Improving Software Maintainability through automatic refactoring of Code Clones

***Abstract*—Duplication in source code is often seen as one of the most harmful types of technical debt as it increases the size of the codebase and creates implicit dependencies between fragments of code. To remove such anti-patterns, the codebase should be refactored. Many tools aid in the de©tection process of such duplication problems but do not determine whether refactoring an identified clone would improve the maintainability.**

We address this shortcoming by first analyzing what preconditions apply to be able to refactor clones. We then propose a tool to detect clones, analyze their context and automatically refactor a subset of them. We use a set of established metrics to determine the impact of the applied refactorings on the maintainability of the system. Based on these results, one could decide which clones would improve system design if refactored. We evaluate our approach over a large corpus of open-source Java projects.

We identify the factors of interest characterizing the impact on maintainability of automatically applied refactorings. We find that the biggest influencing factor is token volume. We define token volume as the combined number of tokens in all clone instances in a clone class. The majority of duplicates with a token volume of 63 or more tokens improve maintainability when refactored. The amount of external data that needs to be passed to the merged location of the duplicate is the other important factor: the majority of clones requiring more than two external parameters decrease maintainability.

***Index Terms*—code clones, refactoring, static code analysis, object-oriented programming**

I. Introduction

Duplicate fragments in source code (also named "code clones") are often seen as one of the most harmful types of technical debt [**?**]. Duplicate fragments create implicit dependencies that make the code harder to maintain as the resolution of erroneous behavior in one location may have to be applied to all the cloned code as well [**?**]. Apart from that, code clones can contribute up to 25% of the code size [**?**].

To reduce duplication of a codebase, there are several refactoring techniques that can be applied. The most used refactoring technique to deal with code clones is "Extract Method" [**?**]. This technique entails that the duplicated code is moved to a new method, which is then called from all locations where the duplicated code was used. To be able to execute such a refactoring, several preconditions apply [**?**].

still missing link on how current solutions do not help improving maintainability - our message from monday night is not crystal clear yet

In this study, we analyze which preconditions apply to determine whether a clone could and should be automatically refactored. We list criteria that influence the maintainability of clones when refactored. We evaluate the context of clones to determine which refactoring techniques are best suited. We define the context of clones as the inheritance relation between clone instances in a clone class and the location of clone instances in the codebase. We showcase the proposed technique as a tool that automatically refactors clones by extracting new methods out of duplicated code.

We measure several maintainability metrics before and after refactoring to determine the impact on system maintainability of applying the refactoring. These metrics are volume, method parameters, duplication and complexity. Using this setup, we gain empirical data about the maintainability improvement of clones on a large set of data in a controlled environment. Because the refactoring is done automatically, we also consider micro-clones, which are shown to be important for refactoring consideration [**?**]. We use this data to define which characteristics of clones have the biggest influence of system maintainability when such a clone is refactored. This allows for more a accurate suggestion of code clones for refactoring and can assist in the refactoring process.

Most clone detection tools can be configured using thresholds. These thresholds indicate the minimum number of lines, tokens and/or statements that must be spanned for duplicate fragments to be considered clones. Often, such thresholds are intuitively chosen [**?**], [**?**] or based on a quartile distribution of empirical data [**?**]. Using the maintainability score we can find support for which thresholds should be chosen to increase the chance to find clones that improve maintainability when refactored.

Over a corpus of 2.267 Java project our tool refactored 12.710 clone classes. We found that the main influencing factor on maintainability when refactored is token volume. We define token volume as the combined number of tokens in all clone instances in a clone class. When refactoring clones with a very small token volume, we found that the maintainability decreases when such clones are refactored. The average clone with a clone volume of 65 or larger improves maintainability when refactored. The other factor with a major influence on maintainability is the number of parameters that are required for the extracted method. We found that if a clone requires more than 2 parameters in the extracted method, it is more likely to decrease maintainability.

Check whether this really fits here

This paper is organized as follows. In Sec. II we revisit key clone refactoring background. Sec. III discusses state-of-the-art in automated clone refactoring research. In Sec. IV we propose a definition for clones such that they could and should be automatically refactored. In Sec. VI we introduce our clone refactoring tool. In Sec. VII we explain the setup of our experiments; we show the results in Sec. VIII and discuss them in Sec. IX.

II. Background

We briefly describe relevant code clone and clone refactoring research.

A. Code clones

We use two concepts to argue about code clones [?]:

Clone instance: A single cloned fragment.

Clone class: A set of similar clone instances.

Duplication in code is found in many different forms. Most often duplicated code is the result of a programmer reusing previously written code [?], [?]. Sometimes this code is then adapted to fit the new context. To reason about these modifications, several clone types have been proposed [?]:

Type I: Identical code fragments except for variations in whitespace (may be also variations in layout), and comments.

Type II: Structurally/syntactically identical fragments except variations in identifiers, literals, types, layout, and comments.

Type III: Copied fragments with further modifications. Statements can be changed, added or removed next to variations in identifiers, literals, types, layout, and comments. A higher type of clone means that it is harder to detect and refactor. Many studies adopt these clone types, analyzing them further and writing detection techniques for them [?], [?], [?].

B. Clone Refactoring

1) Refactoring techniques: The most common technique for refactoring clones is "Extract Method" [?], which can be applied on code inside the body of a method. Applying this technique entails moving functionality from the body of a method to a new method. To reduce duplication with this technique, the contents of a single clone instance are extracted to a new method and all further locations of the clone are replaced by a call to the new method.

2) Preconditions: Clones detected by current detection techniques [?], [?], [?], [?] are subject to a set of preconditions to determine whether they can be refactored [?]:

1) The parameterization of differences between the matched statements should not break existing data-, anti-, and output-dependencies.
2) The unmatched statements should be movable before or after the matched statements without breaking existing data-, anti-, and output-dependencies.
3) The duplicated code fragments should return at most one variable of the same type.
4) Matched branching (`break`, `continue`) statements should be accompanied with corresponding matched loop statements.
5) Matched variables having different subclass types should call only methods that are declared in the common superclass or are being overridden in the respective subclasses.
6) The parameterization of fields belonging to differences between the mapped statements is possible only if they are not modified.
7) The parameterization of method calls belonging to differences between the mapped statements is possible only if they do not return a void type.
8) The mapped statements within the clone fragments should not contain any conditional return statements.

III. Related Work

We outline relevant research to clone refactoring. A significant aspect is the context of clones.

A. Clone context analysis

Golomingi [?] explores mapping the relation between clone instances to refactoring methods. The author analyses the refactoring methods described by Martin Fowler [?] and analyzes what refactoring methods can be used to refactor clones with what inheritance relations. The identified clone relations are: Ancestor, Common Hierarchy, First Cousin, Same Method, Sibling, Single Class, Superclass and Unrelated. We extend this list with several more fine-grained relations, suitable for automatic refactoring (see Sec. VI-B1).

Fontana et al. [?], [?] combine the research by Golomingi [?] with clone types 1 and 3 [?]. They use a large corpus [?] on which they perform statistical analyses of clone relations together with clone types. We repeat this research with a vastly different setup: we use the clone definition from Sec. IV and the relation categories outlined in Sec. VI-B1.

B. Clone refactoring

Krishnan et al. [?] approach clone refactoring as an optimization problem: how variability between cloned fragments influences the refactoring techniques required and their implications on system design. The main focus of this study is to find out which clones **can** be refactored. We extend this work by looking into which clones **should** be refactored. We propose definitions for refactorable clones together with thresholds to be able to limit their negative impact on system design. We measure which clones improve maintainability when refactored. This results in a set of thresholds that can be used to detect and refactor clones that should be refactored.

IV. Defining refactorable clones

We discuss how we can define clones such that they can be automatically refactored while avoiding changing the functional behavior and other negative side effects on the source code, such as decreasing the maintainability of the code.

A. Ensuring equality

Most modern clone detection tools detect duplication by comparing the code textually together with the omission of certain tokens [?], [?]. Clones detected by such means may not always be suitable for refactoring, because textual comparison fails to take into account the context of certain symbols in the code. Information that gets lost in textual comparison is the referenced declaration for type, variable and method references. Equally named type, variable and method references may refer to different declarations with a different

```java
package com.sb.game;

import java.util.List;

public class GameScene
{
  public void addToList(List l) {
    l.add(getClass().getName());
  }

  public void addTen(int x) {
    x = x + 10; // Addition
    Notifier.notifyChanged(x);
    return x;
  }
}
```

```java
package com.sb.fruitgame;

import java.awt.List; // Different type

public class LoseScene
{
  public void addToList(List l) {
    l.add(getClass().getName());
  }

  public void concatTen(String x) {
    x = x + 10; // String concatenation
    Notifier.notifyChanged(x);
    return x;
  }
}
```

Fig. 1. Example of textually equal code fragments that differ in functionality.

implementation (Fig 1 shows an example of this). Such clones can be harder to refactor, if beneficial at all.

To detect automatically refactorable clones, we propose to:

- Compare types by their fully qualified identifier (FQI). The FQI of a type reference describes the full path to where it is declared.
- Compare variable references not only by their name but also by their type (FQI).
- Compare method references by their fully qualified signature (FQS). The FQS of a method reference describes the full path to where it is declared, plus the FQI of the type of each of its parameters.

B. Allowing variability in a controlled set of expressions

Often, duplication fragments in source code do not match exactly [**?**]. When developers duplicate fragments of code, they modify the duplicated block to fit its new location and purpose. To detect duplicate fragments with minor variance, we look into what expressions we can allow variability in, while still being automatically refactorable.

We define the following expressions as automatically refactorable when varied:

- **Literals**: Only if all varying literals in a clone class have the same type.
- **Variables**: Only if all varying variables in a clone class have the same type.
- **Method references**: Only if the return value of referenced methods match (or are not used).

Often when allowing such variance, trade-offs come into play [**?**], [**?**]. For instance, variance in literals may require the introduction of a parameter to an extracted method if the "Extract Method" refactoring method is used, increasing the required effort to comprehend the code [**?**].

C. Gapped clones

Sometimes, when fragments are duplicated, a statement is inserted or changed severely for the code to fit its new context [**?**]. When dealing with such situations, there are several opportunities to refactor so-called "gapped clones" [**?**], [**?**]. "Gapped clones" are two clone instances separated by a "gap" of non-cloned statement(s). We define the following methods to refactor such clones, based on research by Tsantalis et al [**?**]:

- Wrap the difference in statements in a conditionally executed block, one path for each different (group of) statement(s).
- Use a lambda function to pass the difference in statements from each location of the clone [**?**].

For both of these techniques, a trade-off is at play. This is because these solutions increase the complexity and volume of the source code in favor of removing a clone.

V. Survey Existing Clone Detection Tools

To investigate the impact of the definitions introduced in Sec. IV, we need a tool to detect refactorable clones according to these definitions. We conducted a short survey on (recent) clone detection tools to detect such clones that can be refactored. The results of our survey are displayed in table I. We chose a set of tools that are open source and can analyze a popular object-oriented programming language. Next, we formulate the following four criteria by which we analyze these tools:

1) **Should find clones in any context.** Some tools only find clones in specific contexts, such as only method level clones. We want to perform an analysis of all clones in projects to get a complete overview.

2) **Finds clone classes in control projects.** We assembled a number of control projects to assess the validity of the surveyed clone detection tools.
3) **Can analyze resolved symbols.** When detecting the clones proposed in Sec. IV, it is important that we can analyze resolved symbols (for instance a type reference).
4) **Extensive detection configuration.** Detecting our clone definitions, as proposed in Sec. IV, require to have some understanding about the meaning of tokens in the source code (whether a certain token is a type, variable, etc.). The tool should recognize such structures, in order for us to configure our clone type definitions in the tool.

TABLE I
OUR SURVEY ON CLONE DETECTION TOOLS.

Clone Detection Tool	**(1)**	**(2)**	**(3)**	**(4)**
Siamese [?]				✓
NiCAD [?], [?]	✓	✓		
CPD [?]	✓	✓		
CCFinder [?] D-CCFinder [?]	✓	✓		
CCFinderSW [?]	✓			✓
SourcererCC [?] Oreo [?]	✓			✓
BigCloneEval [?]	✓	✓		
Deckard [?]	✓		✓	
Scorpio [?], [?]	✓		✓	✓

Apart from these criteria, we found that the output of these clone detection tools cannot be post-processed to find the clone definition proposed in Sec. IV. This is mainly because these clones are not necessarily a subset of clones detected by these tools and will thus require an analysis of the entire system.

VI. A TOOL FOR AUTOMATIC CLONE REFACTORING

None of the surveyed tools were suitable for our definitions of refactorable clones. Therefore, we built a new tool, CR[1] This tool goes through a 3-step process to automatically refactor clones as shown in Fig. 2. In Sec. VI-A we describe how CR detects refactorable clones. In Sec. VI-B we describe how CR maps the context of clones as input for the refactoring process described in Sec. VI-C.

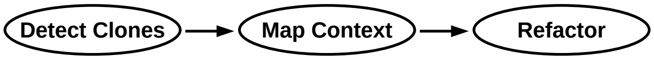

Fig. 2. Abstract flow diagram of CR.

A. Clone detection

We use an AST-based method to detect clones. Clones are detected on a statement level: only full statements are considered as clones. In Sec. IV-B we described that we limit the variability between variable, literal and method reference expressions by a threshold. This threshold is the percentage of different expressions against the total number of expressions in the source code:

$$\text{Variability} = \frac{\text{Different expressions}}{\text{Total expressions}} * 100 \tag{1}$$

After all clones have been detected, CR determines whether clone classes can be merged into gapped clones (see Sec. IV-C). The maximum size of the gap is limited by a threshold. This threshold is the percentage of (not-cloned) statements in the gap against the sum of statements in both clones surrounding it:

$$\text{Gap Size} = \frac{\text{Statements in gap}}{\text{Statements in clones}} * 100 \tag{2}$$

Note that unlike the expression variability threshold, this threshold can exceed 100%. This is because, in theory, the gap can be larger than the clones surrounding it.

B. Context mapping

After clones are detected, we map the context of these clones. We identify two properties of clones as their context: relation [?] and location [?]. We identify categories for each of these properties, to get a detailed insight into the context of clones.

1) Relation: Clone instances in a clone class may have a relation with each other through inheritance. This relation has a big impact on how the clone should be refactored [?]. We define the following categories to map the relation between clone instances in a clone class, partly derived from Fontana et al [?] (see Sec III-A). These categories do not map external classes (classes outside the project, e.g. belonging to a library) unless explicitly stated:

- **Common Class**: All clone instances are in the same class.
  - **Same Method**: All clone instances are in the same method.
  - **Same Class**: All clone instances are in the same class.
- **Common Hierarchy**: All clone instances are in the same inheritance hierarchy.
  - **Superclass**: Clone instances reside in a class or its parent class.
  - **Sibling Class**: All clone instances reside in classes with the same parent class.
  - **Ancestor Class**: All clone instances reside in a class, or any of its recursive parents.
  - **First Cousin**: All clone instances reside in classes with the same grandparent class.
  - **Same Hierarchy**: All clone instances are part of the same inheritance hierarchy.
- **Common Interface**: All clone instances are in classes that have the same interface.
  - **Same Direct Interface**: All clone instances are in classes that have the same interface.
  - **Same Class**: All clone instances are in an inheritance hierarchy that contains the same interface.

[1] Name anonymized for the blind review, camera-ready paper would include a link to the GitHub repo of the tool.

```java
int a = getA();
while(a<1000) {
  a *= 5;
  doC();
}
```

```java
int a = getA();
while(a<1000) {
  a *= 5;
  doB(a);
}
```

Fig. 3. A clone that spans a block partially.

- **Unrelated**: All clone instances are in classes that have the same interface.
  - **No Direct Superclass**: All clone instances are in classes that have the Object class as their parent.
  - **No Indirect Superclass**: All clone instances are in a hierarchy that contains a class that has the Object class as their parent.
  - **External Superclass**: All clone instances are in classes the same external class as their parent.
  - **Indirect External Ancestor**: All clone instances are in a hierarchy that contains a class that has an external class as their parent.

Based on these relations, CR determines where to place the cloned fragment when extracted to a new method. These categories are mutually exclusive: a clone class is flagged as the first relation in the above list that it applies to. The code of clones that have a *Common Class* relation can be refactored by placing the cloned code in this same class [**?**]. The code of clones with a *Common Hierarchy* relation can be placed in the intersecting class in the hierarchy (the class all clone instances have in common as an ancestor) [**?**]. The code of clones with a *Common Interface* relation can be placed in the intersecting interface [**?**], but in the process has to become part of the classes' public contract. The code of clones that are *Unrelated* can be placed in a newly created place. The state-of-the-art uses a utility class [**?**], whereas CR creates a new superclass or interface abstraction. The reason CR creates a superclass or interface rather than a utility class is that it makes the relation explicit and allows previously unrelated clones to become related.

2) Location: The location of a clone instance helps determine what refactoring techniques can be applied to it. We define the following categories of clones based on their location:

1) **Full Method/Constructor/Class/Interface/Enumeration:** A clone that spans a full class, method, constructor, interface or enumeration, including its declaration.
2) **Partial Method/Constructor Body:** A clone that spans (a part of) the body of a method/constructor.
3) **Several Methods:** A clone that spans over two or more methods, either fully or partially, but does not span anything but methods.
4) **Only Fields:** A clone that spans only global variables.
5) **Other:** Anything that does not match with above-stated categories.

The categories denote that a full declaration (method, class, etc.) often denotes redundancy and are often easy to refactor: one of the declarations is redundant and should be removed. Clones in the "Partial Method/Constructor" category can often be refactored using the "Extract Method" refactoring technique [**?**]. Clones consisting of *Several Methods* give a strong indication that cloned classes are missing some form of abstraction, or their abstraction is used inadequately. Clones consisting of *Only Fields* often indicate data redundancy: different classes use the same data.

C. Refactoring

Table IV shows that most clones are found in method bodies. Therefore, we focus on refactoring such clones which entails using the "Extract Method" refactoring technique. We show which clones CR refactors and how it applies these transformations.

1) Refactorability: Several factors may obstruct the possibility to extract code to a new method [**?**]:

- **Complex Control Flow**: This clone contains `break`, `continue` or `return` statements, obstructing the possibility of method extraction.
- **Spans Part Of A Block**: This clone spans a part of a block statement. An example of this is shown in Fig. 3.
- **Is Not A Partial Method**: If the clone does not fall in the "Partial method" category of Sec. VI-B2, the "Extract Method" refactoring technique cannot be applied.
- **Returns Multiple Values**: If a clone modifies or declares multiple pieces of data that it should return.
- **Top-Level Non-Statement**: If one of the top-level AST nodes of the clone is not a statement. For instance, if a (part of) an anonymous class is cloned.
- **Can Be Extracted**: This clone is a fragment of code that can directly be extracted to a new method. Then, based on the relation between the clone instances, further refactoring techniques can be used to refactor the extracted methods (for instance "Pull Up Method" for clones in sibling classes).

Clones that do not fall in the *Can Be Extracted* category may require additional transformations or other techniques to refactor. CR only refactors the clones that *Can Be Extracted*.

2) AST Transformation: CR uses JavaParser [**?**]: an AST-parsing library that allows to modify the AST and write it back to source code. To refactor clones, CR creates a new method declaration and moves all statements from a clone

Big todo –> Find literature, introinstance in the clone class to the new method. This method is placed according to the relation between the clone instances (see Sec. VI-B1). CR analyzes the source code of the extracted method and populates it with the following properties:

- **Parameters**: For each variable used that is not accessible from the scope of the extracted method.
- **A return value**: If the method modifies or declares local data that is needed outside of its scope, or if the cloned fragments already returned data.
- **Thrown exception**: If the method throws an uncaught exception that is not a `RuntimeException`.

CR then removes all cloned code and replaces it with a call to the newly created method. In case of a return value, CR either assigns the call result, declares it or returns it accordingly.

D. Maintainability Metrics

After applying the refactorings, CR measures the impact of the refactoring on maintainability metrics. We first state the metrics that we measure and then define the characteristics of the applied refactoring that influence the metrics.

1) Maintainability metrics:

rephrase this text for clarity: this paper – which paper?

CR measures the impact on maintainability metrics of refactoring source code for each clone class that is refactored. These metrics are derived from Heitlager et al. [?]. This paper defines a set of metrics to measure the maintainability of a system. For each of these metrics, risk profiles are proposed to determine the maintainability impact on the system of a whole.

how did you change them?

To determine whether the maintainability improves when refactoring a given clone, we need to measure the impact of fine-grained changes. Therefore, we measure only a subset of the metrics [?] and focus on the absolute metric changes (instead of the risk profiles). The subset of metrics we choose to focus on are all measured on method level (as the other metrics show a lesser impact on the maintainability for these small changes). These metrics are:

- **Duplication**: Originally [?] measured by taking the amount of duplicated lines. We decided to use the amount of duplicated tokens part of a clone class instead, to have a stronger reflection of the impact of the refactoring by measuring a more fine-grained system property.
- **Method Complexity**: Originally [?] computed using MCCabe complexity [?]. The MCCabe complexity is a quantitative measure of the number of linearly independent paths through a method.
- **Method Interface Size**: The number of parameters that a method has. If a method has many parameters, the code may become harder to understand and it is an indication that data is not encapsulated adequately in objects [?].
- **Method size**: The longer a method is, the harder it becomes to comprehend and maintain [?]. The “extract method” refactoring technique is often used to split up long methods [?]. When refactoring duplicated code fragments in the body of a method, this could improve maintainability of the refactored (reduced size).

why is the original way ok for us?

2) Characteristics of the extracted method: To assess the impact of an applied refactoring, we selected the four characteristics that have the highest impact on the resulting maintainability scores:

- **Size (in tokens)**: The number of tokens in the body of the method. A larger number of tokens means that more duplicate code can be removed, and thus has a positive impact on maintainability.
- **Relation**: The relation category (Sec. VI-B1) by which this methods’ location was determined. Some relations are less favorable for maintainability than others.
- **Returns**: Whether the method calls return, declare, assign or don’t use any data from the extracted method. This affects maintainability because the data must be returned from the extracted method, and assigned at each method call. This increases the volume of the refactoring.
- **Parameters**: The number of parameters the extracted method has. If the extracted method uses data that is not available where it is located, the data must be passed using method parameters. If a method has many parameters, it becomes less reusable and harder to comprehend, thus harder to maintain [?]. Additionally, this increases the volume, because each call to the extracted method must pass all required data.

We hypothesize that these characteristics are the main factors influencing the impact on the maintainability of the system as a result of refactoring the clone. This is because there are no further transformations that are applied that influence the maintainability metrics.

VII. Experimental Setup

Our goal is analysing whether automatic refactoring of code clones leads to an improved maintainability. We first validate our automatic refactoring tool, CR. Next, we run our tool on a reference corpus and measure the change in the maintainability score.

This doesn’t say anything: explain how the section is organized

A. Tool Evaluation

We assess the correctness of CR through unit tests and empirical validation. First, we create a set of 57 control projects to verify the correctness in many (edge) cases. These projects contain clones for each relation, location, and refactorability category. Next, we run the tool over the corpus and manually verify samples of the acquired results. This way, we check both the correctness of the identified clones, their context, and their proposed refactoring.

We also test the correctness of the refactored code using the JFreeChart project [?]. JFreeChart has a high test coverage and working tests, which allows us to test the correctness of the program after running CR.

how many? why not all?

B. Corpus

Our corpus, consisting of open source Java projects, is derived from the corpus of a study that uses machine learning to determine the suitability of GitHub Java projects for data analysis purposes [?].

why did you choose this corpus?

CR requires all libraries of the analyzed projects (see Sec. IV-A). Therefore, we decided to filter the corpus to contain only projects using the Maven build automation system. The filtering scripts are publicly availble The camera-ready version would include the GitHub repo url.

This procedure results in 2.267 Java projects including all their dependencies . The projects vary in size and quality. The total size of all projects is 14.2M lines (11.3M when excluding whitespace) over a total of 100K Java files. This is an average of 6.3K lines over 44 files per project. The largest project in the corpus is *VisAD* with 502K lines.

this is confusing, how many original projects are there? were the dependencies in the original corpus?

figure with the distribution of project sizes?

C. Minimum clone size

When clones are very small, they may never be able to improve maintainability. The detrimental effect of the added volume of the newly created method exceeds the positive effect of removing duplication. We perform all our experiments with a minimum clone size of 10 tokens, as smaller clones are very unlikely to improve maintainability when refactored.

D. Thresholds

Most clone detection tools can be configured using thresholds. These thresholds indicate the minimum number of lines, tokens and/or statements that must be spanned for duplicate fragments to be considered clones. Often, such thresholds are intuitively chosen [?], [?] or based on a quartile distribution of empirical data [?].

E. Calculating a maintainability score

We use four metrics to determine maintainability (see Sec. VI-D1). We determine the value of each metric before and after for each refactored clone class, resulting in a delta metric score. We aggregate the deltas obtained for these metrics to draw a conclusion about the maintainability increase or decrease after applying a refactoring. We base our aggregation on the following assumptions derived from supporting evidence [?], [?]:

- All metrics are equal in terms of weight towards system maintenance effort.
- Higher values for the metrics imply lower maintainability.
- Normalizing each metric delta over all deltas obtained for that metric in our dataset results in equally weighted scores.
- all considered metrics do not interfere with the out-of-scope metrics

any ref?

We use the resulting aggregated maintainability score to analyze for each refactoring whether it increases or decreases the maintainability of the system.

We normalize each obtained metric delta using the “Standard score”, which is calculated as follows :

$$N_{metric} = \frac{\Delta X - \mu}{\sigma} \tag{3}$$

quartile or quantile?

Where ΔX is a metric delta, μ is the mean of all deltas for this metric and σ is the standard deviation of all deltas for this metric. This method works well for normalization of our data because as we divide by the standard deviation, outliers do not influence the resulting scores negatively .

We calculate the maintainability of a refactoring as:

$$N_{duplication} + N_{complexity} + N_{volume} + N_{parameters} \tag{4}$$

paragraph commented below seems more suitable for the discussion

VIII. Results

We first report the context results, then the refactorability results.

A. Clone context

To determine the refactoring method(s) that can be used to refactor most clones, we perform statistical analysis on the context of clones (see Sec. VI-B).

1) Relation: Table III shows the number of clone classes found for the entire corpus for each type of relations between clone instances (see Sec. VI-B1).

Category	***Relation***	***Clone Classes***	***Total***
Common Class	Same Class	22,893	31,848
	Same Method	8,955	
Common Hierarchy	Sibling	15,588	20,342
	Superclass	2,616	
	First Cousin	1,219	
	Common Hierarchy	720	
	Ancestor	199	
Unrelated	No Direct Superclass	10,677	20,314
	External Superclass	4,525	
	External Ancestor	3,347	
	No Indirect Superclass	1,765	
Common Interface	Same Direct Interface	7,522	13,074
	Same Indirect Interface	5,552	

TABLE III
NUMBER OF CLONE CLASSES PER CLONE RELATION

Our results show that most clones (37%) are in a common class. 24% of clones are in a common hierarchy. Another 24% of clones are unrelated. 15% of clones are in an interface.

2) Location: Table IV shows the number of clone classes found for the entire corpus for different locations (see Sec. VI-B2).

Category	***Location***	***Clone instances***	***Total***
Partial	Partial Method	219,540	229,521
	Partial Constructor	9,981	
Full	Full Method	12,990	13,173
	Full Interface	64	
	Full Constructor	58	
	Full Class	37	
	Full Enum	24	
Other	Several Methods	22,749	53,773
	Only Fields	17,700	
	Other	13,324	

TABLE IV
NUMBER OF CLONE INSTANCES FOR CLONE LOCATION CATEGORIES

The results show that 74% of clones span part of a method body (77% if we include constructors). 8% of clones span several methods. 6% of clones span only global variables. Only 4% of clones span a full declaration (method, class, constructor, etc.).

B. Refactorability

Table V shows to what extent clone classes can be refactored by using the “Extract Method” refactoring technique (see Sec. VI-C1).

Relation	*Duplication*	*Complexity*	*Parameters*	*Volume*	*#*	*Score*
Common Hierarchy	**-66.33**	**0.73**	**1.20**	**-8.85**	**2,202**	**0.23**
Superclass	-64.48	0.79	0.94	-7.22	229	0.42
Sibling	-70.07	0.69	1.28	-10.97	1,722	0.23
Same Hierarchy	-44.18	0.95	0.89	1.54	87	0.10
First Cousin	-42.69	0.89	0.93	4.86	144	0.02
Ancestor	-32.75	1.00	0.75	11.00	20	-0.03
Common Interface	**-47.06**	**0.83**	**1.04**	**4.50**	**1,044**	**-0.02**
Same Indirect Interface	-37.08	0.93	0.82	9.96	487	-0.01
Same Direct Interface	-55.79	0.75	1.24	-0.28	557	-0.02
Common Class	**-52.42**	**0.87**	**1.13**	**1.47**	**7,239**	**-0.02**
Same Class	-51.85	0.86	1.03	3.36	4,874	0.04
Same Method	-53.60	0.90	1.32	-2.44	2,365	-0.15
Unrelated	**-45.86**	**0.88**	**1.08**	**9.56**	**2,198**	**-0.15**
No Direct Superclass	-52.24	0.84	1.12	6.04	811	-0.06
External Superclass	-47.09	0.87	1.13	8.77	697	-0.17
External Ancestor	-35.73	0.93	0.95	14.58	586	-0.21
No Indirect Superclass	-44.89	0.84	1.18	14.08	104	-0.30
Grand Total	**-53.26**	**0.84**	**1.12**	**1.33**	**12,683**	**0.00**

TABLE II

INFLUENCE ON MAINTAINABILITY OF REFACTORING CLONES WITH THE SPECIFIED RELATION CATEGORIES.

Category	*All*	*%*
Can Be Extracted	24,157	28.2%
Is Not A Partial Method	21,625	25.3%
Top-level AST-Node is not a Statement	19,887	23.2%
Spans Part of a Block	13,181	15.5%
Multiple Return Values	5,622	6.6%
Complex Control Flow	1,106	1.3%

TABLE V

NUMBER OF CLONES THAT CAN BE EXTRACTED USING THE "EXTRACT METHOD" REFACTORING TECHNIQUE.

the paragraph below belongs to discussion

The results indicate that, given our refactorability criteria, 28% of clones can be automatically refactored. Clones in other categories may require other refactoring techniques or further transformations to be automatically refactorable.

C. Thresholds

CR has refactored 12.710 clone classes and measured the change in the selected metrics (Sec. VI-D1). Using the presented formulas (Sec. VII-E) we determine how the characteristics of the extracted method (see Sec. VI-D2) influence the maintainability of the resulting codebase after refactoring. We explore the data obtained by comparing the before- and after snapshots of the system for each separate refactoring. Using this data, we find supporting evidence regarding which thresholds are most likely to find clones that should be refactored to improve maintainability.

use the formulas labels to refer here

1) Clone Token Volume: Figure 4 shows the results by plotting the clone volume vs the average delta maintainability score. We define the *token volume* as the combined number of tokens in all clone instances in a refactored clone class. For higher token volume numbers we have fewer refactorings that refactor such clones, therefore we represent the x-axis as a logarithmic scale. The trendline intersects the "zero" line (maintainability does not increase nor decrease) at a token volume of 63.

2) Relation: Table II shows our data regarding how different relations influence maintainability. We have marked rows based on less than 100 refactorings red, as their result does not have statistical significance. Relations are ordered by their scores. Scores do not deviate much (-0.15 to 0.23), indicating that the relation between clones has a minor impact on maintainability. Overall, common hierarchy clones have the highest maintainability, whereas unrelated clones have the lowest maintainability.

Redo plot, explain axes units

not

3) Return Value: Table VI shows how the return value of the extracted method influences the maintainability of the resulting system. The return categories are ordered by their scores. Scores do not deviate much (-0.18 to 0.19), indicating that the return category has a minor impact on maintainability. When the result of the call to the extracted method is directly returned, the maintainability score is the highest. When no value is returned, the maintainability score is the lowest. The main reason that no return value is lowest, is that it is linked to a higher number of parameters required for the extracted method.

this sentence about relation impact seems more suitable for discussion

4) Parameters: Fig. 5 shows how an increase in parameters lowers the maintainability of the refactored code. On the primary x-axis, the maintainability is displayed. The secondary x-axis shows the number of refactorings. The y-axis shows the number of parameters.

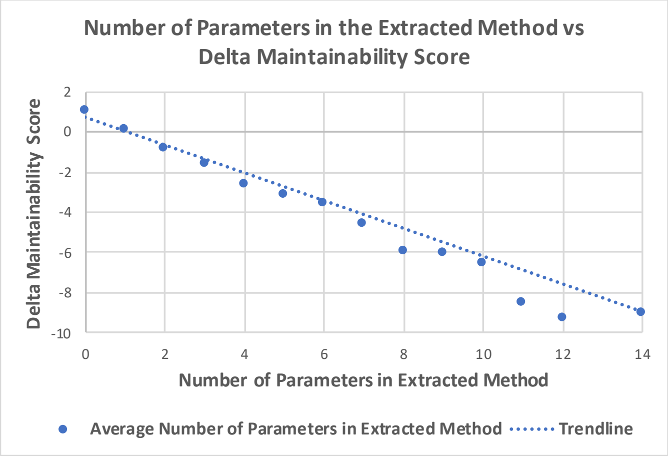

Fig. 5. Influence of number of method parameters on system maintainability.

this sentence about relation impact seems more suitable for discussion

IX. DISCUSSION

A. Clone Context

Regarding clone context, our results indicate that most clones (37%) are in a common class. This is favorable for refactoring because the extracted method does not have to be moved after extraction. 24% of clones are in a common hierarchy. These refactorings are also often favorable. Another 24%

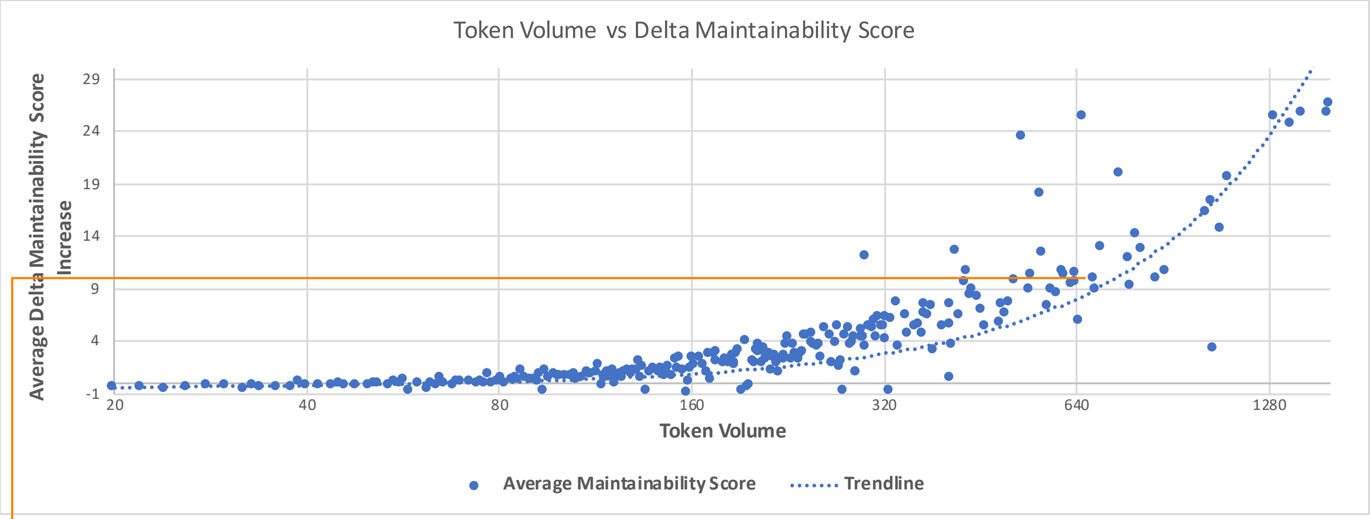

Fig. 4. A graph that shows how the size in tokens of the refactored clone affects maintainability.

Return Category	*Complexity*	*Parameters*	*Size*	*Duplication*	*#*	*Score*
Return	0.85	1.02	-3.84	-55.00	1,571	0.19
Declare	0.94	0.74	11.11	-49.19	5,177	0.15
Assign	0.79	1.07	0.43	-56.29	14	0.12
Void	0.76	1.49	-5.85	-56.35	5,921	-0.18
Grand Total	**0.84**	**1.12**	**1.33**	**-53.26**	**12,683**	**0.00**

TABLE VI
AVERAGE METRIC VALUES FOR REFACTORINGS OF CLONE CLASSES WITH THE SPECIFIED RETURN CATEGORY

of clones are unrelated, which is often unfavorable because it often requires a more comprehensive refactoring. 15% of clones are in an interface .

which ...

Regarding clone locations, 74% of clones span part of a method body (77% if we include constructors). 8% of clones span several methods, which often require refactorings on a more architectural level. 6% of clones span only global variables, requiring an abstraction to encapsulate these data declarations. Only 4% of clones span a full declaration (method, class, constructor, etc.).

B. Extract Method

28% of clones can be refactored using the “Extract Method” refactoring technique (50% if we limit our searching scope to method bodies). About 25% of clones do not span part of a method, therefore they cannot be refactored. Many clones (23%) do not have a statement as top-level AST-Node. Upon manual inspection, we noticed that the main reason is clones in lambda functions or in anonymous classes. About 15% of clones span only part of an AST-Node .

and?

C. Refactoring

Fig. 4 shows an increase in maintainability for refactoring larger clone classes. The tipping point, between a better and a worse maintainable refactoring, seems to lie around a token volume of 63 tokens. There are fewer large clones than small clones, resulting in a very limited statistical significance on our corpus when considering clones larger than 100 tokens.

how limited? give the p-value

Table II shows the results regarding refactorings that are applied to clones with diverse relations. More than 54% of the refactored clones are in a common class. This is significantly more than the percentage of clones in the common class relation reported in Table III. The number of refactored unrelated clones is smaller than the number reported in Table III (24% -> 18%). The main reason for this is that refactoring unrelated clones can change the relation of other clones in the same system. If we create a superclass abstraction to refactor an unrelated clone, other clones in those classes that were previously unrelated might become related.

The maintainability scores in Table II show that the most favorable clones to refactor are clones with a Sibling relation. The most unfavorable is refactoring clones to interfaces. However, the differences in maintainability in this table are generally small ; according to our data relations have only a minor impact on the maintainability of clones.

and?

Regarding the return type of refactored clones, Table VI shows that this has no major impact on maintainability. A method call to the extracted method that is directly returned and no return type extracted methods are slightly more favorable than the others. We think the main reason that the “Return” category is on top is that when a variable is declared at the end of the cloned fragment, CR directly returns its value and removes the declaration. This decreases the volume slightly.

wasn’t this a characteristic? this paragraph is somewhat confusing

Fig. 5 shows that more parameters negatively influence maintainability. Not only the number of parameters metric is negatively influenced, but more method parameters also increase volume for the extracted method and each of the calls to it. Because of that, we see that the trend of the graph in Fig. 5 decreases relatively rapidly.

one paragraph with a general remark on this discussion, summarizing our findings

X. Conclusion

one paragraph summarizing the context of our research

We defined automatically refactorable clones and created a tool to detect and refactor them. We measured statistical data with this tool over a large corpus of open-source Java software systems to get more information about the context of clones and how refactoring them influences system maintainability.

We defined two aspects as part of the context of a clone: relation and location. Regarding relations, over 37% of clones are found in the same class. About 24% of clones are in the same inheritance hierarchy. Another 24% of clones are unrelated. The final 15% of clones have the same interface. Regarding location, over 74% of clones span part of a method. About 8% span several methods. Only 4% of clones span a declaration (method, class, etc.) fully.

We built a tool that can automatically apply refactorings to 28% of the clones in our corpus using the “Extract Method” refactoring technique.

We measured the change in four maintainability metrics to determine the impact of each refactoring on system maintainability. We found that the most wasn't this 63?

prominent factor influencing maintainability is the size of the clone. We found that the threshold lies at a clone volume of 29 tokens per clone instance for system maintainability to increase after refactoring the clone. Another factor with a major impact on maintainability is the number of parameters that the extracted method requires to get all required data. We noticed that the inheritance relation of the clone and the return value of the extracted method has only a minor impact on system maintainability.