Language-Independent Detection of Clones with Renamed Variables

BY
Richard Wettel

Disertation Thesis

Faculty of Automatics and Computer Science of the
"Politehnica" University of Timişoara

Timişoara,
June 2005

Advisor:
Dr. Ing. Radu Marinescu

By eliminating the duplicates, you ensure that the code says everything once and only once, which is the essence of good design (Once And Only Once Rule).

Kent Beck

Contents

Chapter 1

Introduction

1.1 Motivation

1.1.1 Prerequisites

We start with a line-based language-independent code duplication detector named DuDe, introduced in [Wet04], able to detect exact clones or duplication chains made of exact clones separated by non-matching gaps. The duplication chain concept provides a mechanism to detect larger duplicated code fragments, with adaptations at the level of line of code (deletion, insertion or modification of lines of code).

The tool uses textual comparison, while for the granularity of the comparison, the line of code (LOC) was chosen because usually the Copy & Paste activities imply a number of lines of code, rather than a single one. It is appropriate for analyzing software systems written in any language, since the matching is done by textual comparison.

1.1.2 Context

The context in which code duplication appears and the fact that it is a sign of bad software design have already been discussed in [Wet04]. That is why we will focus on the concerns providing the motivation of this thesis and the improvements it proposes.

The main concern of this thesis is the detection of so-called parameterized clones. The author of [Bak93] states that: "...while some of the duplication in a software system may involve sections of code that are identical, much of the duplication involves sections of code that are not identical, but the same except for a systematic change of parameters such as identifiers and constants, e.g. each occurrence of first, last, 0, and fun in one section may be replaced by init, final, 1, and g, respectively, in the other section; this kind of correspondence

between sections of code is called a parameterized match". The same author, who is one of the early researchers concerned about clone detection, mentions the fact that: "commonly, code will have more parameterized matches than exact matches".

Parameterized clones have been also addressed by other approaches ([BYM+98], [KKI02], under various names, *i.e.*, near-miss clones and code portions with renamed variables, respectively. The author of [Bel02] defines them as type 2 clones, along with type 1 clones (exact clones) and type 3 clones (modified beyond variable or constant names).

That is why we consider this special type of clones very important to software maintenance. Furthermore, these are excellent candidates for the refactorings [FBB+99] needed to eliminate the duplicated code. Since the supporting tool [Wet04] could not detect such clones, the author made out of this disadvantage the main motivation for this thesis. We address the detection of such clones, by proposing improvement measures for our supporting tool.

1.1.3 Opponent Forces

Despite the fact that type 2 clones are sometimes obvious by simply visually checking the code, they proved to be much harder to detect automatically.

In order to detect renamed variables, a clone detector would need to be able to:

- analyze the data at a fine granularity level
- have access to semantical information on the analyzed data

An existent approach that zooms up to the level of tokens [KKI02], supported by a tool called CCFinder, involves lexical analysis, which further implies implementations of lexical analyzers for every programming language supported by the tool.

The first problem is that DuDe compares lines of code, thus it would not be able to detect modifications of lexical elements smaller than a line of code (the granularity is too coarse). Moreover, having language-specific knowledge means to lose the language-independence.

What we will try to achieve in this thesis is a set of improvements that will enhance our tool to address the detection of clones with renamed variables without losing the language independence.

1.1.4 Solutions

The main idea behind this was to replace the actual comparison method (the tool checks if two strings are identical) with the computation of a similarity metric, value that could indicate whether two lines of code are very similar to each other. Such an approach would be able to detect code fragments that differ in only variable names. The similarity should be computed with no need for lexical or syntactical information, so that the language independence is further maintained.

Since the two desiderates are antagonistic forces (precise results are based on variable identification, which needs language-dependent information, but the other desiderate is language independence), we will rather propose a solution which is a compromise that combines relevant results and language independence (or a minimum of language dependence).

1.2 Outline

Chapter 2 describes the part of nowadays software engineering fundaments related to the detection, avoidance and elimination of code clones. Then, there is a short section on the signs of bad design called "bad smells" [FBB+99], also related to code duplication and a practical guide to a set of refactorings that can eliminate the duplicated code. Furthermore, there is a section dedicated to design patterns, in particular the one used in the project. In the second part of the first chapter, we describe the tool foundation, which introduces the tool itself, then presents the concept of duplication chain and some structural information around it and ends with the description of the phases the tool executes in the process of clone detection.

After setting up the environment, the problem and the concepts on which the whole approach is based, chapter 3 presents the proposed approach on detecting the clones with renamed variables. It cuts through directly to the idea behind the enhancement to detect clones with renamed variables. Then it introduces the Levenshtein distance, since the approach is mainly based on it. And the idea is concretized in the approximate matching strategies based on two types of similarity. Next, a short description of the new system's architecture is presented, along with UML class diagrams, for a deeper understanding of the approach. In the end, the newly introduced data exchange features are presented, starting with the XML parsing technology, the data format, described by means of an XML schema and some brief information about XML validation.

Chapter 4 puts us in the chair of a critic, and starts to analyze what we realized with this work. We briefly introduce the features of the tool (graphical user interface, configuring the detection process through parameters). Next, DuDe

is subject of an experiment that should prove the applicability of the new proposed approximate matching strategies (conducted on 4 C and Java projects). Then, a deeper examination of the experiment's results drives us to conclusions.

Chapter 5 discusses the state of the art in the field of code duplication detection, related to object-oriented design. There has been international concern towards clone detection and tools support, materialized in 2 conferences (2002 and 2003). The first conference has been in some way a benchmark to some of the tools in this field of research. The position of the tool in the context of actual tools is also discussed. After discussing the pros and cons of the existent tools, we present some ideas close to the ideas in our approach and the differences between our approach and theirs.

Chapter 6 draws a line, taking us to the conclusions. There is a section on good and bad regarding the new approach. A brief evaluation of the personal contribution of this work and a final report on possible future work on this tool.

Chapter 2

Fundaments

2.1 Principles

This section's goal is to introduce some of the idioms and principles that are closely related to code duplication.

2.1.1 Clone Related Bad Smells

Kent Beck and Martin Fowler define some signs of problems that can be addressed by refactoring the code, which they call "bad smells in code". The first mentioned symptom is code duplication, described by them as "number one in the stink parade". If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

Code duplication

The simplest duplicated code problem is when you have the same expression in two methods of the same class. Then all you have to do is *Extract Method* and invoke the code from both places.

Another common duplication problem is when you have the same expression in two sibling subclasses. You can eliminate this duplication by using *Extract Method* in both classes then *Pull Up Field*. If the code is similar but not the same, you need to use *Extract Method* to separate the similar bits from the different bits. You may then find you can use *Form Template Method*. If the methods do the same thing with a different algorithm, you can choose the clearer of the two algorithms and use Substitute Algorithm.

If you have duplicated code in two unrelated classes, consider using *Extract Class* in one class and then use the new component in the other. Another possibility is that the method really belongs only in one of the classes and should

be invoked by the other class or that the method belongs in a third class that should be referred to by both of the original classes. You have to decide where the method makes sense and ensure it is there and nowhere else.

Switch Statements

One of the most obvious symptoms of object-oriented code is its comparative lack of switch (or case) statements. The problem with switch statements is essentially that of **duplication**. Often you find the same switch statement scattered about a program in different places. If you add a new clause to the switch, you have to find all these switch, statements and change them. The object-oriented notion of polymorphism gives you an elegant way to deal with this problem.

Most times you see a switch statement you should consider polymorphism. The issue is where the polymorphism should occur. Often the switch statement switches on a type code. You want the method or class that hosts the type code value. So use *Extract Method* to extract the switch statement and then *Move Method* to get it onto the class where the polymorphism is needed. At that point you have to decide whether to *Replace Type Code with Subclasses* or *Replace Type Code with State/Strategy*. When you have set up the inheritance structure, you can use *Replace Conditional with Polymorphism*. If you only have a few cases that affect a single method, and you don't expect them to change, then polymorphism is overkill. In this case *Replace Parameter with Explicit Methods* is a good option. If one of your conditional cases is a null, try *Introduce Null Object*.

Parallel Inheritance Hierarchies

Parallel inheritance hierarchies is really a special case of shotgun surgery. In this case, every time you make a subclass of one class, you also have to make a subclass of another. You can recognize this smell because the prefixes of the class names in one hierarchy are the same as the prefixes in another hierarchy.

The general strategy for eliminating the duplication is to make sure that instances of one hierarchy refer to instances of the other. If you use *Move Method* and *Move Field*, the hierarchy on the referring class disappears.

2.1.2 Eliminate Clones by Refactoring

This section will introduce definitions of refactoring, from different points of view and continues with the description practical refactorings for the elimination of code duplication.

Definition 2.1 (Refactoring) *The process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure is called* **refactoring**.

It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written[FBB+99].

Definition 2.2 (Refactoring) *A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior is called* **refactoring**.

Refactorings Explained

This section will make clear some of the refactorings proposed in the three cases that have to do with code duplication.

1. *Parameterize Method*
   Several methods do similar things but with different values contained in the method body. Create one method that uses a parameter for the different values.
   Motivation: You may see a couple of methods that do similar things but vary depending on a few values. In this case you can simplify matters by replacing the separate methods with a single method that handles the variations by parameters. Such a change removes duplicate code and increases flexibility, because you can deal with other variations by adding parameters.

2. *Pull Up Field*
   Two subclasses have the same field. Move the field to the superclass.
   Motivation: If subclasses are developed independently, or combined through refactoring, you often find that they duplicate features. In particular, certain fields can be duplicates. Such fields sometimes have similar names but not always. The only way to determine what is going on is to look at the fields and see how they are used by other methods. If they are being used in a similar way, you can generalize them. Doing this reduces duplication in two ways. It removes the duplicate data declaration and allows you to move from the subclasses to the superclass behavior that uses the field.

3. *Pull Up Method*
   You have methods with identical results on subclasses. Move them to the superclass.
   Motivation: Eliminating duplicate behavior is important. Although two duplicate methods work fine as they are, they are nothing more than a breeding ground for bugs in the future. Whenever there is duplication, you face the risk that an alteration to one will not be made to the other. Usually it is difficult to find the duplicates. The easiest case of using

Pull Up Method occurs when the methods have the same body, implying there's been a **copy and paste**. Of course it's not always as obvious as that. You could just do the refactoring and see if the tests croak, but that puts a lot of reliance on your tests. I usually find it valuable to look for the differences; often they show up behavior that I forgot to test for. Often Pull Up Method comes after other steps. You see two methods in different classes that can be parameterized in such a way that they end up as essentially the same method. In that case the smallest step is to parameterize each method separately and then generalize them. Do it in one go if you feel confident enough.

4. *Extract Superclass*
   You have two classes with similar features. Create a superclass and move the common features to the superclass.
   Motivation: Duplicate code is one of the principal bad things in systems. If you say things in multiple places, then when it comes time to change what you say, you have more things to change than you should. One form of duplicate code is two classes that do similar things in the same way or similar things in different ways. Objects provide a built-in mechanism to simplify this situation with inheritance. However, you often don't notice the commonalities until you have created some classes, in which case you need to create the inheritance structure later. An alternative is Extract Class. The choice is essentially between inheritance and delegation. Inheritance is the simpler choice if the two classes share interface as well as behavior. If you make the wrong choice, you can always use Replace Inheritance with Delegation later.

5. *Form Template Method*
   You have two methods in subclasses that perform similar steps in the same order, yet the steps are different. Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.
   Motivation: Inheritance is a powerful tool for eliminating duplicate behavior. Whenever we see two similar methods in a subclass, we want to bring them together in a superclass. But what if they are not exactly the same? What do we do then? We still need to eliminate all the duplication we can but keep the essential differences. A common case is two methods that seem to carry out broadly similar steps in the same sequence, but the steps are not the same. In this case we can move the sequence to the superclass and allow polymorphism to play its role in ensuring the different steps do their things differently. This kind of method is called a template method [Gang of Four].

2.1.3 Design Patterns

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. Your design should be specific to the problem at hand

but also general enough to address future problems and requirements. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Yet experienced object-oriented designers do make good designs. What is the magic behind the solutions of experienced designers, that makes such a difference?

Expert designers do not necessary know to solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

Most of the programmers when reading the requirements have at least once had the feelings they already solved that problem, or a similar one. If they could remember the essence of the solution, they would only have to adapt it to the specifics of the problem and not reinvent it all over.

Definition 2.3 (Pattern) *"Each* **pattern** *describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Ale79]. (C. Alexander, The Timeless Way of Building, 1979)*

The book "Design Patterns: Elements of Reusable Object-Oriented Software" [GHJV95] by the Gang of Four (GOF) does exactly that: makes that experience knowledge persistent, by associating a software architecture problem that often comes up, the solution to that problem and the consequences of applying that solution with a name:

- the problem explains when to apply the pattern, namely the problem and the context it is associated with
- the solution describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution does not describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
- The consequences are the results and trade-offs of applying the pattern. They are critical for evaluating design alternatives and for understand-

ing the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them. The consequences help us putting in balance the advantages and disadvantages of one solution or another and choosing the one the serves or purposes the best.

- the name of the pattern is the element that enriches our design vocabulary and lets us express in a word or two a design problem, its solutions, and consequences. Naming a pattern lets us design at a higher level of abstraction.

While different mechanisms offered by the object-oriented programming languages (*i.e.*, inheritance) provide means for the reuse of software, design patterns are the key for the reuse of design.

In this chapter we will discuss some of the patterns used in the architecture of DuDe and other patterns that have to do, in a way or the other, with code duplication.

Observer

The **Observer** pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

The structure is presented in Fig. 2.1. There may be many observers and the

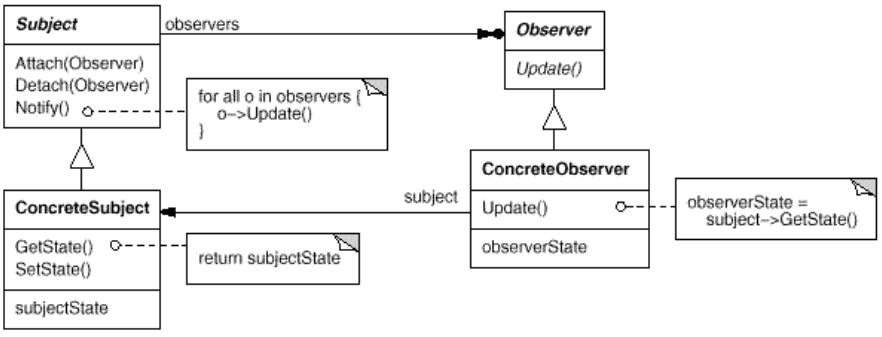

Figure 2.1: Observer

only thing they have to share is extending the Observer abstract class. Each observer may react differently to the same notification from the ConcreteSubject. The data-source (Subject) should be as decoupled as possible from the observer to allow observers to change independently of the subject. The Subject is completely decoupled, for it knows only that it has a list of subscribers (Observer objects) that it has to notice when something in its state changes. This is why the Observer pattern is also known as Publish-Subscribe. An example of interaction between a subject and two observers is presented in Fig. 2.2.

The consequences of applying the Observer pattern are:

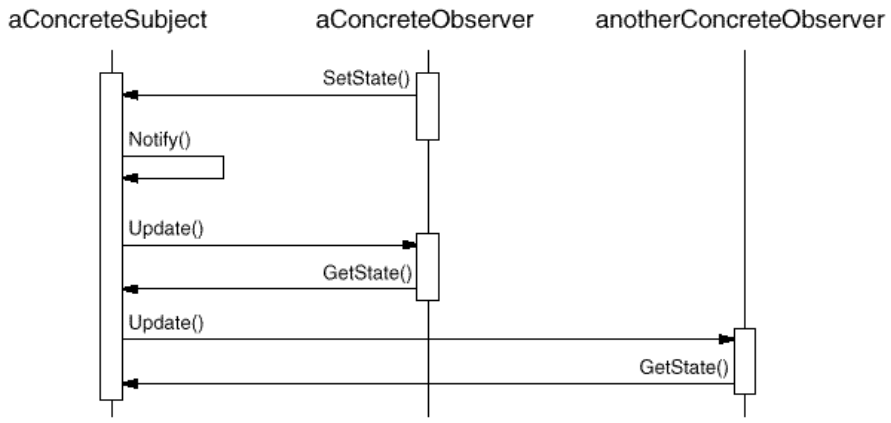

Figure 2.2: Sequence diagram for Observer

1. The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers.

2. Abstract coupling between Subject and Observer. All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.

3. Support for broadcast communication. Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.

4. Unexpected updates. Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A minor operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that aren't well-defined or maintained usually lead to false updates, which can be hard to track down.

Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

The structure (Fig.2.3) of this pattern helps avoiding the coupling of the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

An example of building the chain is described in Fig. 2.4. After building

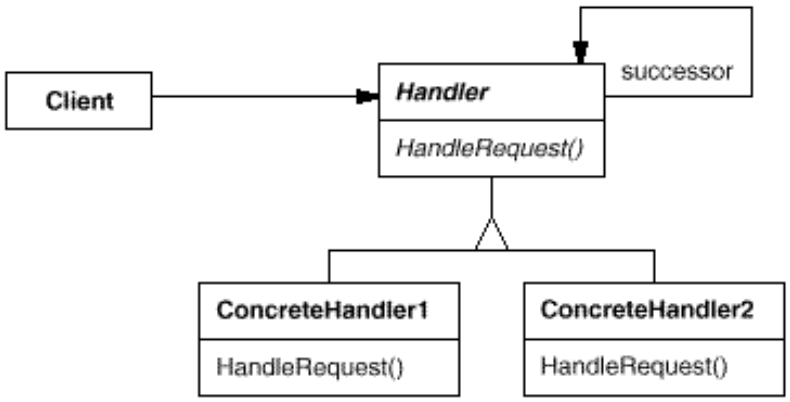

Figure 2.3: Class diagram for Chain Of Responsibility

the chain, a client calls the *handle()* method of the first element of the chain. If this cannot handle the request, it transmits the request over to the next element in the chain.

A typical example of applying this pattern is context-sensitive help facility for a graphical user interface. The user can obtain help information on any part of the interface just by clicking on it. The help that's provided depends on the part of the interface that's selected and its context; for example, a button widget in a dialog box might have different help information than a similar but-

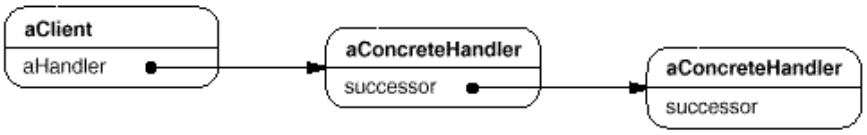

Figure 2.4: Building the chain

ton in the main window. If no specific help information exists for that part of the interface, then the help system should display a more general help message about the immediate context the dialog box as a whole, for example. This interaction is captured in the sequence diagram presented in Fig. 2.5.

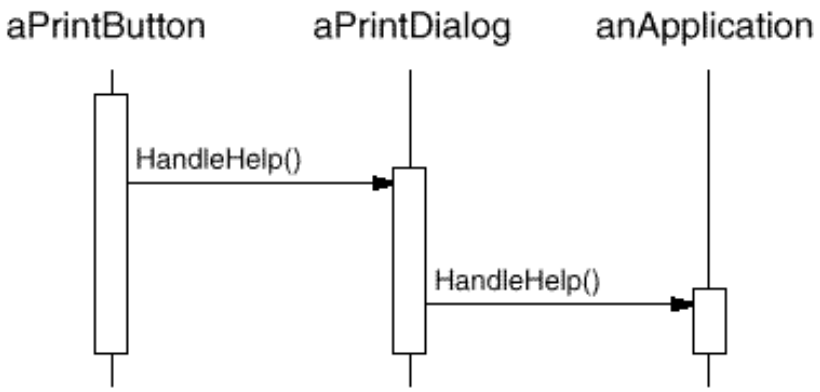

Figure 2.5: Interaction within the chain

The problem here is that the object that ultimately provides the help isn't known explicitly to the object (e.g., the button) that initiates the help request. What we need is a way to decouple the button that initiates the help request from the objects that might provide help information. The Chain of Responsibility pattern defines how that happens.

The consequences of applying this pattern are:

- *Reduced coupling.* The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled "appropriately." Both the receiver and the sender have no explicit knowledge of each other, and an object in the chain doesn't have to know about the chain's structure.

- *Added flexibility in assigning responsibilities to objects.* Chain of Responsibility gives you added flexibility in distributing responsibilities among objects. You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time. You can combine this with subclassing to specialize handlers statically.

- *Receipt isn't guaranteed.* Since a request has no explicit receiver, there's no guarantee it'll be handledthe request can fall off the end of the chain without ever being handled. A request can also go unhandled when the chain is not configured properly.

Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

This pattern is directly related to code duplication. The use of this pattern can avoid duplicating code. The Template Method pattern should be used to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.

1. When common behavior among subclasses should be factored and localized in a common class to avoid **code duplication**. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.

2. To control subclasses extensions.

Consider an application framework that provides Application and Document classes. The Application class is responsible for opening existing documents stored in an external format, such as a file. A Document object represents the information in a document once it's read from the file.Applications built with the framework can subclass Application and Document to suit specific needs. For example, a drawing application defines *DrawApplication* and *DrawDocument* subclasses; a spreadsheet application defines *SpreadsheetApplication* and *SpreadsheetDocument* subclasses. The relations between the classes are described in the Fig. 2.6 class diagram.

The abstract Application class defines the algorithm for opening and reading a document in its *OpenDocument* operation:

```
void Application::OpenDocument (const char* name)
{
    if (!CanOpenDocument(name))
```

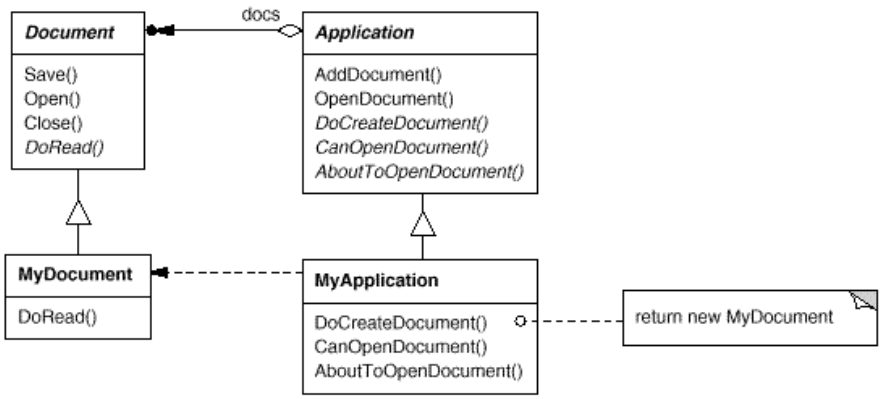

Figure 2.6: Example of applying TemplateMethod

```
    {
        // cannot handle this document
        return;
    }
    Document* doc = DoCreateDocument();
    if (doc)
    {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
```

OpenDocument defines each step for opening a document. It checks if the document can be opened, creates the application-specific Document object, adds it to its set of documents, and reads the Document from a file.

OpenDocument is a template method. A template method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior. Application subclasses define the steps of the algorithm that check if the document can be opened (*CanOpenDocument*) and that create the Document (*DoCreateDocument*). Document classes define the step that reads the document (*DoRead*). The template method also defines an operation that lets Application subclasses know when the document is about to be opened (*AboutToOpenDocument*), in case they care.

Template methods are a fundamental technique for code reuse. They are particularly important in class libraries, because they are the means for factoring out common behavior in library classes. It's important for template methods to

specify which operations are hooks (may be overridden) and which are abstract operations (must be overridden). To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding.

Decorator

Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.

For example, if we had a *TextView* object that displays text in a window. *TextView* has no scroll bars by default, because we might not always need them. When we do, we can use a *ScrollDecorator* to add them. Suppose we also want to add a thick black border around the *TextView*. We can use a *BorderDecorator* to add this as well. We simply compose the decorators with the *TextView* to produce the desired result. The following object diagram shows how to compose a *TextView* object with *BorderDecorator* and *ScrollDecorator* objects to produce a bordered, scrollable text view (Fig. 2.7).

The *ScrollDecorator* and *BorderDecorator* classes are subclasses of Decorator,

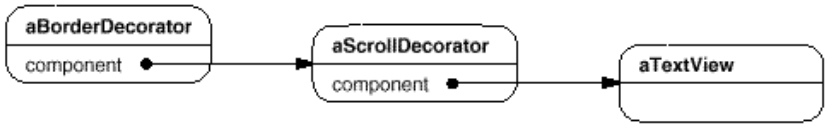

Figure 2.7: Example of composing

an abstract class for visual components that decorate other visual components (Fig. 2.8.

VisualComponent is the abstract class for visual objects. It defines their draw-

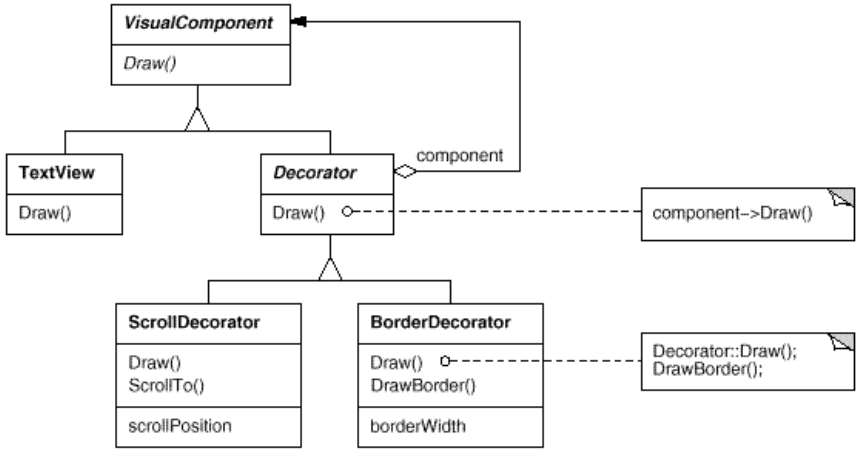

Figure 2.8: Example of applying Decorator

ing and event handling interface. The *Decorator* class simply forwards draw requests to its component and the *Decorator* subclasses can extend this operation.

Decorator subclasses are free to add operations for specific functionality. For example, *ScrollDecorator*'s *ScrollTo* operation lets other objects scroll the interface if they know there happens to be a *ScrollDecorator* object in the interface.

The important aspect of this pattern is that it lets decorators appear anywhere a *VisualComponent* can. That way clients generally can't tell the difference between a decorated component and an undecorated one, and so they don't depend at all on the decoration.

The structure of this pattern is described in the next class diagram (Fig. 2.9).

2.2 Tool Foundation

2.2.1 Introducing DuDe

The current paper is built on the foundation provided by [Wet04]. In order to be able to introduce the improvements proposed in this thesis, it is necessary to present the tool we started with. The tool is a Java program, called DuDe, aimed at automatic detection of code duplication. It uses string comparison to detect code clones, which is a language independent approach. Due to its

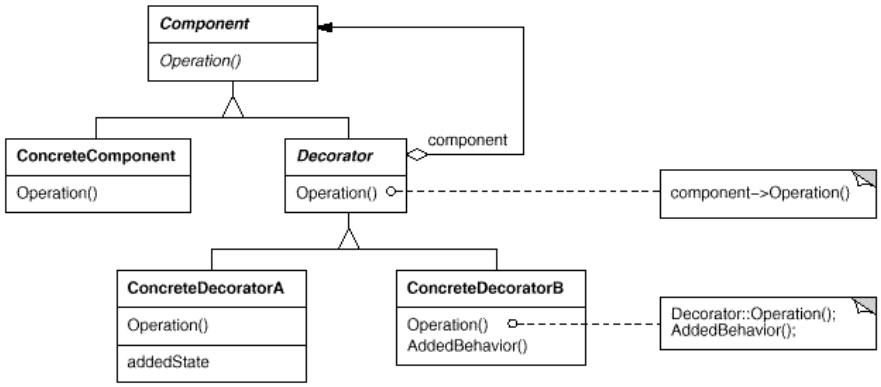

Figure 2.9: Structure of Decorator

high scalability (it has successfully analyzed projects up to 600 KLOC), it is appropriate to the analysis of industrial-size software systems.

Since the copying of a code fragment is often accompanied by modifications of the code needed to adapt it to the current problem (modification, insertions or deletions of statements), DuDe was built around the concept of code duplication chains, which will be presented next.

Target Audience

The target of DuDe's detection were exact clones (type 1 clones in [Bel02], which are identical fragments of code belonging to either two different files or to the same file in different places, and also duplication chains, meaning clones that have been affected by changes at the level of line of code (type 3 clones in [Bel02], which can be deletion, insertion or modifications of lines of codes.

2.2.2 Need for Duplication Chains

Imagine we have the two pieces of code (Fig. 2.10) with mainly the same task: to read some sensors and then to send those values to a display device (one to an LCD and the other to the console, for debugging reasons). This is a trivial example of scattered duplication belonging to a single duplicated block. The first piece of code has an extra line to initialize the LCD.

Because of the modified lines of code they could be detected as 3 exact clones, which is rather false. They belong to the same duplicated block, that can be easily refactored. One could rightly argue that there are approaches which can detect renamed variables. What happens if the lines of code are modified further than just variables or if there are lines appearing in only one of the two

code fragments? Even such a tool would find at least two smaller clones, that could be considered too small for refactoring.

```
initSensors(tSensors);
readSensors(tSensors);
lcd.init();
int i = 0;
while(i < tSensors.length){
 temp[i] =tSensors[i].getTemp();
 lcd.println("T"+i+"="+temp[i]);
 i++;
}
regulateTemp(temp);
```

```
initSensors(tSensors);
readSensors(tSensors);
int i = 0;
while(i < tSensor.length){
 temp[i] =tSensor[i].getTemp();
 System.out.println("T"+i+"="+temp[i]);
 i++;
}
regulateTemp(temp);
```

Figure 2.10: Scattered duplication

Thus, detected clones might not be relevant if they are too small or analyzed in isolation. Our main goal in[Wet04] was to capture, along with the usual clones, blocks of scattered clones that may have common origin, which will be further referred to as **duplication chains**.

2.2.3 Anatomy of a Duplication Chain

Duplication chain. A *duplication chain* can be a complex element (the representation of the duplicated code block), composed of a number of smaller, exact clones, separated pairwise by non-matching gaps.

Exact chunks. An *exact chunk* is a group of consecutive line-pairs that are detected as duplicated (is an exact copy). In the context of duplication chains, exact chunks are the non-altered parts of a duplicated block, that preserved its identity. An exact chunk appears in a scatter plot as a continuous diagonal, as it can be seen on the previous example's scatter plot representation (Fig. 2.11).

Non-matching gaps. A *non-matching gap* (Fig. 2.12) is a pair of code fragments made of completely different lines of code, located between two consecutive exact chunks of the same duplication chain. The two code fragments corresponding to a non-matching gap can have different sizes.

- same sizes - means there has been a modification of their constituting lines of code
- one of the fragments is missing - there has been a deletion/insertion of lines of code

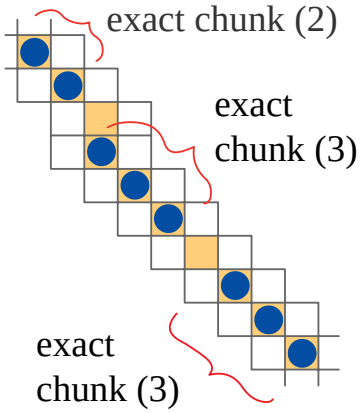

Figure 2.11: Exact chunks

This way, non-matching gaps of a duplication chain reflect the changes that have been made to the originating duplicated block. In a scatter plot representation, non-matching gaps appear as shortest non-marked paths linking two consecutive diagonals (exact clones). The position and distance between two consecutive exact chunks belonging to a duplication chain reflects the adaptation process of the duplicated block in that area, in terms of lines of code (insertion, deletion, modification). Thus, while apparently less important in clone detection, the non-matching parts around exact chunk provide some extra information about the adaptation process.

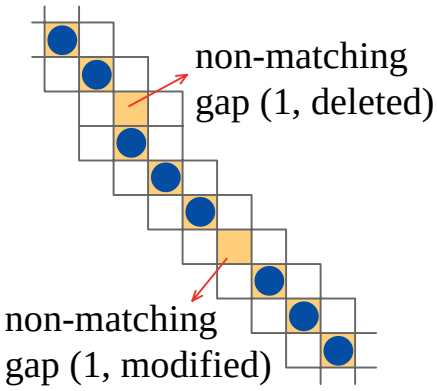

Figure 2.12: Non-matching gaps

Duplication chain type The *type* of a duplication chain is a summary of operations performed on the code of one code fragment in order to transform it

into the second code fragment. We identified four duplication chain types (Fig. 2.13):

- exact: is a exact chunk with no other exact chunks in its close vicinity (impossible to extend). This type is similar to a usual identical clone from other approaches, where no operations have been performed on the copied code

- delete/insert: composed of exact chunks linked by non-matching gaps left behind by insertions or deletions (only one of the gap's two fragments of code exists)

- modified: exact chunks linked by non-matching gaps in a configuration that reflects line modifications (the gap's two fragments have the same number of LOC)

- composed: the gaps between the exact chunks have more than one type of configurations, which suggests the duplicated block has been subject to a set of surgery operations (insertions, deletions, modifications)

Duplication chain size In the context of size, we want to capture those code fragments pairs that contain a *significant* amount of duplication. An exact clone is significant if the clone's size (in LOC) is larger than a threshold. A duplication chain is significant if the chain's size is large enough and if the components are in a certain relation.

Duplication chain signature. The *signature* is a textual fingerprint of the duplication chain's harmony, capturing its structural configuration in terms of exact chunks, non-matching gaps and the metrics around them.

In terms of the archeology metaphor, the signature could be associated with a "map" storing the places where all the related items where discovered. We believe that it might be possible to find a relation between signatures and types of problems associated to code duplication.

The signature is made of pairs letter-value, separated by dots. The significance of a letter is: E means exact chunk, and the value that follows it is the SEC, while I, D or M stands for the type of non-matching gap (Insert, Delete or Modified) and the value that follows it is the LB. A pair of letter-value that describes a non-matching gap is always located between 2 pair-letter values describing exact chunks. The signature shows that a duplication chain can be composed of one single exact chunk, or it can be composed of more duplication chunks. It always starts and ends with a duplication chunk (E).

The signature of the duplication chain is closely related to its type(Fig. 2.13). Although not the focus of this paper, we believe that the signature of a duplication chain can provide some additional information on the types of adaptations performed on its code.

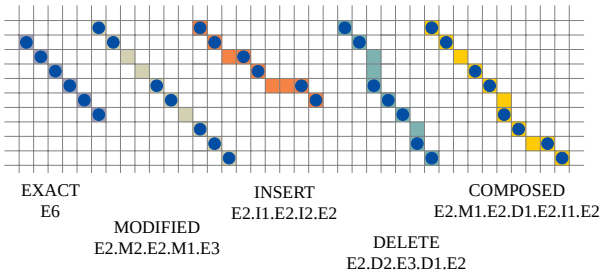

Figure 2.13: Types and signatures

2.2.4 Proportional Harmony

Since a duplication chain is made of both copied and non-matching lines of code and because terms like "close enough" and "significant amount" are vague and free to interpretation, we must set some rules that such a structure have to obey in order to be qualified for the title of duplication chain.

First, we will define some metrics related to a chain's properties:

1. Size of Exact Chunk (SEC) is measured in terms of LOC and is a key metric, because it reflects the degree of the granulation left behind by the adaptation phase of the copy-paste-adaptation process. Furthermore, this is closely related to how painful the refactoring to eliminate this duplication could be.

2. Line Bias (LB) is the size (in LOC) of the non-matching gap between two consecutive exact chunks. The LB value may allow us to decide if two exact chunks belong to the same duplicated block, since it provides a measure of distance between them. The lower the distance (LB), the higher the probability that the two exact chunk are part of the same duplication block and possibly the higher the refactoring potential.

3. Size of Duplication Chain (SDC) is the size (in LOC) of the more meaningful block of duplication, which actually suggests the magnitude of that duplication. In the particular case of an exact type duplication chain, SDC is the same as SEC.

A harmonious duplication chain will have:

- the SDC larger than a goal-specific threshold (minSDC). This condition makes sure that the total length of the duplication chain is large enough to qualify it as significant

- every LB (one for every two consecutive exact chunks) smaller than a threshold (maxLB). This will quantify the "neighborhood" aspect as it makes sure that the consecutive pieces of the chain are not too far from each other to be considered related

- every SEC larger than a threshold (minSEC). This measure will avoid detecting duplication chains containing "duplication crumbs", *i.e.*, very small duplicated code fragments

In the harmony context, there is a relation between SEC and LB: the SEC should always be larger that LB, because it is not desirable to detect duplication chains with gaps larger than its exact chunks.

There are no such things as perfect threshold values. Still, from our experience we found that following threshold values are adequate: minSDC = 8, maxLB = 2 and minSEC = 3. The minSDC of 8 is justified because we considered that significant duplication chains should be larger than the minimum configuration duplication chain of 2 exact chunks with SEC = 3, separated by a minimum length non-matching gap with a LB = 1 between them.

2.2.5 Detection of Duplication Chains

In [Wet04] we proposed an approach of lightweight line-matching, enhanced with the concept of chain duplication.

Phase 1: Code Preprocessing

The first phase starts with reading the source-files line by line, eliminating the white spaces, so that the various indentation styles would not make the difference. Then we eliminate noise (*i.e.*, lines of code made of syntactic elements like a single closing brace, or an **else** keyword or empty lines). An optional feature, and at the same time the only language-dependent part of our approach is ignoring the comments in the analysis process. This phase provides a set of relevant (noise free) lines of code in a raw form (without white spaces).

Phase 2: Populate the scatter-plot

As in the original scatter-plot approach, we compare every line of code (specifically, relevant code) with every line of code in the project (including itself). As a result of this comparison, the matrix will be divided in two symmetric areas, around the main diagonal, which is always completely marked (result of self comparison). With these facts in mind, we started to work only with one half of the matrix (excluding the diagonal), in order to avoid storing redundant information (it is enough to compare line X with line Y, we do not have to compare Y with X).

After comparing the lines of code with each other, we store the result in the matrix: the intersection of X (horizontal) and Y (vertical) will store the result of comparing line X of the matrix with line Y of the matrix (marked for match or unmarked otherwise).

Phase 3: Build the duplication chains

Starting with the left-upper matrix cell, we look for a marked one, in order to find a starting point for a possible duplication chain. If successfully, we start going on diagonal direction towards the lower-right cell. The algorithm will accept as a continuation of the chain either a marked cell that continues an exact chunk or a marked cell situated in the immediate vicinity, in order to merge the last exact chunk with the one starting with this specific cell. The algorithm will start with a 0 length line bias and increase its "sight" until it finds the next marked cell or until it reaches the maximum line bias, which puts an end to the chain. Also the sizes of the exact chunks in a chain must be greater than the minimum threshold set up before detection.

After closing a chain, we continue with the next cell after the one that started the previous chain.

When a duplication chain has reached the minimum accepted length, it will be stored in a duplication chain list which will be passed as a result of the detection process. A scatter plot representation, augmented with the compared lines is illustrated in a more complex example, an adapted duplication that summarizes all the types of operations on the lines of code (Fig. 2.14).

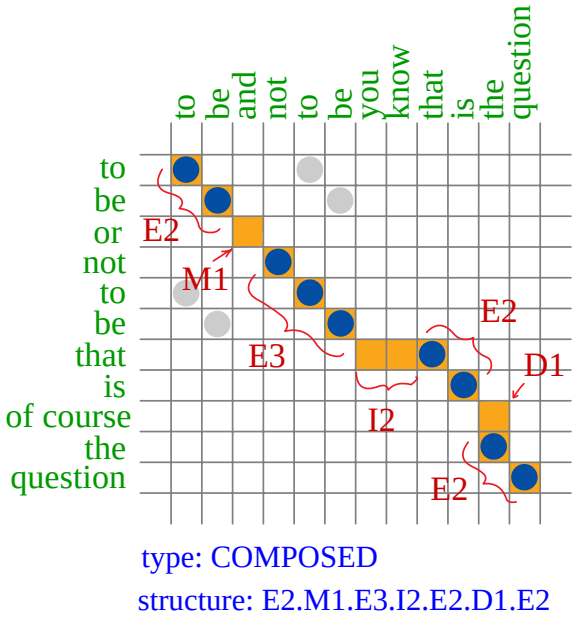

Figure 2.14: Complex duplication chain

Chapter 3

Approach

3.1 Clone Detection Improvements

3.1.1 The Idea

There are two levels of granularity where some type of code comparison appears:

1. a coarse granularity level, reached when comparing sequences of lines of code in order to build chains of code duplication
2. a finer granularity level, which comes up when comparing sequences of characters (lines of code) in order to mark the scatter-plot elements

The concept of chain of code duplication provides flexibility at the coarse granularity level, offering the means to detect type 3 clones (with modifications larger than renamed variables). At the finer granularity level, there was no flexibility in the previous approach [Wet04]. The lack of flexibility is caused by the rather large granularity of the comparison unit (line of code) in the context of type 2 clone detection (clones with renamed variables).

The straightforward means to enable the clone detector to find clones with renamed variables (constants) would be to establish a finer level of granularity (token) for the comparison. With this matching and by means of lexical information about the analyzed system it would be easy to determine variables and constants and to give them generic names. The high precision of this approach is obtained with the price of losing language independence (it needs at least lexical analyzers for every language supported).

Our approach to detect type 2 clones without giving up the language independence is to replace the exact matching technique used so far in order to determine the cloned code with an **approximate matching**. Intuitively, while the result of an exact-matching based comparison is the boolean representation of the answer to the question: "Are the two lines of code identical?", the result of an approximate-matching based comparison could be the answer to the

question: "How much are the two lines of code alike?" (similarity percentage). Using a minimum threshold of similarity, we would mark not only lines of code pairs that are identical, but also lines of code pairs that are "similar enough". Putting this in the scatter-plot metaphor context, this would mean establishing a grey-scale and marking the scatter-plot elements with the grey tones corresponding to the similarity degree of the intersecting lines of code. The threshold would be a filter eliminating the elements marked with a grey tone lighter than the minimum threshold tone.

This idea led us to the search of an appropriate metric that measures the similarity between two strings. The Hamming distance is used to determine the similarity between two strings of the same length, which does not seem appropriate for lines of code, which can cover various lengths. Another important metric is the Levenshtein distance, which is further discussed.

3.1.2 The Levenshtein Distance

The Levenshtein distance between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution. It is named after the Russian scientist Vladimir Levenshtein, who considered this distance in [Lev66]. It is useful in applications that need to determine how similar two strings are, such as spell checkers (it is also called edit distance).

A commonly-used bottom-up dynamic programming algorithm for computing the Levenshtein distance involves the use of an (n + 1) (m + 1) matrix, where n and m are the lengths of the two strings. Here is pseudocode for a function LevenshteinDistance that takes two strings, **str1** of length lenStr1, and **str2** of length lenStr2, and computes the Levenshtein distance between them:

```
int LevenshteinDistance(char str1[1..NOChr1], char str2[1..NOChr2])
   // d is a table with NOChr1+1 rows and NOChr2+1 columns
   int d[0..NOChr1, 0..NOChr2]
   // i and j are used to iterate over str1 and str2
   int i, j, cost

   for i from 0 to NOChr1
       d[i, 0] := i
   for j from 0 to NOChr2
       d[0, j] := j

   for i from 1 to NOChr1
       for j from 1 to NOChr2
```

```
            if str1[i] = str2[j] then cost := 0
                                 else cost := 1
            d[i, j] := minimum(
                                 d[i-1, j  ] + 1,     // insertion
                                 d[i  , j-1] + 1,     // deletion
                                 d[i-1, j-1] + cost   // substitution
                               )

    return d[NOChr1, NOChr2]
```

The complexity of the algorithm is O(m×n), where n and m are the length of str1 and str2. The Levenshtein distance is the value of the element positioned at the intersection of the last row with the last column and this implies that in order to calculate de Levenshtein distance, one would have to calculate the whole matrix, which can be a serious time consuming process.

An example of the matrix built for the calculation of the Levenshtein distance is Fig. 3.1. The two compared strings are *source* and *voice* and the resulting distance of 3 is the value of the element in the right, down corner.

		-1	0	1	2	3	4
			v	o	i	c	e
-1		0	1	2	3	4	5
0	s	1	1	2	3	4	5
1	o	2	2	1	2	3	4
2	u	3	3	2	2	3	4
3	r	4	4	3	3	3	4
4	c	5	5	4	4	3	4
5	e	6	6	5	5	4	3

Figure 3.1: Example of Levenshtein distance matrix

3.1.3 Levenshtein-Based Similarity

Since the Leveshtein distance tells us only how many operations would be needed to transform one string to another, the lengths of the strings have no influence on the result. Besides from being counterintuitive in the similarity context (the distance between identical strings is 0), the similarity between two strings (not necessarily of the same length) should also consider their lengths. Therefore we define the **Levinshtein distance based similarity** as follows:

$$Sim_L[\%] = \left(1 - \frac{LD}{max(NOChr_1, NOChr_2)}\right) \cdot 100 \tag{3.1}$$

where LD is the Levenshtein distance, while $NOChr_1$ and $NOChr_2$ are the lenghts (in number of characters) of the two strings being compared.

Despite the fact that the Levenshtein distance is a measure of the degree of similarity between two strings, in the particular case of comparing two lines of code (viewed as simple strings) there are some reasons that make it less appropriate:

1. Using the Levenshtein distance for every pair of lines of code in the system instead of the exact matching could prove a very expensive computational effort. Such a serious time overhead could lead to scalability problems

2. An even more important drawback of this approach would be the relevance of computing the Levenshtein distance, which is defined as the minimal number of *characters* one have to replace, insert or delete to transform one string into the other. This issue will be further illustrated by an example and a counterexample.

Example: let's consider the two lines of code:

```
for(int i=0; i<100; i++) tab[i]=0;                    (a)
for(int x=0; x<100; x++) tab[x]=0;                    (b)
```

We calculate the similarity degree based on the formula (3.1):

$$Sim_L[\%] = \left(1 - \frac{4}{34}\right) \cdot 100 = 88.23 \tag{3.2}$$

Counterexample: instead of the (b) line of code from the previous example, let's compare (a) with (c), which we will represent - out of page layout reasons - on more than one row (although in reality it is on a single row):

```
for(int aLargerVariableReplacingI = 0;
        aLargerVariableReplacingI < 100;
        aLargerVariableReplacingI++)
            tab[aLargerVariableReplacingI]=0;         (c)
```

Based on the formula (3.1), the similarity degree between these 2 lines of code is:

$$Sim_L[\%] = \left(1 - \frac{96}{130}\right) \cdot 100 = 26.15 \tag{3.3}$$

From a lexical point of view, the two examples presented are practically identical (a variable has been renamed). However, the measured degrees of similarity are radically different. The counterexample demonstrates that in certain cases, this similarity measurement method can provide unprecise results. For example, if we would have set the minimum threshold to be 80%, (a) and (b) would have been detected, but not (a) and (c), or (b) and (c), due to the fact that the variable in (c) has a very high length.

3.1.4 Token-Level Similarity

The main problem regarding the Levenshtein distance based similarity is the too fine granularity considered when comparing strings. To make it more appropriate to our declared goal, we needed to raise this granularity: instead of a sequence of characters, we chose to consider the string as a sequence of tokens. Usually, the symbols of programming languages are separated from each other by white spaces, hence we could be able to separate them without any lexical knowledge. However, many of the programming languages (*i.e.*, C, Java) allow one to put together such lexical elements (for example, the *i++* statement).

In order to extract the sequence of tokens from a line of code, while keeping the language independence to a minimal level, we defined a filter based on regular expressions, which is aware of lexical elements common to many programming languages that can stay glued to other elements (separators, operators). In the preprocessing phase of the analyzed code, by means of this filter, the lexical elements in every line of code will be separated by white space from its predecessor and successor lexical elements. For example, the following line of code:

```
for(int i=0;i<100;i++)   tab[i]=0;
```

after preprocessing will look like this:

```
for ( int i = 0 ; i < 100 ; i ++ ) tab [ i ] = 0 ;
```

Every line of code put in such a form will be quite easy to transform into a sequence of tokens (lexical elements), without having any knowledge weather a token is a variable, a numerical value or an operator.

This adapted version of the Levenshtein distance is given by the number of operations with tokens needed to transform a sequence of tokens (line of code) into the other sequence of tokens. The possible operations with tokens are modification of tokens (possible renaming of variables or constants), deletions and insertions of tokens. Raising the granularity could focus on the transformations

that really happen when copying, pasting and modifying code, since the programmer operates with lexical elements rather than with simple characters.

The formula in this case is similar:

$$Sim_{L_T}[\%] = \left(1 - \frac{TLD}{max(NOTok_1, NOTok_2)}\right) \cdot 100 \tag{3.4}$$

where TLD is the Token-level Levinshtein-based Distance, while $NOTok_1$ and $NOTok_2$ are the lengths (in number of tokens) of the two compared strings.

Applying this formula on the previously presented example and counterexample, one can observe that the resulting similarity degrees are equal, since both **i** and **aLargerVariableReplacingI** represent one lexical element (a variable, in this particular case), no matter how many characters they are composed of:

$$Sim_{L_T}[\%] = \left(1 - \frac{4}{21}\right) \cdot 100 = 80.95 \tag{3.5}$$

3.1.5 Matching Strategies

The two distinct matching types based on the similarity degree measurement together with the exact matching from the early versions of DuDe were implemented as matching strategies. DuDe will allow the user to select one of this strategies based on the purpose of the analysis, the size of the analyzed projects (the matching strategies scale differently).

3.2 System architecture

The architecture of the enhanced clone detecting engine is described in the UML class diagram (Fig. 3.2). In the middle of the diagram stands the *Processor* class, that is the main character behind the duplication chains detection process. A Processor object works with *Entity* objects as input, in order to be dependent on an abstract entity (in conformance to Dependency Inversion Principle). The classes that specialize *Entity* are *SourceFileEntity*, which is the input for the stand-alone version of DuDe and *MethodEntity*, which is the input provided by the Insider reengineering platform, in DuDe's integrated version. A *DirectoryReader* object reads recursively all the files in a directory and creates the corresponding *SourceFileEntity* objects, passing them to the *Processor*.

The improvements necessary to add support for type 2 clones is mainly in

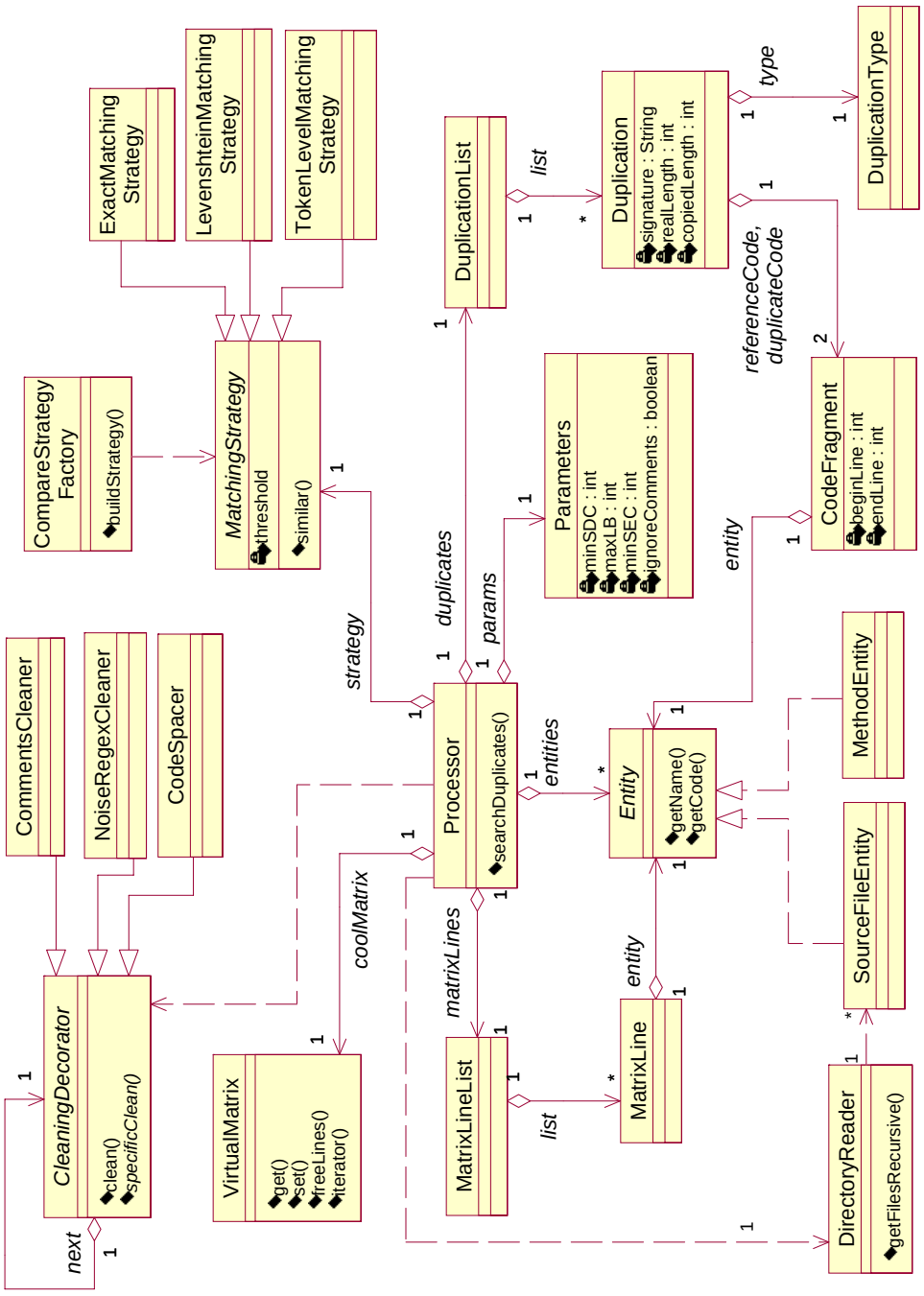

Figure 3.2: System architecture (UML)

the *MatchingStrategy* abstract class and the concrete strategies that implement it. They are used by the *Processor* to take decisions in wether to mark or not a certain matrix (scatter-plot) cell. Depending on the concrete matching strategy (*ExactMatchingStrategy*, *LevenshteinMatchingStrategy* or *TokenLevelMatchingStrategy*) the method that measures if the two compared strings are "similar enough" is differently computed for each of them. This is an implementation of the Template Method design pattern, hence the concrete strategies are different only when it comes to their implementation of the **similar(String s1, String s2)** method. While the *ExactMatchingStrategy* returns a positive result only if the two strings are identical, the other two approximate matching strategies compute the similarity based on the distances (classic Levenshtein and the token-level approach), and compare it with the threshold. Two strings are similar enough, thus their corresponding matrix cell gets marked, only if the computed similarity is higher than the threshold.

The expected output of the detection process is a *DuplicationList* object, that contains all the duplication chains found by the *Processor* (class *Duplication*).

The *VirtualMatrix* is the element that brings the efficient management of memory and makes possible the dividing of the matrix into areas. When a matrix zone is finished, the area is cleaned from the memory, by calling its freeLines(int row) method. The internal structure of the *VirtualMatrix* can provide an Iterator for every row (every row is a HashMap), which makes the chain building way more rapid, after having the markings in the scatter-plot.

The *CleaningDecorator* is a combination of the **Decorator** and **Chain of Responsibility** design patterns [GHJV95], which implements a class that cleans the code (brings it to a relevant form). If we will need another cleaner, we have to create a new class that extends the *CleaningDecorator* abstract class, and we can add it in the "chain" of cleaners. This way, it is possible to dynamically combine different cleaners. For the new approach, where we need to divide lines of code into tokens, we replaced the old *WhiteSpaceCleaner* with a new filter called *CodeSpacer*, that will surround operators and separators with white spaces and in the end will eliminate multiple successive whitespace. We also replaced the old *NoiseCleaner* with a new filter caller *NoiseRegexCleaner*, which eliminates unwanted noise based on a list of regular expressions defining the noise. If the *ignoreComments* option is ON, than the processor uses a chain of cleaners made of: a comments cleaner, a codespacer and a noise regex cleaner. The *Processor* object works with a CleaningDecorator, which is an abstract class, this way the heuristic: *program to an interface, not to a implementation* is fully respected. The sequence diagram (Fig. 3.3) describes the code cleaning process in terms of time.

When the *Processor* calls the cleaner's *clean()* method, the message goes to the first cleaner in the chain. The CommentCleaner cleans the comments off (by calling it's own *specificClean()* method) and then calls the *clean()* method

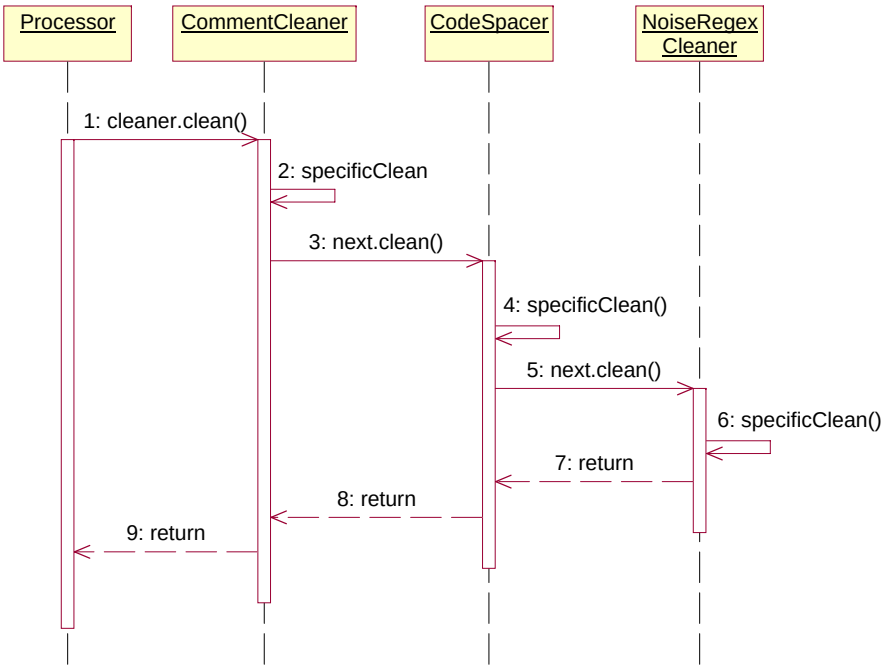

Figure 3.3: Code cleaning process (sequence diagram)

of the next cleaner in the chain (he does not need to know what type of cleaner is next, thanks to polymorphism). The next cleaner, a CodeSpacer surrounds the separators and operators with white space and cleans the extra white space and then further delegates the next cleaner to clean the code by calling it's *clean* method. The last cleaner (NoiseRegexCleaner) cleans the lines considered noise (which are defined in the noise.regex file, by means of the flexible mechanism of regular expressions) and then returns, because it is the last in the chain (it does not have a *next* cleaner). Step-by-step, the clean code return to the *Processor* object.

In order to loosen the coupling between the *Processor*, the various *Importer* instances on the one hand and the controller of the graphical user interface (Fig. 3.4) on the other hand, we used the **Observer** design pattern. While the *GUI* is a concrete observer and has knowledge of the concrete *Subject*s it works with, the concrete *Subject*s (*Importer* and *Processor*) are not aware to whom they are providing the results of their work. The *Importer* and *Processor* are concrete *Subject*s, since they usually need time to provide results, while the *Exporter* does not provide results other than the confirmation that the data has been successfully exported. For the export and import features, there are at the moment two formats available: the XML format, chosen as the standard format and a proprietary format named DUP, that has been used for experimental

purposes to compare DuDe's results with the results of the tools analyzed in [Bel02].

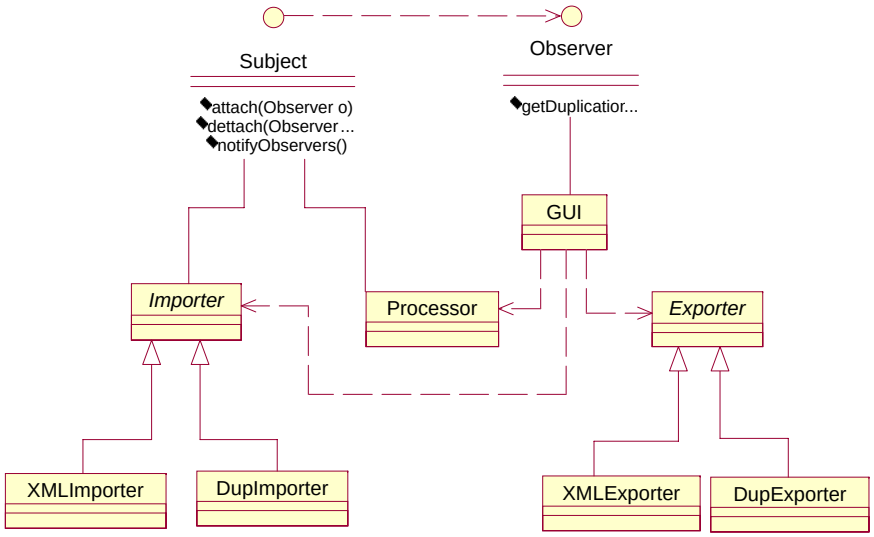

Figure 3.4: Data exchange within DuDe

3.3 Data Exchange

One feature that we missed in the previous versions of DuDe was the possibility to import results and settings from another analysis session. The need for this was obvious in particular with large systems, where the detection process would take many hours. From here it would be very useful if we could save this data and restore the session someday without having to run the detection process again. Furthermore, exporting the data in a file could make it usable by other analysis tools.

3.3.1 XML Parsing

Due to the fact that a large number of software vendors have adopted the XML standard and since there are already many libraries for the manipulation of data within XML files, we chose this format as the main data exchange format for DuDe. Moreover, an XML document with elements and attributes chosen, based on the business domain, proves easy to read not to machines only, but

also to humans.

The Extensible Markup Language (XML) is a simple, very flexible text format, designed to describe data. Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. XML with an XML Schema is designed to be self-descriptive.

Over time, there have been implemented many mechanisms to access XML documents, form which two major models (APIs) will be mentioned here:

- The SAX (Simple API for XML) model allows for simple parsers by allowing parsers to read through a document in a linear way, and then to call an event handler every time a markup event occurs (the parser encounters different entities within the XML document). This is the protocol that most servlets and network-oriented programs will want to use to transmit and receive XML documents, because it's the fastest and least memory-intensive mechanism that is currently available for dealing with XML documents.

- The DOM (The Document Object Model) API is based on an entirely different model of document processing than the SAX API. Instead of reading a document one piece at a time (as with SAX), a DOM parser reads an entire document. It then makes the tree for the entire document available to program code for reading and updating. At the core of the DOM API are the Document and Node interfaces. A Document is a top level object that represents an XML document. The Document holds the data as a tree of Nodes, where a Node is a base type that can be an element, an attribute, or some other type of content.

Simply put, the difference between SAX and DOM is the difference between sequential, read-only access, and random, read-write access. Since DuDe is an intensive memory consumer and the data amount it export/import can reach high quantities, we chose the SAX model over the DOM model. In the case of large files, DOM is less appropriate due to the fact that it loads the whole document right from the start in order to build its tree. SAX , with its serial access mode would serve our purpose better.

3.3.2 Data Format

We will present the format of the exported/imported data by means of its XML schema. An XML schema defines the content model (also called a grammar or vocabulary) that its instance documents will represent. The schema is used by the XML parsers to check an XML document for well-formedness and validity. In order to gain an overall view of DuDe's XML schema for exchange files, we

will first present an intuitive diagram (Fig. 3.5). With this self-explanatory diagram in mind, we will briefly explain the XSD file (XML Schema Definition), as a means to describe the exchange data.

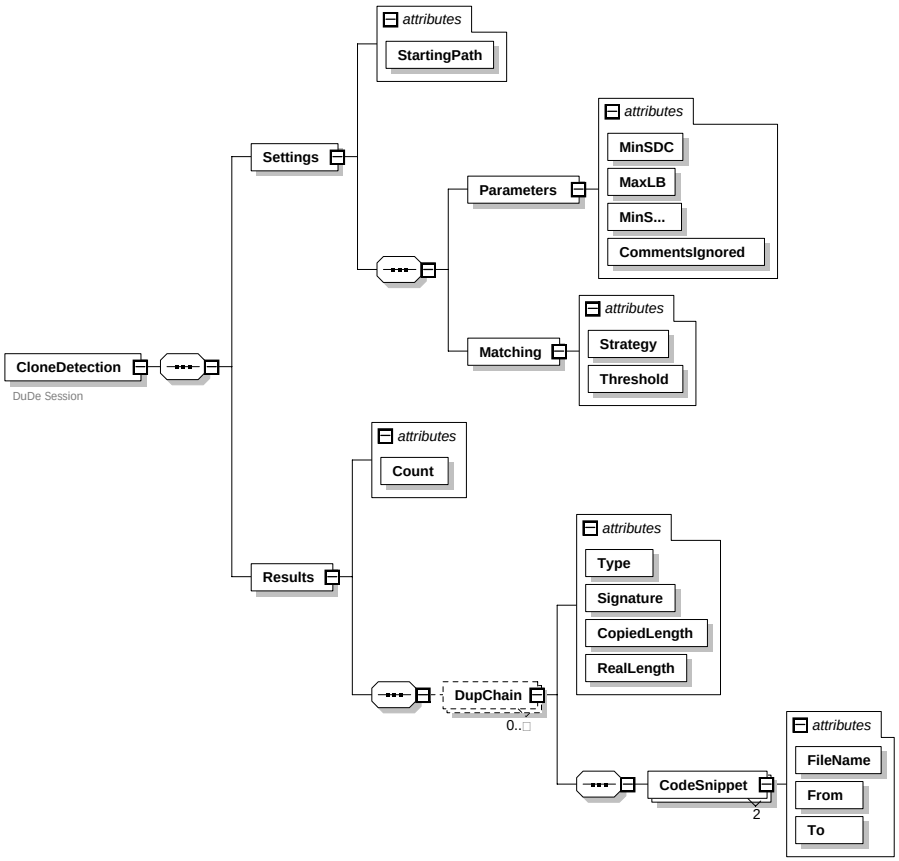

Figure 3.5: Diagram of the XSD file

The root element, which as in every XML document contains all the data, is **CloneDetection**, representing a clone detection session. The data DuDe is exchanging, for a detection session, is made of the configuration, represented by the Settings element and the detected duplication chains, which are comprised by the Results element, as seen in Fig. 3.6.

The **Settings** element contains all the tunings made to the tool in order to obtain exactly the same results, assuming we have access to the source code of the analyzed project. The configuration of the tool is an important fact, since it assures the repeatability of the experiments. The Settings element has an attribute named *StartingPath*, which is used to set the path to the project

```
<?xml version="1.0" encoding="UTF-8" ?>
- <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
  - <xs:element name="CloneDetection">
    - <xs:annotation>
        <xs:documentation>DuDe Session</xs:documentation>
      </xs:annotation>
    - <xs:complexType>
      - <xs:sequence>
        - <xs:element name="Settings">
          - <xs:complexType>
            - <xs:sequence>
              + <xs:element name="Parameters">
              + <xs:element name="Matching">
              </xs:sequence>
              <xs:attribute name="StartingPath" type="xs:string"
                use="required" />
            </xs:complexType>
          </xs:element>
        - <xs:element name="Results">
          - <xs:complexType>
            - <xs:sequence>
              + <xs:element name="DupChain" minOccurs="0"
                  maxOccurs="unbounded">
              </xs:sequence>
            - <xs:attribute name="Count" use="required">
              - <xs:simpleType>
                - <xs:restriction base="xs:int">
                    <xs:minInclusive value="0" />
                  </xs:restriction>
                </xs:simpleType>
              </xs:attribute>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
```

Figure 3.6: The XSD file, not fully expanded

directory and two elements: Parameters and Matching.

The **Matching** element illustrated in Fig. 3.7 is related to the way the strings are being compared and it has two attributes: *Strategy* (the matching strategy) and *Threshold* (which is the minimum value of the similarity degree needed in order to be considered "similar enough"). The similarity threshold, which

is expressed as a percentage, has a *maxInclusive* constraint of 100, while the strategy has to be one of the enumerated values (Exact Matching, Token Level Matching, Levenshtein Matching), which are all possible names of strategies. Another value for the strategy would lead to file invalidation.

```
- <xs:element name="Matching">
  - <xs:complexType>
    - <xs:attribute name="Strategy" use="required">
      - <xs:simpleType>
        - <xs:restriction base="xs:string">
            <xs:enumeration value="Exact
              Matching" />
            <xs:enumeration value="Token Level
              Matching" />
            <xs:enumeration value="Levenshtein
              Matching" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    - <xs:attribute name="Threshold" use="required">
      - <xs:simpleType>
        - <xs:restriction base="xs:int">
            <xs:minInclusive value="0" />
            <xs:maxInclusive value="100" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
```

Figure 3.7: The Matching element, fully expanded

The **Parameters** element, has four attributes representing the concrete parameters used in the clone detection process: *minSDC* (minimum size of duplication chains), *maxLB* (maximum line bias), *minSEC* (minimum size of exact chunks) and *CommentsIgnored*. Each of the parameters belong to a data type (int, boolean) and can be further controlled by constraints. For example, the minSDC attribute has a minInclusive constraint of 1 (a duplication chain of size 0 makes no sense), while the maxLB has the same constraint with value 0 (we want to be able to detect clones with no gaps). All these attributes are required (use=”required”) in order to have a valid XML exchange file.

After importing data from an XML file, besides the checking and setting of the starting path, the parameters are automatically set as the ones imported (including the matching strategy) and the user is one-click-away (the Search button) from repeating the experiment.

The **Results** (Fig. 3.6) element has one attribute named *Count*, which represents the number of duplication chains found in the clone detection session. This information is used to present the progress of the task (the XML parsing) to the user, by displaying a progress bar. Otherwise, there would be impossible to guess the number of clones in an event-based parsing (SAX). The Results element has a number of **DupChain** elements, with a constraint indicating a minimum of 0 occurs (minOccurs="0") and no upper bound (maxOccurs="unbounded").

Every DupChain element has four attributes *Type*, *Signature*, *CopiedLength* (number of relevant lines of code that have been copied) and *RealLength* (the total number of lines of code on which the duplication chain extends, including blank lines or lines with comments). In addition to that, every DupChain element has precisely two **CodeSnippet** elements (minOccurs="2" maxOccurs="2"), which define the exact location of the involved code, by means of the three attributes: *FileName*, *From* (starting line of code) and *To* (ending line of code).

An example of an XML file, conforming to the presented schema is illustrated in Fig. 3.8. While it is a precise, machine readable document, it is also easy to read by the maintenance engineer.

Validation

Validation is the process of verifying that an XML document is an instance of a specified XML schema. While parsing the XML document, the parser signalizes every abnormality: unsatisfied constraints, missing elements or attributes. The schema, besides the fact that avoids the coding of various tests in the parser, assures that the data exported by another tool is valid to import in DuDe and viceversa.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<CloneDetection>
  <Settings StartingPath="c:\home\ricky\cs\eclipse-ant">
    <Parameters MinSDC="8" MaxLB="2" MinSEC="3" CommentsIgnored="true"/>
    <Matching Strategy="Tokenized Distance" Threshold="80"/>
  </Settings>
  <Results Count="8">
    <DupChain Type="DELETE" Signature="E5.D1.E6.D2.E10" CopiedLength="21" RealLength="52">
      <CodeSnippet FileName="src\ant\BuildEvent.java" From="81" To="138"/>
      <CodeSnippet FileName="src\ant\BuildEvent.java" From="152" To="203"/>
    </DupChain>
    <DupChain Type="EXACT" Signature="E24" CopiedLength="24" RealLength="25">
      <CodeSnippet FileName="src\ant\DirectoryScanner.java" From="239" To="263"/>
      <CodeSnippet FileName="src\ant\DirectoryScanner.java" From="299" To="323"/>
    </DupChain>
    <DupChain Type="MODIFIED" Signature="E5.M1.E5.M1.E11" CopiedLength="23" RealLength="27">
      <CodeSnippet FileName="src\ant\DirectoryScanner.java" From="339" To="365"/>
      <CodeSnippet FileName="src\ant\DirectoryScanner.java" From="460" To="489"/>
    </DupChain>
    <DupChain Type="COMPOSED" Signature="E3.M1.E5.M1,I1.E3.M1,I1.E3" CopiedLength="17" RealLength="20">
      <CodeSnippet FileName="src\ant\IntrospectionHelper.java" From="351" To="370"/>
      <CodeSnippet FileName="src\ant\IntrospectionHelper.java" From="381" To="405"/>
    </DupChain>
    <DupChain Type="EXACT" Signature="E20" CopiedLength="20" RealLength="23">
      <CodeSnippet FileName="src\ant\taskdefs\compilers\DefaultCompilerAdapter.java" From="154" To="176"/>
      <CodeSnippet FileName="src\ant\taskdefs\rmic\DefaultRmicAdapter.java" From="141" To="163"/>
    </DupChain>
    <DupChain Type="MODIFIED" Signature="E9.M1.E3.M1.E3" CopiedLength="17" RealLength="17">
      <CodeSnippet FileName="src\ant\taskdefs\GenerateKey.java" From="186" To="202"/>
      <CodeSnippet FileName="src\ant\taskdefs\GenerateKey.java" From="208" To="224"/>
    </DupChain>
    <DupChain Type="MODIFIED" Signature="E7.M1.E3.M1.E8" CopiedLength="20" RealLength="20">
      <CodeSnippet FileName="src\mail\MailMessage.java" From="247" To="266"/>
      <CodeSnippet FileName="src\zip\ZipEntry.java" From="160" To="179"/>
    </DupChain>
    <DupChain Type="DELETE" Signature="E5.D2.E5.D2.E8" CopiedLength="18" RealLength="48">
      <CodeSnippet FileName="src\zip\ZipLong.java" From="70" To="121"/>
      <CodeSnippet FileName="src\zip\ZipShort.java" From="70" To="117"/>
    </DupChain>
  </Results>
</CloneDetection>
```

Figure 3.8: Exported XML file

Chapter 4

Evaluation of the tool

4.1 Features

The graphical user interface (Fig. 4.1) offers a simple, yet powerful access to the duplication chains detecting engine. The integrated workspace is composed of:

- Menu bar that provides access to the whole set of commands, arranged in a hierarchical structure.
- Toolbar, that provides buttons for the most used commands of the menu bar.
- Configuration panel, accessible through the **Set parameters** command or button. This settings panel provides control of both the parameters of the detection process and the matching parameters (strategy and similarity threshold).
- Results panel, consisting of a various modes sortable list of **Duplication chains**
- Visualization panel, called **Code Viewer** for visual analysis of the duplicated code involved in a duplication chain.
- Status bar, where the user is provided with brief information about the success or failure of the current task and the progress status, by means of a progress bar, visible when needed.

In order to analyze a project, first you have to set the starting path, where the source files of the project are located. Then, you can modify the searching parameters, the matching strategy and the threshold and hit the **Search** button. The searching process can be stopped at any time, by hitting the **Stop** button. The status bar contains a progress bar, visible only during a search or import process.

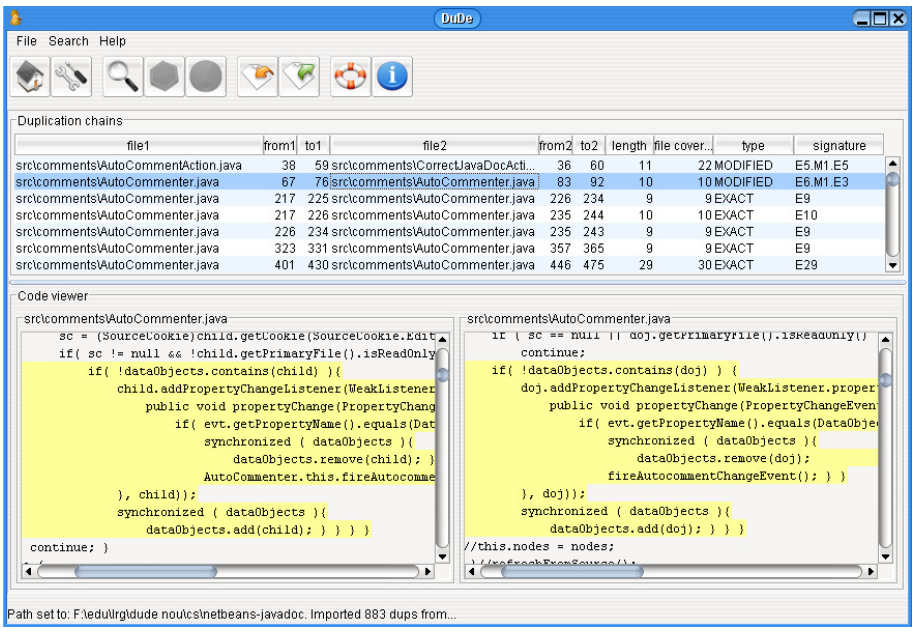

Figure 4.1: DuDe's Graphical User Interface

After the searching is over, if any duplication chains were found, they will be shown in a list of chains which can be sorted by any of: entity's name, index to the first or the last line of code in the chain, length, type etc.

The meanings of the different columns in the results table are:

- **file1**, **file2** are the 2 files that share the duplicated code
- **from1**, **to1** and **from2**, **to2** are the line indexes in the 2 files where the duplicated code starts and ends
- **copied length** is the length of the duplication chain
- **file coverage** is the number of duplicated lines of code between the start line and the end line of the duplication. It is usually greater than the copied length, because it may contain the noise and blank lines that were rejected in the code cleaning phase, just before the analysis
- **type** of the duplication chain is the chain's type (previously discussed)
- **signature** is a chain of <symbol><size> elements (separated by '.'), where size represents the number of lines and the symbols can be: E (exact) for exact chunks and M (modified), I (insert) or D (delete) for gaps, or a combination of M and I/D, separated by ',' for combined gaps.

In order to validate the duplication chains or to examine the results in a visual manner, a mouse click on any of the items in the results list will display the contents of the 2 files involved in that duplication in the **Duplicate Viewer** panel, with the replicated code highlighted in yellow.

A useful feature offered by the tool is the possibility to consult a set of statistical data (Fig. 4.2) gathered during the last searching operation:

- number of analyzed entities,
- total number of lines of code
- number of relevant lines of code (clean lines of code)
- number of duplicated lines (the ones that are part of at least one duplication chain),
- the percentage of duplicated lines, a metric called coverage
- the number of found duplication chains
- elapsed time

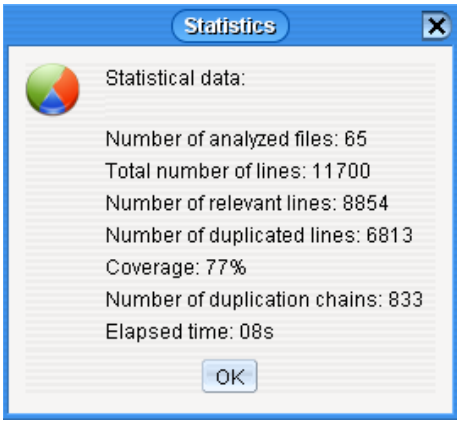

Figure 4.2: Statistical Report

At the bottom of the window, there is a status bar, where the user will be given information about operations (how many duplication chains were found during a searching operation, current starting path or saving results).

In order to save a successful clone detection session, the user can click the **Export to XML** button to export the data in an XML file, which can be later on imported, a task that is less time consuming than the searching itself. This command will save not only the results, but also the current configuration of the tool.

The **Import from XML** gives the maintenance engineer the possibility to:

- continue analyzing the results of a successful clone detection session, by directly using the imported results
- repeat a clone detection experiment, without the need to separately set the configuration parameters, these being automatically set to the original values during the import process.

4.1.1 Tunable Parameters

The configurability of the tool is reflected by the parameters related to the proportional harmony and the to the matching strategy:

- minLength: minimum accepted length for the duplication chains (in LOC). It defines a filter for the searching operation, which will eliminate the duplication chains considered irrelevant for the current case study.
- maxLineBias: the maximum size of non-matching gaps, or the line bias (number of modified, inserted or deleted lines) between 2 exact chunks within a duplication chain.
- minExactChunk: minimum accepted size of the exact chunks within a chain.
- ignoreComments: the duplication searching engine can include the comments in its analysis or not (optional, only for C,C++ and Java comments)
- Matching Strategy, can be selected one of: Exact Matching, Levenshtein Matching and Tokenized Matching
- Similarity Threshold, is a value between 50 and 100 which is the minimum accepted similarity degree, and can be modified just for the approximate matching strategies (it is 100 for the exact matching strategy)

4.2 Experiment

The primary goal of this experiment is to see how the approximate matching strategies compare with each other in order to validate the presumption that the tokenized one provides more relevant results and is more scalable.

The experiment's secondary goal was to compare the approximate matching strategies with the exact matching, in terms of time performance and scalability, number of detected clones and coverage (ratio between duplicated LOC and

the total LOC).

By means of selecting the matching strategy presented in 3.1.5), the different approaches could be tested with the same working conditions.

4.2.1 Experimental Setup

We choose 4 small to medium size projects written in Java and C that were study cases in Bellon's paper on evaluation of clone detecting tools [Bel02]. The 4 projects, covering the size range from up to 3 MB, are presented in Fig. 4.3. The Java projects are in the higher half of the table, while the C projects are located in the lower half of the table, so that we can later quickly make the distinction between them. The C projects and the Java projects are sorted by size (or number of lines). The **LOC** column is the number of lines of code, while **Relevant LOC** is the number of lines of code excluding the comments and the blank lines (we refer them as relevant lines of code) and this is actually the set of lines of code that DuDe analyzed.

Project name	Language	No. of Files	Size. (KB)	LOC	Relevant LOC
netbeans-javadoc	Java	101	708	14360	8245
eclipse-ant	Java	178	1503	34744	14780
weltab	C	53	450	11591	9234
cook	C	590	2814	80408	49041

Figure 4.3: Test projects

The experiment was conducted on a Pentium IV 2.8 GHz and 512 MB RAM machine running Windows XP. The configuration of the tool was set to: minSDC = 8, maxLB = 2, minSEC = 3, ignoreComments = on. For every project, the tool ran three times (once for every matching strategies). The similarity threshold was set to 80 % for the approximate matching strategies.

4.2.2 Interpretation of the Experiment's Results

The results are concentrated in three tables. The first table (Fig. 4.4) presents the time performance of the tool using the different matching strategies. While, with the exact matching, all the projects were scanned within seconds, with the longest one in under 2 minutes, the time performance with the approximate matching strategies falls dramatically. Compared to the exact matching strategy, the Levenshtein distance based matching strategy is between 600 and 1600 times slower, while the token level matching strategy is between 200 and

300 times slower. This is hardly a precise result, because the exact matching strategy depends purely on the number of relevant lines of code that will be compared, while the approximate matching strategies depend on both the number of relevant lines and the length (in characters or tokens) of the lines of code. While for the exact matching strategy, the comparison ends by the first non-matching pair of characters, each of the approximate matching strategies builds *the whole* matrix for every comparison. However, some time related conclusions we can draw are:

1. The approximate matching strategies are way slower than the exact matching strategy

2. Between the two approximate matching strategies, the Token-level Matching is 3 to 6 time faster than the Levenshtein Matching.

	Processing time (hh:mm:ss)		
Project name	**EM**	**LM**	**TM**
netbeans-javadoc	00:00:03	01:20:46	00:12:47
eclipse-ant	00:00:10	02:40:48	00:36:03
weltab	00:00:03	00:45:57	00:15:23
cook	00:01:19	14:09:12	04:33:00

Figure 4.4: Time performance

The next two tables (Fig. 4.5) present two closely related metrics: the number of detected clones and the coverage. We will analyze these two tables correlated, because there are interesting things that can be inferred from the correlation of the two, which can be missed when analyzing them separately.

Judging by the number of detected clones with the Exact Matching Strategy (**EM**), it seems that the C projects chosen have way more duplicaton than the Java projects.

However, the **netbeans-javadoc** Java project analyzed with the Tokenized Matching (**TM**) has many duplications, that are probably clones with renamed variables (883 compared to only 51), while at the same time the coverage does not grow so sudden (only 38 % compared to 16 %). A possible explanation for this discrepancy comes from the families of clones, where the discovery of another duplicated code fragment (which slightly increases the coverage) would bring a high number of clones (the new code fragment with, sequentially every member of the clone family).

	Number of detected clones		
Project name	**EM**	**LM**	**TM**
netbeans-javadoc	51	392	883
eclipse-ant	43	108	130
weltab	793	2139	1777
cook	2721	5537	6870
	Coverage (%)		
Project name	**EM**	**LM**	**TM**
netbeans-javadoc	16	33	38
eclipse-ant	5	11	12
weltab	77	83	82
cook	19	35	32

Figure 4.5: Number of Clones vs Coverage

The **eclipse-ant** Java project provides moderated duplication in both coverage and number of clones. At the same time, the coverage did not jump so high when switching between EM and TM (38 compared to 16). This project behaves constant and moderate in all the cases. The coverage is very low (12 % maximum), which confirms the quality of such a mature, open-source community project. While the number of duplication chains gets approximately double when switching from **EM** to **LM** or **TM**, the coverage follows the same pattern. This could be a proof of good design, naming conventions (similar results from both approximate matching strategies), and furthermore, a balanced distribution among the types of clones.

The **weltab** C project has the highest overall coverage and very high values for every matching strategy. Moreover, looking at the slight difference between the exact matching and the approximate matching strategies, we can say that there is a very small percentage of clones with renamed variables. Although the size of *weltab* is almost insignificant compared to the sizes of the other "competitors", it has the highest coverage, which is an obvious cry for refactoring (over 75% coverage means more than 3/4 of the lines of code have at least one replica in the project).

The **cook** C project, is the biggest project among all. While providing the biggest number of clones, it is not by far the worse project, due to the good coverage results. It could be a second best after **eclipse-ant**. Moreover, many of its duplication (more than half) are made of generated code, which is not subject to refactoring, thus is less relevant. Nevertheless, the case provides a

strange result: while in terms of number of clones, LM is higher than TM, in terms of coverage TM is higher than LM. The higher coverage of the TM could be the correct result, while the higher number of clones with LM could have the following explanation: some of the similar pairs could have been missed by LM (see the example with he variables i and *aLargerVariableReplacingI*) and this would lead to lower coverage on the one hand and a higher number of smaller clones for LM on the other hand. The TM would have higher coverage due to the similarities missed by LM and would have larger but fewer clones (just consider one clone for TM and two half-size clones for LM).

4.2.3 Experimental Conclusions

It is time to draw conclusions on the practical applicability of the presented approach:

- The exact matching is clearly superior in terms of performance to any of the approximate matching strategies that we studied. This is a natural consequence of the complex dynamic algorithms that compute the Levenshtein distance.
- Quite as we expected, the approximate matching strategy at token-level is a few times faster than the classical (character-level) approach. This can be explained through the fact that the number of tokens per LOC is a few time smaller than the number of characters per LOC (since a token can be made of a number of characters).
- Also, the relevance of the results is better for the tokenized approach, since the renamings are applied to lexical elements (tokens).

Chapter 5

State of the Art

The problem of detecting clones in systems is an established software engineering problem known to occur in many contexts, including during pattern detection, software refactoring and perfective maintenance, system quality evaluation, and class library reengineering.

Software clones have been a focus of research for at least a decade, and dozens of papers on the topic have appeared. Current levels of interest in the topic appear heightened: concerning both the phenomenon of software clones (how, when, and why they occur, etc.), and the construction of clone detection tools.

Clones have been considered potential problems for maintenance [JO93]. Many automated and semi-automated techniques for detecting clones have been proposed over the years ([Bak92], [MLM96], [BYM+98]). Similar sorts of problems and techniques occur also in other contexts such as memory compaction, efficient delta-based storage, and plagiarism and copyright infringement detection (e.g., [Gri81]).

5.1 Early Concerns

Ralph Johnson has taken a parse-tree based approach [Joh91] to finding replicated code, but at that time the exhaustive search used on parse trees to identify identical subtrees or subtrees related by change of parameter was found to be unsuccessful because of time and space usage. Kenneth W. Church and Jonathan I. Helfman published a paper [CH93] that presented a tool called Dotplot, which they described as *a program for exploring self-similarity in millions of lines of text and code*. Programs aimed at detecting student plagiarism have typically used statistical comparisons of style characteristics such as the use of operators, use of special symbols, frequency of occurrences of references to variables, or the order in which procedures are referenced [Jan88]. Brenda S. Baker, in her

'92 paper [Bak92] presented Dup, *A Program for Identifying Duplicated Code*.

5.2 Actual Concerns

Over the last years, researchers from all over the world spent increasing effort in the field of software clone detection.

5.2.1 International Workshops

In October 2002 the First International Workshop on Detection of Software Clones took place in Montreal, Canada. It was held in conjunction with ICSM'2002 and the Workshop on Source Code Analysis and Manipulation (SCAM'2002). The workshop's concern was mainly to present the results of a tool comparison experiment, lead between January and April 2002. There were 4 tools which were presented in the published papers on this workshop ([BYM+98], [KKI02], [Kri01] and [DRD99]) that were compared and the whole experiments, along with the results was presented in another work [Bel02].

The 2nd International Workshop On Detection Of Software Clones (IWDSC'2003) was held in conjunction with WCRE'2003 in Victoria, British Columbia, Canada, in November 2003. The aim of this half-day workshop was to bring together researchers within the field of clone detection to critically assess the current state of research, and to establish new directions and partnerships for research. Various techniques have been proposed for automatically and semi-automatically detecting clones and refactoring them. Among the main concerns of this workshop:

- a study concerning why, how and when clones occur in industrial software systems [KG03]
- a research effort to define new similarity measures and new approaches aiming at reducing the computational cost [EM03]
- a study [NS03] of the effect of XP (eXtreme Programming)on the use of code duplication
- discussion about software clones in the Web sites domain, with emphasis on the issued of generated code [KHAM03]
- inspired by the information retrieval (IR) domain [WL03] proposed techniques for evaluating the clone detectors based on simple performance measures borrowed from information retrieval

5.3 Fierce Competition on Clone Detection

Baxter *et. al* [BYM+98] presented in the paper simple and practical methods for detecting exact and near miss clones over arbitrary program fragments in program source code by using abstract syntax trees. He compares subtrees searching for exact matches or similarity (near-exact). This approach is more precise than the one based on comparing strings of characters, but on the other hand is more language dependent (needs a parser for every programming language) and is harder to scale up, because of the memory needed to store the abstract syntax trees. It should also work slower because the processing time needed to build those trees. By using standard parsing technology, their tool (called CloneDR) detects clones in arbitrary language constructs, and computes macros that allow removal of the clones without affecting the operation of the program.

Kamiya *et. al* presented a *Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code* called CCFinder [KKI02]. The paper proposes a new clone detection technique, which consists of transformation of input source text and token-by-token comparison. Their process of clone detection is done in 4 steps: a lexical analysis on the code (where whitespace is eliminated), transformation of the resulting token sequence, match detection and formatting the resulting clones (mapping the token positions into places in the correspondent files). Kamiya's paper also describes some metrics for evaluating clone pairs and clone classes.

Krinke's approach [Kri01] is based on fine-grained program dependence graphs (PDGs) which represent the structure of a program and the data flow within it. In these graphs, his tool called Duplix tries to identify similar subgraph structures which are stemming from duplicated code. Therefore he considers not only the syntactic structure of programs but also the data flow within (as an abstraction of the semantics). As a result, there is no tradeoff between precision and recall.

The result of Rieger and Ducasse's approach in [DRD99] is a visual tool called Duploc which is written in Smalltalk and developed in the Software Composition Group at the University of Bern. Duploc is a lightweight, visual tool that can generate a scatter-plot out of a set of files, and every mark on the scatter-plot is a match between 2 lines of code. Clicking on a mark will open a view of the 2 files involved with the implicated line of code highlighted. In their paper, Rieger and Ducasse present some patterns (*dot configurations*) that often come up, like exact copy, modified, delete or insert, which are the inspiration behind the types of duplication chain identified in [DRD99]. Unfortunately, Duploc has some problems when dealing with bigger projects. Another fact is that Duploc has to be used by a trained person; identifying some of the duplications is not very easy and is directly influenced by some approach-related factors like: the current zoom level, the currently displayed area of the scatter-plot.

5.4 Gapped Clones

An interesting approach is [UKKI02], based on the CCFinder [KKI02], discusses the detection of so called "gapped clones", a concept very close to the one of duplication chains. However, this approach is located at a finer granularity (token level), not at the line of code level. Moreover, they do not take into account the case of combined gaps (*i.e.*, gaps made of both inserted or deleted and modified tokens). The paper proposes a method to visualize the gapped clone just as if they were actually detected, based on the detection results of conventional code clones. While this approach only shows all the candidates of gapped clones based on the output of CCFinder, our approach literally detects them, being built around the duplication chain concept.

5.5 Levenshtein Distance Used in Clone Detection

The authors of [GADLF02] have proposed an interesting approach, involving the use of the Levenshtein distance for clone detection. The paper's main concern is around the detection of duplicated web pages, which they consider to be pages with the same structure but with different data (information to be displayed). They map the HTML tags to an alphabet of symbols and then compare the resulting strings by applying the Levenshtein distance. Comparing the distance with a threshold, they are able to tell if the pages are structurally identical, similar or largely different. Such a Levenshtein distance based approach, while using the same distance is not scalable to industrial size projects for at least two reasons. In order to map every token type of a software system to an alphabet of symbols, the tool would get aid from at least a lexical analyzer, which would result in the loss of language independence. Moreover, there could be scalability problems, since the modern programming languages offer a large number of lexical constructs.

5.6 Clones With Variables Renaming

Baker was interested in detecting "sections of code that are the same except for a systematic change of parameters such as identifiers and constants" in [Bak92]. Baxter introduces the concept of "near-miss clones" in [BYM+98] in its abstract syntax tree based clone detection approach. The idea behind this is to compute similarity based on identical node and different nodes in the ASTs. The authors of [KKI02] also address the detection of clones with renamed variables, by substituting the variables and constants names with generic names in the preprocessing phase. While this approach detects type 2 [Bel02] clones, it also introduces false positives (if one of the code fragments operates with the same constant and the other code fragment uses different constants every time, the generic name substitution would hide this information).

Chapter 6

Conclusion

The combination of flexibility at both levels of granularity, one by means of the duplication chain concept at the level of lines of code sequences and the other through matching techniques based on similarity, is a powerful mechanism. None of the two concepts, taken separately could address both problems that these two together do:

- detection of clones with renamed variables (type 2 clones)
- detection of clones with other types of modifications (type 3 clones)

In terms of scalability, none of the similarity based matchings is appropriate for exhaustive analysis of industrial-size systems. However, the token-level similarity approach, due to both the increased relevancy of its results and the better time performances, can be used to detect clones with renamed variables (candidates for refactoring), once the problematic areas of the system have been identified with the exact matching strategy.

6.1 Evaluation of Contribution

The similarity based on the Levenshtein distance, applied at the token-level is a novel language-independent approach to detect clones with renamed variables. Moreover, we provided strong tool support for the approximate matching strategies and we also assessed the applicability, scalability and time performance of the techniques presented in this thesis, based on concrete case-studies.

6.2 Pros and Cons

Compared to the approach based on exact matching of lines of code, the token-level Levenshtein distance based proposed in this thesis provides both advantages and disadvantages. Both of these approaches find a place, however in different situation, depending on the purpose of the analysis and the size of the analyzed project.

6.2.1 Pros

- It is able to detect clones with renamed variables, which are an important but less obvious part of the clones. Such clones are missed by clone detectors which operate with line of code granularity
- Increased flexibility, with multiple combinations that can result by combining the comparison of lexical elements sequences with the concept of duplication chain
- High scalability
- Language independency (no need for lexical or syntactical analyzers)

6.2.2 Cons

- The weaker time performances are caused by the complex dynamic algorithm used to compute the Levenshtein distance (especially the non adapted version of the similarity based on the Levenshtein distance, which performs comparisons on character level)
- In theory, there is a possibility to obtain false positives, for the similarity between token sequences is computed based on no lexical or semantical information (the detector cannot determine whether a precise token is really a variable or a constant). However, the case studies have proved that in practice the probability to obtain false clones is extremely low, if a proper similarity threshold is chosen.

6.3 Future Work

We are interested in studying, with aid of the supporting tool, possible correlations between types of clones (and clones signatures) and general solutions to eliminate them, which leads us towards a more active involvement in code duplication refactoring.

Bibliography

[Ale79] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, New York, 1979.

[Bak92] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[Bak93] Brenda S. Baker. A Theory of Parameterized Pattern Matching: Algorithms and Applications (Extended Abstract). In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 71–80, May 1993.

[Bel02] Stefan Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis, Universität Stuttgart, September 2002.

[BYM+98] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings ICSM 1998*, 1998.

[CH93] Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines for text and code. *J. Computational and Graphical Statistics*, 2(2):153–174, June 1993.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM '99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, September 1999.

[EM03] Massimiliano Di Penta Ettore Merlo, Giuliano Antoniol. Complexity and feasibility issues in object oriented clone detection. 2003.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[GADLF02] Massimiliano Di Penta Giuseppe Antonio Di Lucca and Anna Rita Fasolino. An approach to identify duplicated web pages. In *Proc. of the 26 th Annual Computer Software and Application Conference (COMPSAC) 2002*, pages 481–486, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[Gri81] Sam Grier. A Tool that Detects Plagiarism in PASCAL Programs. *SIGSCE Bulletin*, 13(1), 1981.

[Jan88] Hugo T. Jankowitz. Detecting Plagiarism in Student PASCAL Programs. *Computer Journal*, 1(31):1–8, 1988.

[JO93] Ralph E. Johnson and William F. Opdyke. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742 of *Lecture Notes in Computer Science*, pages 264–278. Springer-Verlag, November 1993.

[Joh91] Ralph Johnson. Personal communication. 1991.

[KG03] Cory Kapser and Michael W. Godfrey. Toward a taxonomy of clones in source code: A case study. In *Proceedings of the First International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*. IEEE, September 2003.

[KHAM03] Holger M. Kienle and Anke Weber Hausi A. Müller. In the web of generated "clones". 2003.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eigth Working Conference on Reverse Engineering (WCRE'01)*, pages 301–309. IEEE Computer Society, October 2001.

[Lev66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics - Doklady*, 10(8):707–710, February 1966.

[MLM96] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Automatic detection of function clones in a software system using metrics. In *Proceedings of ICSM (International Conference on Software Maintenance)*, 1996.

[NS03] Eric Nickell and Ian Smith. Extreme programming and software clones. 2003.

[UKKI02] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *Proceedings Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*, pages 327–336, Gold Coast, Australia, December 2002. IEEE.

[Wet04] Richard Wettel. Automated detection of code duplication clusters. Diploma thesis, Department of Computer Science, "Politehnica" University of Timişoara, June 2004.

[WL03] Andrew Walenstein and Arun Lakhotia. Clone detector evaluation can be improved: Ideas from information retrieval. 2003.