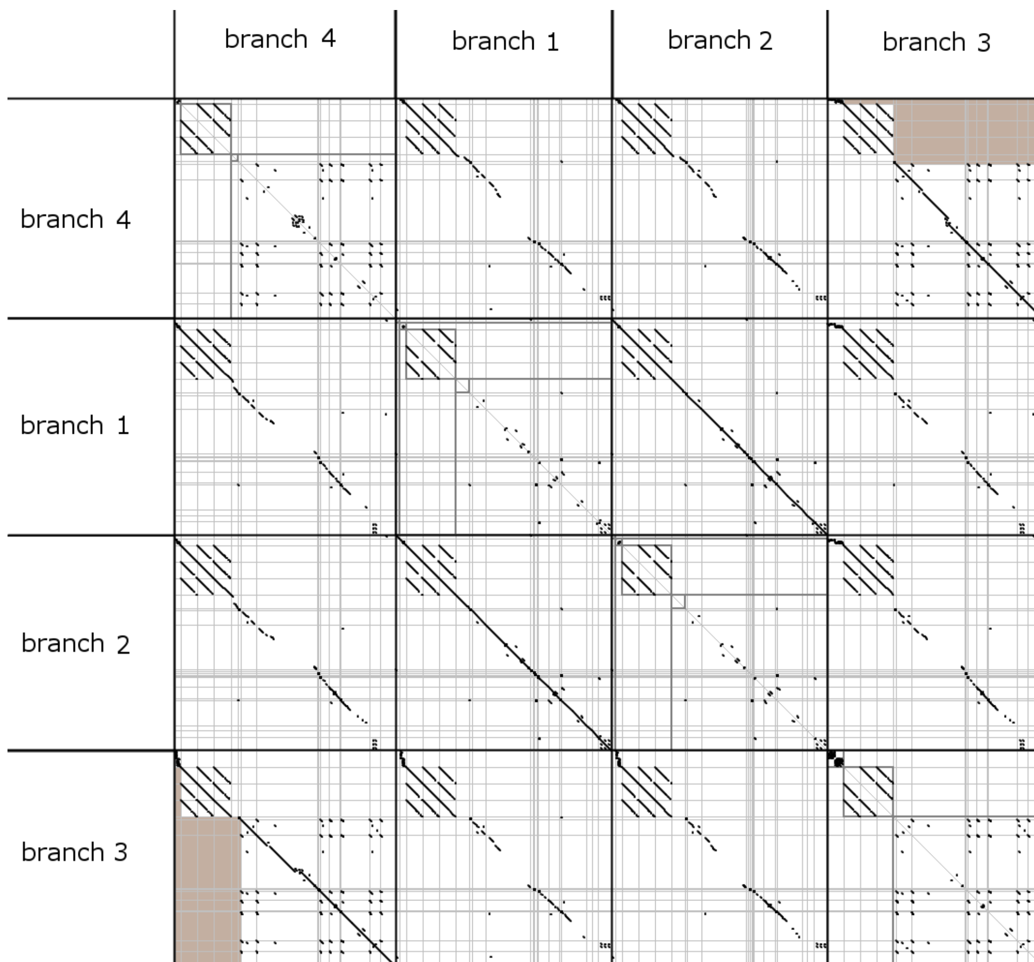

Figure 4.7: The scatter plot visualization with GemX of the front-end code of the test application. The order of the branches in the scatter plot could not be affected. Hence, their order varies between figures.

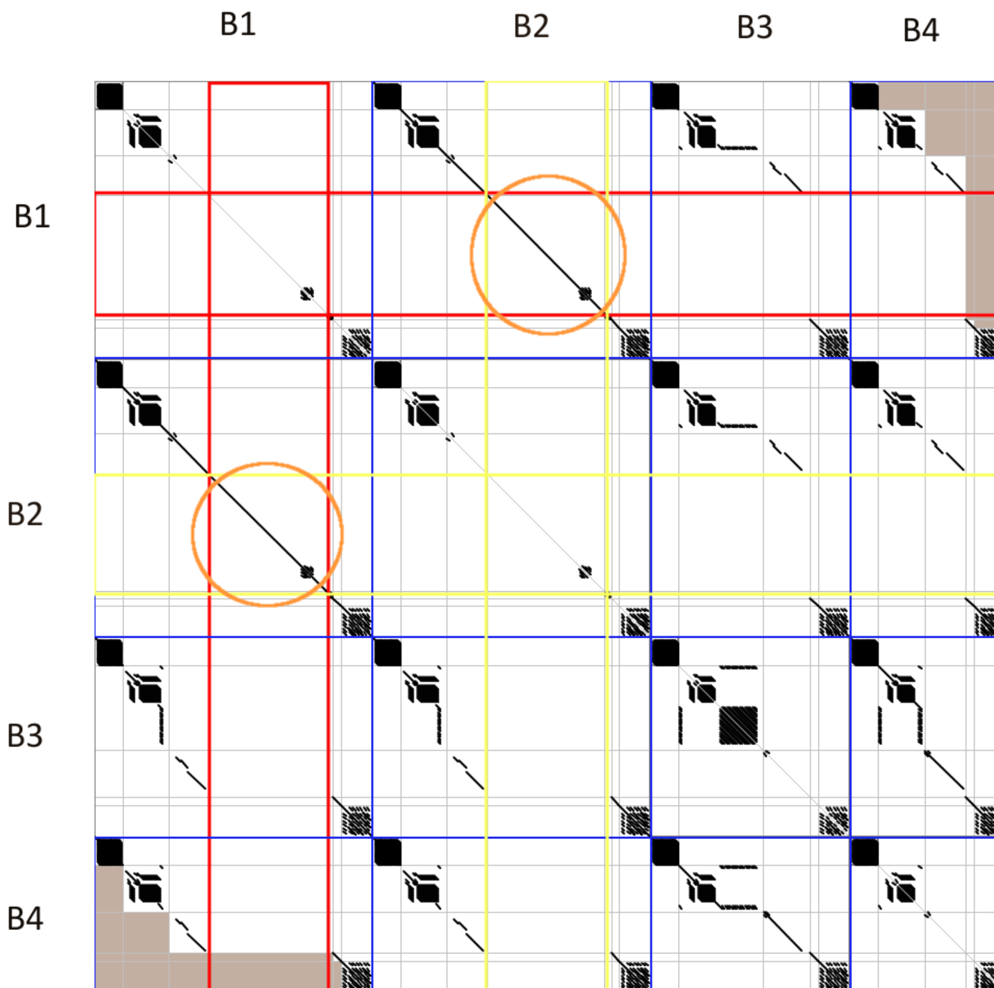

Figure 4.8: Here is a case, where an entire file forms a custom feature. The branches 1 and 2 have the file, branches 3 and 4 do not. The contents of the file is not copied to anywhere else either within this partition. Orange circle frames the contiguous diagonal that indicates the entire files are clones of each other. Red and yellow highlight the cloning results of the file, indicating they are not copied to anywhere else. Branch lines are highlighted with blue color, BX indicates the order of the branches on the axes.

Customizations can be detected as broken diagonals within corresponding files, but there can also be entire files that are not found in some or any other branches. For example, the file highlighted with orange circle can be seen in Figure 4.8. The file is identical between branches 1 and 2, but is not found at all in branches 3 and 4.

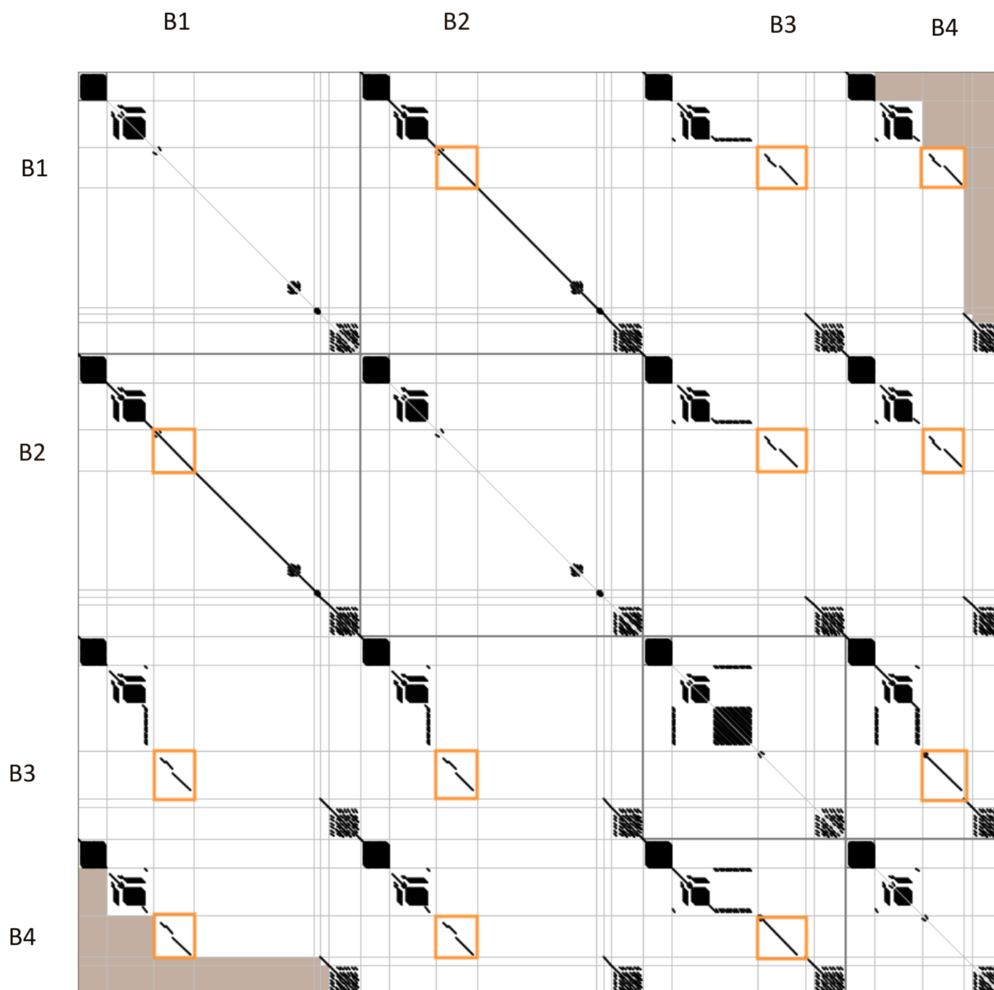

Figure 4.9: The file is a clone with modifications. The same file from each branch is highlighted with orange border. In terms of type II cloning, the file is the same between branches 1 and 2. Between all the other branches there are additions and deletions between the corresponding files. An example code fragment from the highlighted file is presented in Figures 4.10 and 4.11.

Even though there might be custom files in some branches, they are still from the same origin. Thus, the branches potentially still include the same files. That is, if the branches are not diverged fundamentally. In Figure 4.9, a file that is expected to correspond to one another in the different branches, is higlighted in every branch. The file under comparison can be detected to be the same between branches 1 and 2, because the diagonal between the files is not broken at any part. Furthermore, the file is also almost the same

between branches 3 and 4. The diagonal is only broken at the end of the file. However, the diagonals between the branch pairs <1, 2> and <3, 4> are broken at several places. Yet, the base of the file is still essentially the same, since there is a noticeable diagonal through the entire file.

In Figure 4.10, there is a source view from Figure 4.9. The source view shows the file from branches 3 and 1. At this part of the file, the source is identical between branches 3 and 4, and branches 1 and 2. That is why the both sources are entirely highlighted, even though the branch pairs <3, 4> and <1, 2> differ at this part. The GemX source view does not differentiate where the external clone exists, or how many external clones there are. It just tells whether there are external clones or not.

Nevertheless, the source view of AtomiQ shows all the external clones as different blue bars on the left side of the line numbers. Moving the mouse over the bar reveals a tooltip which tells the exact location of the cloned fragment in a file and a directory. In Figure 4.11 is the same source view presented with AtomiQ. The branches are marked in the figure. However, AtomiQ only recognized exact clones, so the type II clone fragment highlighted with turquoise in Figure 4.10 is not recognized as a clone.

Figure 4.10: The source view of GemX does not differentiate where or how many external clones the code fragment has. It only shows whether the fragment has an external clone or not. For example, on the left side, the method starting from line 152 exists only in branches 3 and 4. The gray highlighting of the code shows the code fragment has a clone, and the gray bar on the left side of the line numbers indicate the fragment has an external code. On the right is the source view of branch 1, which does not include the method. The clone class selected is a type II clone with modifications in variable names and literal values.

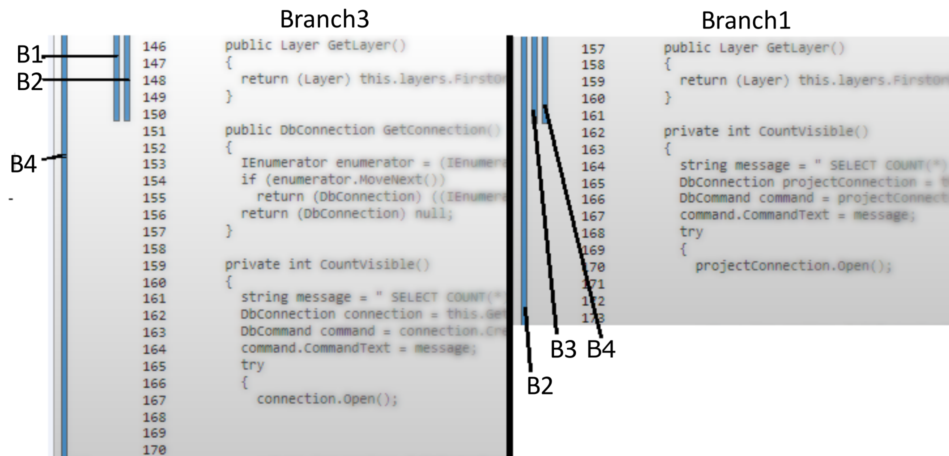

Figure 4.11: The same lines as in Figure 4.10 presented with AtomiQ. The source view of AtomiQ draws a blue bar for each external clone of the code fragment under examination. The mouse overlay on top of the bar reveals a tooltip, which shows in which file the external clone exists. The branches in which the external clones exist are marked in the picture. For example, B1 and B2 correspond branches 1 and 2 respectively. Since AtomiQ only finds exact clones, the method that is a type II clone is not found to be a clone between branches 1 and 3. Notice that AtomiQ starts the line numbers from 0, hence the one line number difference between the source views.

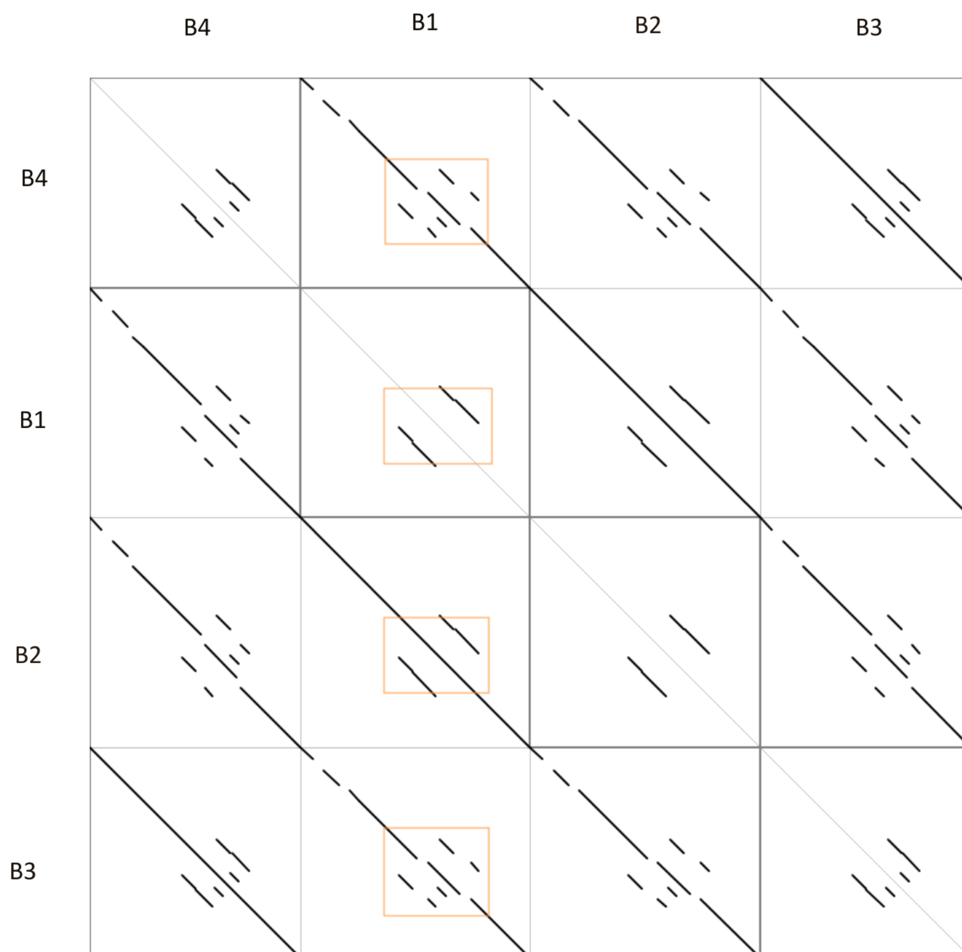

Figure 4.12: A scatter plot of a single file between the four branches B4, B1, B2 and B3. The interesting part here is the long diagonal that has two fractions between branch pairs 1, 2 and 3, 4. That means, the branches 1 and 2 have customizations compared with the branches 3 and 4. The source view is presented in Figure 4.13.

Moreover, the interesting clones can be identified from the source view of AtomiQ, even though there are overlapping clones. The source view of the code fragment highlighted with orange in Figure 4.12 is presented in Figure 4.13. AtomiQ differentiates internal clones with red color. Since the overlapping clones are essentially the same clones as the internal clones, they can be regarded to be irrelevant.

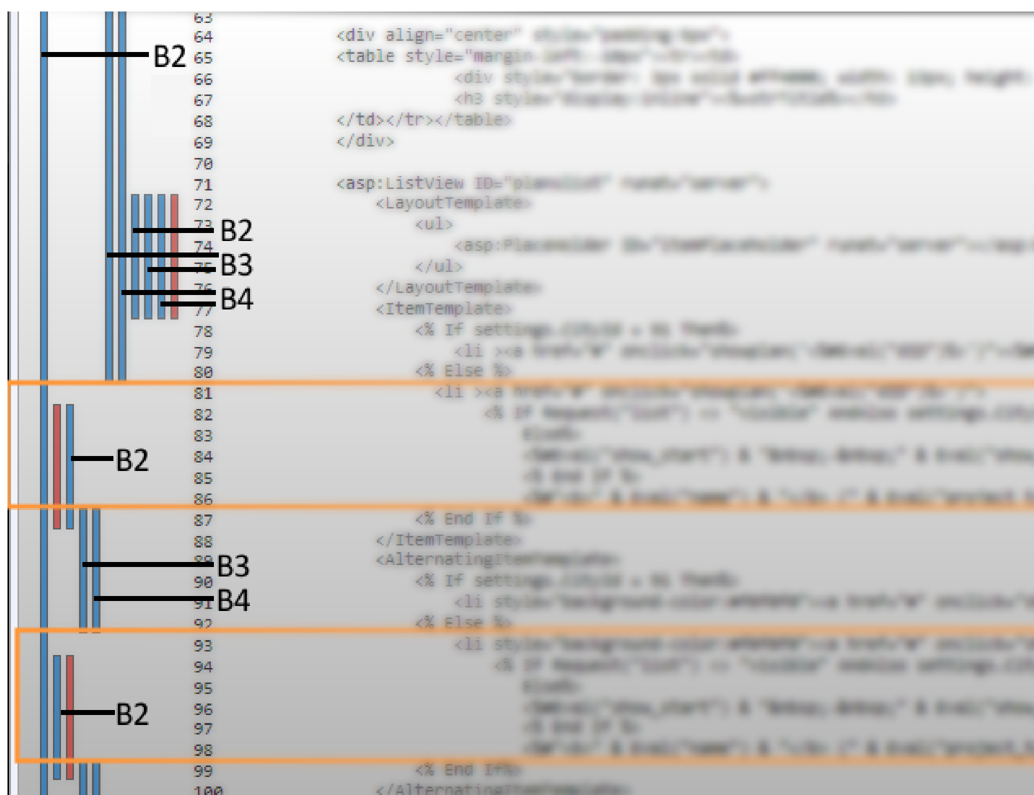

Figure 4.13: The source view of Figure 4.12. The broken parts of the diagonal are highlighted with orange border. The other clones are irrelevant in terms of similarity between branches. However, in this case, the internal duplication is at the same lines as the uninteresting parts, so it is easy to conclude which parts belong to the long diagonals.

However, the source view of AtomiQ can get confusing. The source view can get cluttered because of the number of clones. Additionally, there can be occasional characters within the code that cause AtomiQ to recognize consequent fragments as separate clones. In Figure 4.14 there really is not custom code in branch 1 in comparison to any other branch, even though the bar of the branch 4 breaks between lines 128 and 129. If AtomiQ would draw clones from the same files in the same column, the continuity of the bars would be easier to interpret. Alternatively, different colors for different files could be used.

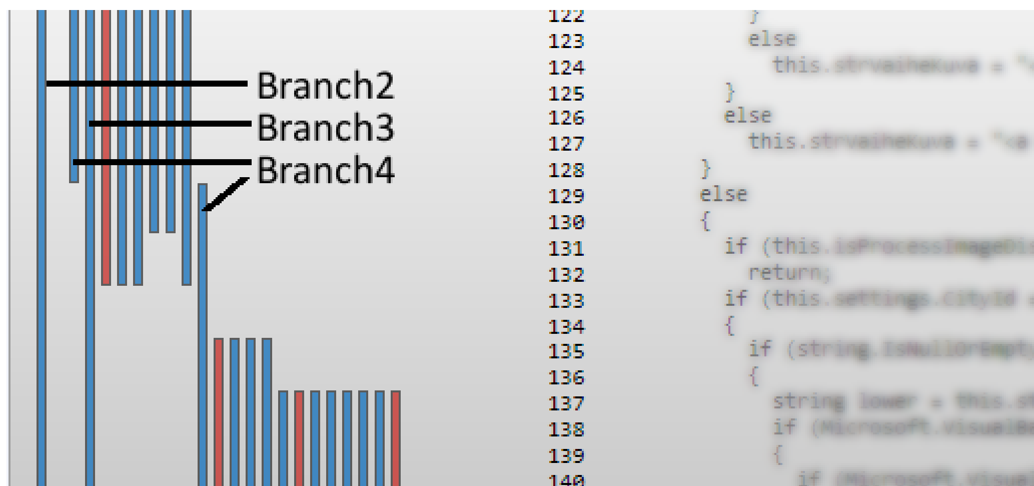

Figure 4.14: Since AtomiQ does not expect such large amounts of duplication, the source view may get confusing. Here, it is again fairly easy to interpret that the externally and internally duplicated code fragments of the same length are uninteresting. However, there are also two blue bars, which do not have a corresponding internal clone, and are not the longest duplicated fragments from the other branches. Furthermore, the long continuous bars are interesting. However, one of the bars is not continuous, even though there is no differentiating line between the bars. Possibly there is some character in the branch 4 that any other branch does not have there, which causes AtomiQ to recognize two clones instead of one.

4.3.3 The Impact of Altering the Threshold Values

The source view can get confusing in two ways. First, there may be too few recognized clones around of the interesting customization in the source code. That is a result of setting the similarity threshold values too high. That means the code fragments that should be interpreted as clones are not found. In other words, false negatives can be distinguished only from the source view. In contrast, if the similarity threshold values are set low, more irrelevant code fragments are interpreted as clones. These can be considered to be false positives in this study.

Figure 4.15: A file in branch 3 has a customization at lines from 90 to 94. The customization is not duplicated to any other branch. On the left is a source view with minimum clone length of 50, and on the right with length 40. As can be seen, the beginning of the function is not recognized as a clone with too high minimum clone length. With lower requirement on clone length, the customization is easy to perceive from the otherwise highlighted source view. The same source lines are presented with AtomiQ in Figure 4.16.

In Figure 4.15, there can be seen that lowering the minimum clone length from 50 to 40 already recognizes the beginning of the function as a clone. Setting the sharper level to 0 had similar impact. Hence, the sharper level seemed to have an affect to the comparison granularity. Granularity free approach is suitable for our case studies, since we are interested in similar files rather than only comparing similar functions or other code blocks. However, lowering the sharper level also increases the size of the false positives. Hence, similarly to lowering any other threshold values, lowering the sharper level may clutter the source view. Additionally, lowering the sharper level only aids detecting the false negatives at the beginning and end of the structural blocks. False negatives in the middle of structural blocks would still not be recognized with lower sharper level.

The same code fragment is in Figure 4.16. Here, the beginning of the function is already recognized as a clone with the default settings chosen in the study. Additionally, AtomiQ is granularity free, so it would have also recognized the fragment with a higher similarity length. However, the lines from 94 to 100 are not recognized. This cannot be fixed with altering any configuration values, since the code fragments are not recognized as clones because the fragment is a type II clone. AtomiQ does not have an option to recognizing other than type I clones.

Figure 4.16: The same source view from Figure 4.15 presented with AtomiQ. Again, there is a difference of one in the numbering of lines between GemX and AtomiQ. Hence, the custom part here are the lines from 89 to 93. However, AtomiQ does not recognize the lines from 94 to 100 as clones since there are type II changes in variable names.

Nonetheless, the results of AtomiQ can also be impacted by altering the chosen similarity length. In Figure 4.17, there is a source view of the branch 1. There are modifications in the consecutive variable initializations. These could be considered as false positives by some clone detection tools, but in this study, anything that reveals differences between branches are not interpreted as false positives. Here, branches 1 and 2 are the same considering the code fragment under comparison. There are no code fragments in branches 3 and 4 that would have 10 consecutive duplicated lines in the branch 1. However, there are 5 consecutive duplicated lines. The code fragments less than 5 lines would not be recognized.

Figure 4.17: Since AtomiQ seeks only exact clones, names of the variables are not normalized. On the left, a threshold value of 10 is used for similarity length. On the right, the threshold value is set to 5. As can be seen, the threshold value has a great impact on whether false negatives are introduced.

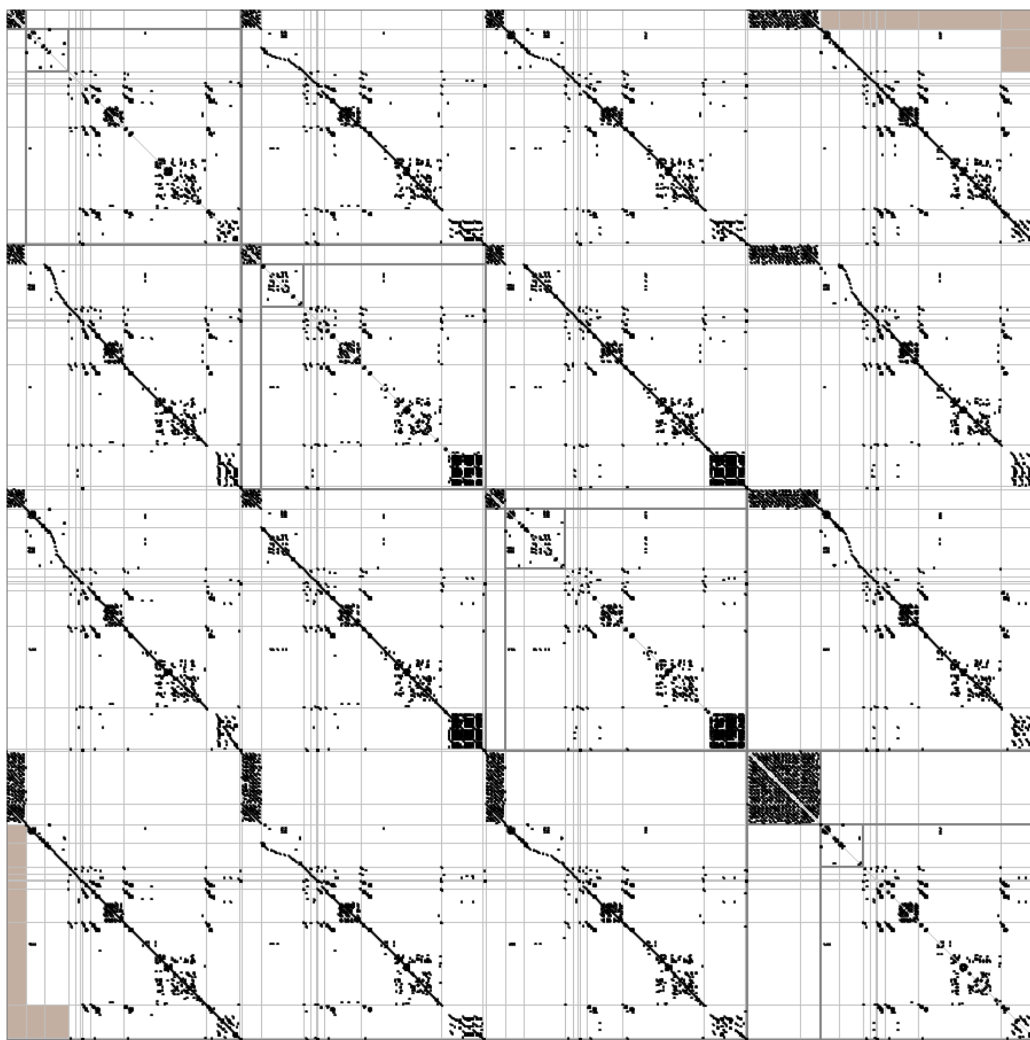

Figure 4.18: Low threshold value reveals uninteresting noise in the scatter plot. In other words, too low threshold values may introduce false positives. The scatter plot is drawn with minimum clone length 10 and minimum TKS 5.

The downside of lowering the threshold values can be seen in Figure 4.18. Searching for too small code clones results in cluttered scatter plot view. In this study, the uninteresting clones that would hide the interesting customizations are considered false positives. However, lowering the threshold may reveal also interesting patterns. In Figure 4.19 there is a scatter plot between two branches drawn with the default settings. The diagonals in two files are scattered radically in the places where they are circled with orange. When the same view is examined with minimum clone length of 15 and min-

imum token sequence of 6, it can be seen that the diagonal is there. Some empirical testing of the values was required in order to reveal the diagonal. It is useful to test different values in order to see whether the code base really is altered radically or are there just some different lines here and there, which cut the diagonal.

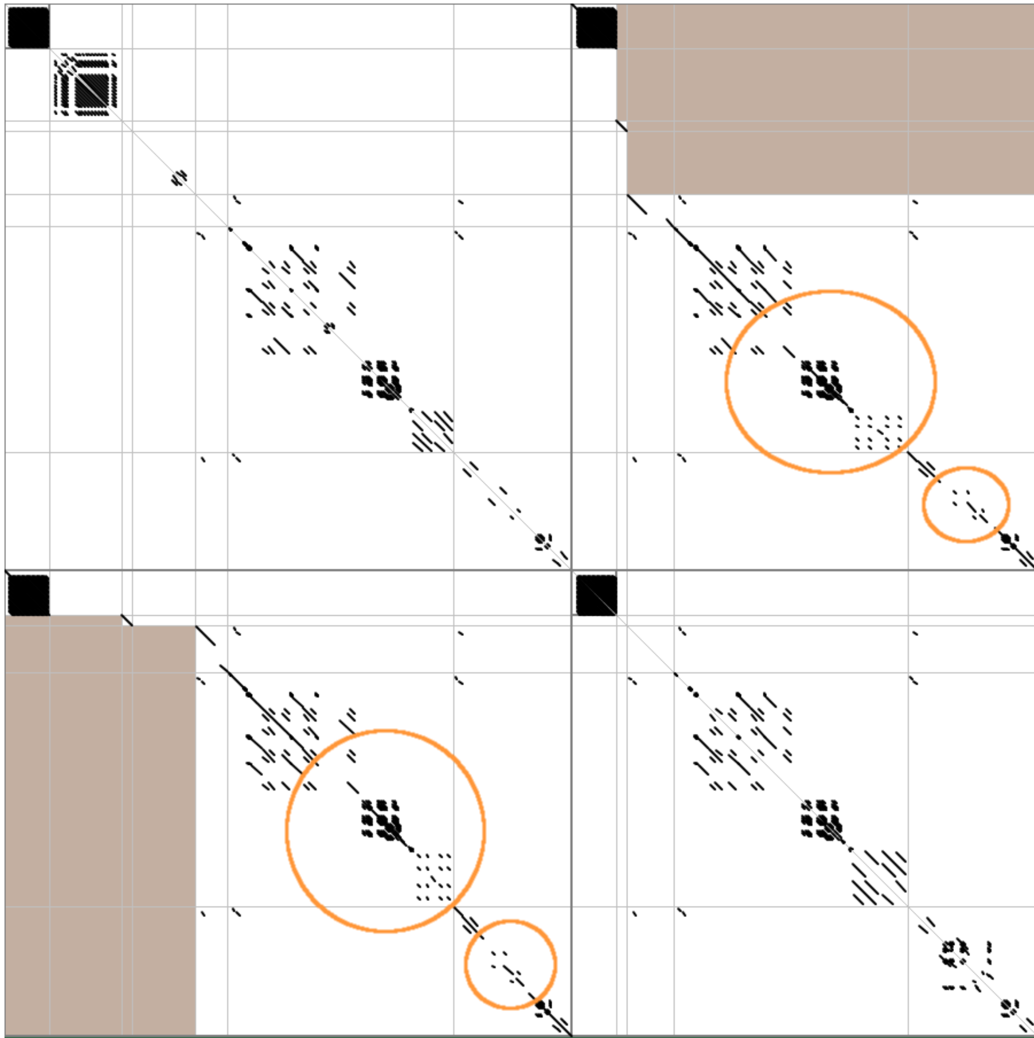

Figure 4.19: Changing the threshold value may also help to see hidden patterns. Here is a scatter plot of two branches, drawn with configuration parameters of minimum clone length 50 and minimum TKS 12. In Figure 4.20 are the same files and branches, drawn with lower threshold values.

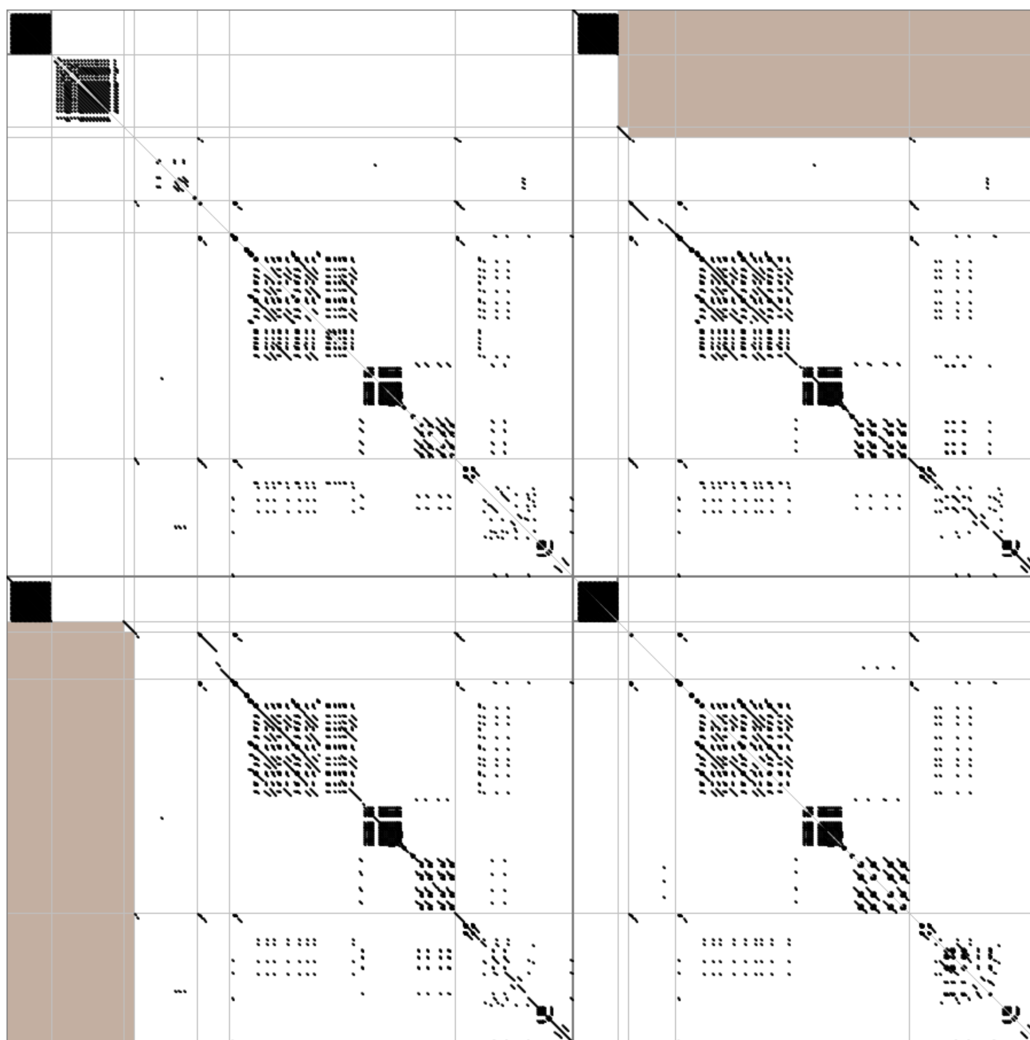

Figure 4.20: Reducing the minimum clone length and TKS may reveal a fractioned diagonal. Here are the same files of the same branches as in the Figure 4.19, only drawn with threshold values of minimum clone length of 15 and minimum TKS 6.

The remaining question is, can one accidentally hide customizations by altering the threshold values too low or too high. Customizations are not completely hidden as is shown in Figures 4.21 and 4.22, even when the values are set so low that everything else is recognized as a clone. However, especially the source view of AtomiQ has become cluttered with the small clone fragments. Setting the value too high increases the need for manual checking of the code.

```
158          Response.
159        End If%>
160      <% If strPlanNumber
161      <%= Request(
162    </p>
163
164    <p>
165      <%= strProject
166    </p>
167
168    <% If Not String.
169          Response.Write(
170          Response.Write(
171          Response.Write(
172      End If
173    %>
174
175    <%
176      If Not String.
177          Response.Write(
178          Response.Write(
179      End If
180    %>
181
182    <% If bToteutuksessa
183          If Not String.
184              Response.Write(
185              Response.Write(
186              Response.Write(
187          ElseIf Not String.
188              Response.Write(
189              Response.Write(
190              Response.Write(
191          End If
192      End If
193    %>
194
195    <%
196      If Not String.
197          Response.Write(
```

Figure 4.21: Lowering the threshold values do not usually hide the entire customization. Here is a source view drawn with threshold values minimum clone length 10 and minimum TKS 5. As can be seen from the bars on top of the line numbers and on the left side of it, the line 176 is altogether neither recognized as internal nor external clone.

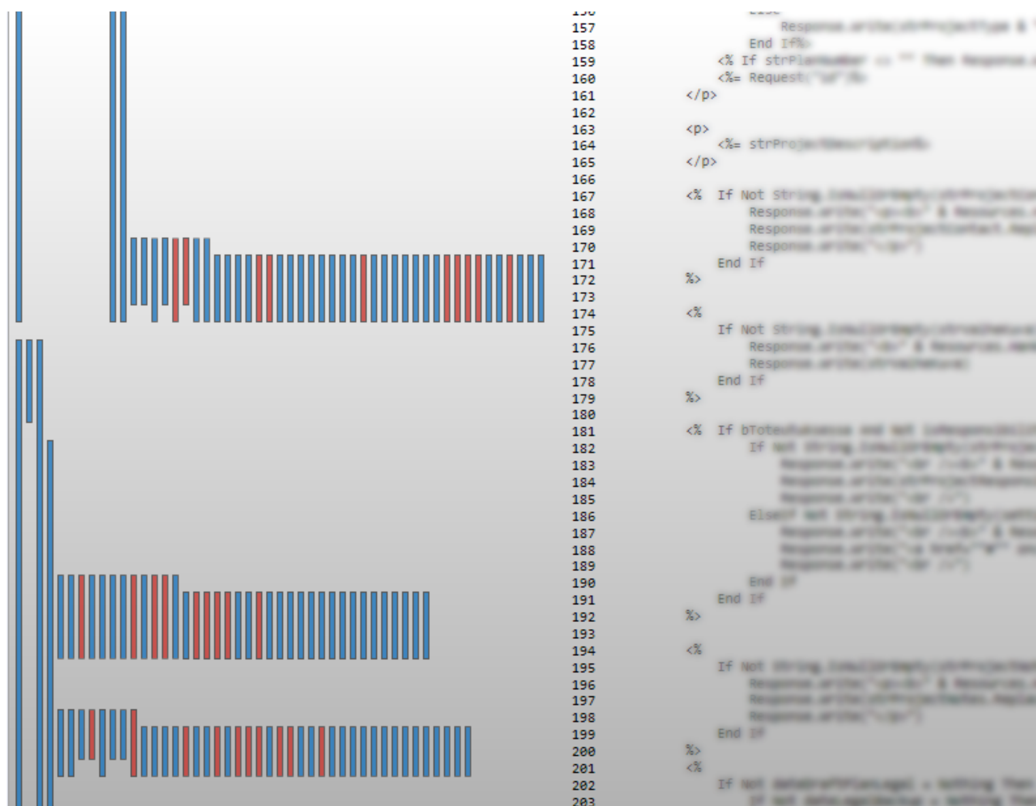

Figure 4.22: Here is the same source from Figure 4.21 drawn with AtomiQ. The threshold value for minimum length is set to 2 lines. Again, the line with the customization, here line 175, produces a gap in the bars that indicate external cloning. However, the low similarity length value produces a lot of uninteresting clones, as can be seen from the large number of short bars on the left side of the line numbers.

4.4 Results

4.4.1 Processing of the Found Customizations

Each located customization needs to be processed somehow. What to do with the customization depends on its nature. For example, the customization can be:

- a bugfix
- an improvement
- a feature

- functionally the same thing as in the other branches
- or the code fragment may have varied so much that it does not have much to do with other ones anymore.

Bugfixes and improvements should be determined whether they are still relevant in the productized version. Since newer technologies are used in the productized version, the code base is partially rewritten. Yet, if the existing code fragment is used in the product, bugfixes and improvements should naturally be included.

A feature requires specification whether it would also be useful to other clients. If the feature increases the value of the product, and does not conflict with any other functionality, it could be decided to be included in the productized version.

However, if the feature conflicts with other functionalities or wishes of clients, it may not be possible to make it a part of the standard product. It could still be included in the product as configurable feature, where a setting would define whether to use this custom logic or not.

A code fragment that is not recognized a clone can still be essentially the same. It can be functionally the same. Type IV clones are not recognized with GemX. Also, there could be statements that do not have an effect on the functionality of the program. For example, there could be logging or debug messages between lines. If the statements are in between the cloned lines, the clone fragments may be too short for the clone detection tool to notice.

The result can also be that the files between branches truly differ significantly. Then, the searching of customizations is no longer worthwhile, but to study the differences otherwise. For example, if a large part of the program has already been transformed so much that the branch does not remind the other branches anymore, separating the code base for that client could have been truly reasonable development choice and have resulted to an entirely different application.

In summary, each customization needs to be either included in the product as a standard or a configurable part of the program, or disregarded altogether. An expert opinion is needed in order to make these kind of decisions. An expert in this case is someone who knows the code base and/or can interpret the customizations in it, and has the ability to make the decisions on how

to proceed. The interpretation of the customized code and the decision on what to do with it is outside the scope of this thesis. The process developed here aims to provide the background information needed in order to make these kind of decisions.

4.4.2 Evaluation

Process
The overview with the scatter plot helps to get started with the productization process. As we discussed among colleagues, the overview gives a sense of control over the situation. At first glance, it gives an impression of which branches are most alike. It can be used to estimate the workload of how much there is to be done, in order to productize the application entirely. Even minor changes may feel insurmountable and frustrating if you do not have an overall picture of how much work there really is to be done, in other words, exactly how many "minor changes" there really are.

The overview itself can already aid to decide whether to productize or not. Sometimes it is more cost-effective to not productize. Then the method can be used to only document differences. The situation can be under control even if the code base is not productized.

The analysis presented here is explorative by nature. The evaluation of its accuracy would require further research. The precision of the results could be evaluated by comparing, how many of the detected clones are true positives. A true positive in the research would be a code fragment that was desired to be found as a clone. Thus, the proportion of the found true positives in comparison to all found clones in the system would define the precision of the results. Respectively, the recall of the results could be evaluated by comparing the proportion of the found true positives in comparison to all clones in the system. (Bellon et al., 2007)

In order to evaluate such, the set of all clones in the system would should be determined. In addition, all the customizations would need to be known or be able to be detected by some other method in order to validate that how many of them were really found. Such quantitative analysis is outside the scope of this thesis.

Tools
Since none of the tools were optimal for this kind of research we needed to combine the results of two different tools. An optimal tool for this purpose

would contain everything that is needed. For example, an overview with a scatter plot and a source view that would show in which files and branches (directories) the external clones are. Additionally, the possibility to view files simultaneously could be convenient. Since AtomiQ showed only type I clones, and GemX also type II, both source views were needed in order to navigate to the interesting differences. If the source view was from the same tool, it would naturally show the same results as the scatter plot. In this research, the most interesting clones are the ones corresponding to the same lines of the same file in another branch. Hence, separating these clones from the other clones would facilitate the interpretation of the visualizations. The separation could be done by visualizing the clones differently from others, or by adding user-interaction properties.

AtomiQ visualizes all external clones with blue color. In this study, each branch was located in a separate directory. Thus, coloring external clones in different directories with another color would clarify the results. Ordering the bars according to the location and name of the file could also facilitate the interpretation. Additionally, continuing the bar in the same column would help, if there is no overlap between the clones. Alternatively, possibility to hide the uninteresting clones from the source view would clarify the situation.

The tools do not really detect type III clones but type I and II. They can be interpreted as type III clones if consequent clones are from same file. Hiding shorter clones also hides the short parts of the long gapped clones. Hence, hiding clones based on length is not an option for screening out uninteresting clones. Hiding internal clones from the scatter plot is an option with CCFinderX, but it potentially complicates the identification of which clone are relevant, and which are not. Clones can also be filtered out based on different metrics with GemX. However, extensive filtering may hide interesting patterns. One option would be to develop a method that would conclude, which of the clones are relevant in this type of research. For example, visualizing only the clone that can be interpreted as longest possible type III clone between the corresponding files between the branches. Ueda et al. (2002b) proposed filtering out non-gapped clones with their technique gap-and-clone scatter plot. This idea could be adopted to recognizing the external clones between branches.

4.4.3 Proposition of the Process

The code clone visualizations can be utilized to get an overview of the similarity between branches. Additionally it can be used to navigate to the dif-

ferences in the source. Getting an overview of the overall situation and being able to browse through the differences can expedite the process of productizing. We adapt the visualization mantra of Shneiderman (1996 Shneiderman [1996]), and define the steps for the process of using code clones to detect differences between customer-specific branches, as follows:

Step 1: Get an overview.
The scatter plot represents cloned fragments as diagonals. The overview of the situation shows which branches are similar with each other. If there are large gaps in the diagonals between branches, decreasing the threshold values for clone similarity may reveal hidden patterns. On the other hand, if the overview is cluttered, increasing the values may reduce noise and ease the interpretation of the scatter plot. The overview can be used to search unexpected patterns in the code base. For example, a code fragment may be cloned to a different file in another branch. Moreover, the overview may reveal that different features are clones of each other. These features could be abstracted as the same feature in the product.

Step 2. Partition the code base.
The overview can be overwhelming, especially with large code bases. Therefore, partitioning the code base to logical units is advisable. Exploring fewer files simultaneously eases the task. The partitioning of the code base can be done based on the results of the overview. For example, if the cloning is restricted within certain directories between the branches, these directories can be partitioned to one logical unit. On the other hand, an expert advise can be used to partition the code base. An expert can be someone who knows the source code, and can tell which components are interesting to overview simultaneously.

Step 3. Zoom in on details.
The broken diagonals reveal the lines which include customizations in the code. These are the details on which you want to zoom in. The source view of AtomiQ can also be used for traditional clone detection. The source view of a feature under examination shows where else the code fragment has been cloned. This technique can be used to find out, which branches include a particular feature. However, since the examination is done with AtomiQ, only exact clones are found. Furthermore, the extent of the customization is not always accurate. The code fragments before and after the customization are not necessarily highlighted as part of the preceding and following clones, even though they would be. With GemX, by setting the threshold values too high, the short clones in the middle of uncloned parts may be detected as

false negatives. AtomiQ on the other hand only detects type I clones, so it highlights less code as clone as the GemX as does. Increasing the threshold values for clone similarity may narrow down the uninteresting clone candidates. In consequence, it may clarify the source view.

Step 4. Suggest a course of action.
Nevertheless, the exact extent of the customization is not usually relevant. The reason for the customization can usually be interpreted, even though some irrelevant lines of code are identified as part of the customized part. However, this requires a certain level of expertise. The expertise can be familiarity with the code base, or general ability to understand the code at hand. Considering the scope of this thesis, the priority is to be able to find the customizations in the first place. Nonetheless, after a customization is found, its existence needs to be documented. Additionally, the decision of whether and how to include it in the product, needs to be made somewhere along the line.

Chapter 5

Conclusions

The original problem was that there was a need for *a systematic way to productize software code that has diverged to customer-specific tailored branches.* Since the specifications and implementations of the product versions of the both case studies have already been started, we only need to find out the customizations between branches. Each customization may be a factor that increases the value of the product from the perspective of the client. Hence, the customizations cannot just be discarded in a version upgrade. Thus, in order to be able to switch a customer-specific code base to the product version of the code base, it must include all the custom features the client previously had, or removal of these features needs to be justified. In any case, one must be aware of all the customizations in the code base. The purpose of this thesis was to explore, whether code clones could be utilized in the situation.

The traditional way to approach this sort of situtation is to compare the branches manually with a file comparison tool. If the code bases are relatively small, or do not differ significantly, the manual comparison is a feasible option. However, if the code bases are large, the task may prove excessive and there may not be resources for the job. For example, manual comparison of our larger case study would mean browsing 200K LOC in total, comparing two files with each other at a time.

We defined two research questions to narrow down the problem:

Are there techniques to evaluate code similarity, when the code has been modified?
File comparison tools rather show more insignificant changes than accidentaly ignore a significant change. In situations similar to our case studies, the

tools will highlight a lot of uninteresting differences from the productization perspective. In addition to the custom changes made for each client, the standard code base may have mutated also. For example, variable names may have changed due to refactoring actions. Furthermore, there is bound to be changes in whitespace, comments and layout in general, since the code bases have been developed separately for years. As an answer to the research question: code clones can be utilized in this kind of situation to fade out the uninteresting differences between branches. Then, the focus may be shifted to the more significant customizations.

Are there any tools that would aid comprehending undocumented legacy software?
The problem with the case studies was that it was not known, how much the branches differ. File comparison tools may provide the information whether files are the same or different, or how many lines between files differ, but they do not provide information about how similar they are. A code clone visualization, the scatter plot, provides an overview that aids to form a mental model of the similarity situation between the branches. This overview itself already can aid to estimate the workload of productization. The amount of customizations correlate with the amount of work. If the customizations are too extensive, there will not be much similarities between files anymore. In that case, the process described in this thesis is not applicable. However, whether there is enough similarity in order to use this method, can be easily seen from the overview produced by the scatter plot. If the branches are too far apart, code clones can be still used in more traditional way to check whether a code fragment has been copied to somewhere else.

The output of the thesis is a process proposal, of how to utilize the code clone visualizations. The process is applicable in a situation, where the following conditions exist:

- there are several code bases, which are conceptually the same product
- the code bases are similar enough, so that the general corresponding components can be located between the branches
- the comparison of the code bases with file comparison tools only would require so much manual work that it would not be cost-effective.

Adapting the visualization mantra of Shneiderman (1996), we defined the following steps for the proposal of the process:

1. *Get an overview.* The overview can aid to form a mental model of the similarity situation. The overview can also reveal surprising patterns in the code base, if code fragments have been copied to a different location in another branch or unexpected components are alike. However, the process has been only tested with a relatively small code base (approximately 3K LOC times 4 branches). Increased size of the code base under analysis will at some point result in cluttering of the overview, which emphasizes the importance of step 2.

2. *Partition the code base.* Partitioning the code base to logical units can be done based on knowledge of the code base, or it can be interpreted from the overview. Suitable units can be, for example, known components of the code base, or files included in a directory. The units for comparison can be interpreted from the overview by searching units, which have clones between each other, but little cloning relations with other units. For example, if the clones of the code fragments between branches are within the corresponding directories, partitioning based on directories could be used.

3. *Zoom in on details.* The source view can be used to interpret the nature of the customization. In addition, the source view of AtomiQ can be used to examine from which other branches the code fragment under inspection is found.

4. *Suggest a course of action.* After a customization is found, it must be handled somehow. The nature of the customization determines how it needs to be processed. At least, the existence of the customization should be documented. In addition, a proposal whether to include it in the product or not could be presented. Nonetheless, the analysis of the nature of the customizations and making the architectural decisions are outside the scope of this thesis.

Since the nature of this thesis was explorative, the feasibility of the process would require further study. The future research could be an interview of people encountering a similar situation. For example, do they feel that they get a better mental model of the similarity situation with the process. Or, do they find interpretation of the customization results easy or confusing.

Another aspect of future study would be to technically optimize tools for this kind of situation. We needed to use two different tools to get the information we wanted. However, different clone detection tools find different results, so the interpretation of the source view required extra effort. An

optimal tool would have included both a suitable visualization for overview and a source view. The scatter plot visualization served the purpose of the overview in this study. The source view required showing the locations of the external clones in order to interpret which of them were relevant. Optimizations for the visualization techniques would ease the interpretation.

The key question here is, how the *external clones between the branches* should be visualized. Technically, this could be done by visualizing external clones from separate directories differently. Adapting the idea from AtomiQ source view, the clones from corresponding files from different branches could be visualized clearly. For example, AtomiQ could be modified to assign different color for clones in different directories. Additionally, it is important that the source view can visualize the cloning situation of the fragment in comparison to every other branch simultaneously. Source view for visualizing only the comparison of two files is not suffiecient for this purpose.

Furthermore, user-interaction options for filtering the results could be added. GemX offers filtering by different metrics, but AtomiQ does not. Clearing the uninteresting clones manually from the source view would ease the task of analyzing the customizations. Other option would be to define programmatically, which clones are interesting from this perspective. For example, the internal clones, and the external clones corresponding to the internal clones between branches, could be considered as false positives. Additionally, the longest possible type III clones between files could be highlighted in the other clones. Ueda et al. (2002 Ueda et al. [2002b]) presented this idea to filter out non-gapped clones with their technique gap-and-clone scatter plot. However, each pattern may reveal something of the code base. Hence, extensive filtering of the results may potentially hide something interesting.

The tools used in this study did not search type III directly, even though the visualization of two code fragments within the same file would seem like a gapped clone. More accurate results could have been obtained if they did, since the short cloned statements between added or modified statements would have been recognized to be a part of the larger clone, instead of being detected as false negatives. Hence, the idea of detecting type III clones could be adapted, yet visualizing the results without hiding the gaps of type III clones. For example, the scatter plot could not draw a continuous diagonal through the entire type III clone, because the customizations (broken diagonals) would be then hidden.

However, plotting the type II clones also hid some potential customizations.

Adjusting the options of which kind of type II clones to search would be relevant. Normalizing identifier values might hide some hardcoded variable assigments, which are also one kind of customizations. Hence, extracting the customizations the way defined in this process is not solely sufficient method for determining the differences between branches. All things considered, the purpose of the process is to *expedite the process of productization*, not to make interpretations, decisions or testing on behalf of programmers or product owners.

Bibliography

Peter Artz, Inge van de Weerd, Sjaak Brinkkemper, and Joost Fieggen. Productization: transforming from developing customer-specific software to product software. In *International Conference of Software Business*, pages 90–102. Springer, 2010.

Muhammad Asaduzzaman. *Visualization and analysis of software clones.* PhD thesis, University of Saskatchewan Saskatoon, 2012.

Tool AtomiQ. AtomiQ code similarity finder. Last accessed october 2017. `http://www.getatomiq.com/`.

Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.

Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9), 2007.

Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. Function clone detection in web applications: a semiautomated approach. *J. Web Eng.*, 3 (1):3–21, 2004.

Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, 2(2):153–174, 1993.

Johnatan A de Oliveira, Eduardo M Fernandes, and Eduardo Figueiredo. Evaluation of duplicated code detection tools in cross-project context. In *Proceedings of the 3rd Workshop on Software Visualization, Evolution, and Maintenance*, pages 49–56, 2015.

Giuseppe A Di Lucca, Massimiliano Di Penta, and Anna Rita Fasolino. An approach to identify duplicated web pages. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 481–486. IEEE, 2002.

Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE, 1999.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the design of existing code, 1999.

Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 321–330. IEEE, 2008.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Software pioneers*, pages 701–717. Springer, 2002.

Tool CCFinderX & GemX. The archive of CCFinder official site. Last accessed october 2017. `http://www.ccfinder.net/ccfinderxos.html`, 2008.

Michael W Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

Benedikt Hauptmann, Veronika Bauer, and Maximilian Junker. Using edge bundle views for clone visualization. In *Proceedings of the 6th International Workshop on Software Clones*, pages 86–87. IEEE Press, 2012.

Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

Zhen Ming Jiang, Ahmed E Hassan, and Richard C Holt. Visualizing clone cohesion and coupling. In *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, pages 467–476. IEEE, 2006.

J Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 171–183. IBM Press, 1993.

J Howard Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, page 16. IBM Press, 1996.

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. A token-based code clone detection tool-ccfinder and its empirical evaluation. *Techinal report, Osaka University, Department of Information and Computer Scineces, Inoue Laboratory*, 2000.

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

Cory Kapser and Michael W Godfrey. " cloning considered harmful" considered harmful. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 19–28. IEEE, 2006.

Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 83–92. IEEE, 2004.

Walid Koleilat and Niv Shaft. Extracting executable skeletons. 2007.

Raghavan Komondoor and Susan Horwitz. Tool demonstration: Finding duplicated code using program dependences. In *European Symposium on Programming*, pages 383–386. Springer, 2001.

Jens Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309. IEEE, 2001.

Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *icsm*, volume 96, page 244, 1996.

Foyzur Rahman, Christian Bird, and Premkumar Devanbu. Clones: What is that smell? 17(4-5):503–530, 2012.

Damith C Rajapakse and Stan Jarzabek. An investigation of cloning in web applications. In *International Conference on Web Engineering*, pages 252–262. Springer, 2005.

Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 100–109. IEEE, 2004.

Chanchal K Roy. Detection and analysis of near-miss software clones. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 447–450. IEEE, 2009.

Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. 541(115):64–68, 2007.

Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE, 1996.

Jeffrey Svajlenko and Chanchal K Roy. Evaluating modern clone detection tools. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 321–330. IEEE, 2014.

Robert Tairas, Jeff Gray, and Ira Baxter. Visualization of clone detection results. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 50–54. ACM, 2006.

Michael Toomim, Andrew Begel, and Susan L Graham. Managing duplicated code with linked editing. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 173–180. IEEE, 2004.

Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 403–414. IEEE Press, 2015.

Yasushi Ueda, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Code clone analysis tool. In *Proc. International Symposium on Empirical Software Engineering (ISESE 2002)*, 2002a.

Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *Software Engineering Conference, 2002. Ninth Asia-Pacific*, pages 327–336. IEEE, 2002b.

Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 67–76. IEEE, 2002c.

Vera Wahler, Dietmar Seipel, J Wolff, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 128–135. IEEE, 2004.

Jesse Yli-Huumo et al. The role of technical debt in software development. *Acta Universitatis Lappeenrantaensis*, 2017.

Yali Zhang, Hamid Abdul Basit, Stan Jarzabek, Dang Anh, and Melvin Low. Query-based filtering and graphical view generation for clone analysis. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 376–385. IEEE, 2008.

Minhaz F Zibran. Analysis and visualization for clone refactoring. In *Software Clones (IWSC), 2015 IEEE 9th International Workshop on*, pages 47–48. IEEE, 2015.