# Automated Detection of Code Duplication Clusters

by
Richard Wettel

Diploma Thesis

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

Timişoara,
June 2004

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**

## Acknowledgments

I am very proud to have Dr. Radu Marinescu as my advisor. I am profoundly indebted to him for his trust in me and for seeing my skills even when I was sceptical about it. Due to his permanent encouragement I was able to understand that research is not something unreachable and too much above us, but rather something that can be done with hard work and enthusiasm. Being around him is a permanent source of inspiration to me.

The experiences I've been through in the LOOSE Research Group made me feel fortunate. The great ideas that Dani Raţiu had, expressed in his rather reserved way due to modesty, helped me many times over this project (by the way, pair programming is really fun), as well as Pepi's permanent availability to discuss algorithms (and to share the scalability progress!). I learned a lot with you guys. I would also like to thank Cristina and everyone else who tested the tool, starting with its early versions. Without the feedback I got from her, DuDe would have never become that reliable.

And to my lovely wife, Simy, for always being there for me.

Timişoara, Richard Wettel
june 13, 2004

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

### 1.1.1 Context

*All systems change during their life-cycles. This must be borne in mind when developing systems expected to last longer than the first version.*

**I. Jacobson [JCJO92]**

All software systems are subject to continuous evolution and maintenance activities in order to eliminate defects and extend their functionalities.

We need to deal with code duplication in order to prevent some problems that will appear when trying to adapt to the changes that are imminent in a real software system (one that stands its first release).

### 1.1.2 Code duplication. Making of...

**Definition 1.1 (Code clone)** *A code clone is a code portion in source files that is identical or similar to another [KKI02].*

Code duplications (or code clones) appear for a variety of reasons:

- Code reutilization by copying existing solution
- Failure to identify or use abstract data types
- Performance enhancement
- Accidents

Code reutilization misunderstood is when developers systematically copy previously existing code which solved a problem similar to the one they are currently trying to solve. Programmers intent on implementing new functionality, find some working code that performs a computation nearly identical to the one desired, copy it entirely and then modify in place. In large systems, this method may even become a standard way to produce variant modules. When building device drivers for operating systems, much of the code stays the same, and only the part of the driver dealing with the device hardware needs to change. In such a case, it is often for a device driver author to copy a whole existing, trusted driver and just modify it where needed. While this is actually good reuse practice, it complicates the maintenance issue of removing a bug found in the "trusted" driver by replicating its code (and reusing its bugs) over many new drivers.

Some duplication owe their existence to justifiable performance reasons. Systems with tight time constraints (like real-time systems) are often hand-optimized by replicating frequent computations. An extra function call may mean extra time that the system does not not afford to waste.

Duplicating code proves easy and cheap during the software development phase, but it makes software maintenance much harder. Typically, the amount of duplicated code in a software system stands between 5% and 10% of the code, but it can reach up to 50% [KG03b].

In conclusion, copying code can be done out of the urge to finish the problem under time pressure and with the tentation to use an already implemented solution in a impropriate way (and not by using the mechanisms provided by the modern object-oriented programming languages).

### 1.1.3 Why is code duplication a bad practice ?

Associated Problems

- Errors can be difficult to fix (they are not located in only one place)
- Errors and bugs may be copied as well
- Change in requirements may be difficult to implement
- Code size unnecessarily increased.

So, detecting this symptom may prove to be a step towards finding some design problems. Duplication is a clear sign of bad information and complexity management. Fowler considers it as the most striking of the *bad smells in code* [FBB+99].

Copying and pasting code becomes a problem when it comes to changing the code later. Every software system that is supposed to be kept alive after its first release will inevitably have to face changes in the requirements.Redundant code obstructs the system understanding during the maintenance activities, because of the amount of extra code to maintain and when one logical source of change affects many replicated code fragments scattered throughout a program, to adapt to a change, a programmer must find and update all the instances of the duplicated fragment. To do this, he needs to know whether some code fragment is duplicated and the exact location of its clones. Still, duplicating the code is usually done under time pressure, and the probability of having the duplications documented is quite low.

Continuity, one of Meyer's criteria for evaluating modularity [Mey88], is also affected because small changes to the specification would affect a larger number of components if the code was duplicated.

*The act of copying indicates the programmer's intent to reuse the implementation of some abstraction. The act of pasting is breaking the software engineering principle of encapsulation.* [BYM+98a].

One of the *signs of rotting designs* is immobility [Mar02b]. A system affected by it has tangled code, that is practically impossible to reuse, where lots of semantic duplication occur.

The essence to this section is that code duplication is bad, when it comes to maintenance. The dynamics of the software requirements leaves no doubt that either the code will change to adapt to the new specifications or will die. Because of errors that will have to be removed in every duplicated sequence of code, or modifications to a duplicated code will make maintenance more expensive than ir already is. We have ro address the code duplication issue.

### 1.1.4 Solutions

Poorly designed code does mainly the same thing in several places. An important aspect of improving design is to eliminate duplicate code. By eliminating the duplicates, you ensure that the code says everything once and only once, which is the essence of good design [FBB+99]. Achieving this is usually done by refactoring the duplicated code. In order to do that, one must first find the duplications.

M. Rieger and S. Ducasse ([RD98]) mentioned some of the reengineering goals related to the identification and removal of duplicated code:

- Identifying duplicated code in large scale system (100.000 lines) to huge system.

- Improving maintenance. Detection helps the maintainer of a system to make sure that some code fragment, where an error has been fixed, is not copied a number of times with the error still in it or is fixed differently at each location by maintainers who have no knowledge of each other's activities.
- Reducing maintenance cost. By detecting clones of a piece of code to be maintained and merging the code into one instance, the multiplied effort otherwise necessary to maintain all the clone instances is removed.
- Improving the code readability. By identifying duplicated code and refactoring it, the size of code is reduced. The level of abstraction is elevated when similar code pieces are refactored in a new method. In one of the FAMOOS [BBC+99] case studies, there was a method of 6000 lines of C++ code, which is a nightmare in complexity by any standards.
- Improving compilation time. The less lines of code you have, the faster your system is compiled.
- Reducing the footprint of the application. The less lines of code you have, the smaller the executable of your application gets.

Poorly designed code usually takes more code to do the same things, often because the code quite literally does the same thing in several places. Thus an important aspect of improving design is to eliminate duplicate code. By eliminating the duplicates, you ensure that the code says everything once and only once, which is the essence of good design. But before all these happen, we need to locate the code duplication.

### 1.1.5 Desperate need for dedicated software tools?

Spotting the duplicated code can be sometimes obvious, but most of the time is more subtle or can be easily missed, especially with large software systems (like legacy systems, whose maintenance is not an easy task), which often contain large amounts of duplicated code. Code analysis can be a time consuming activity, therefor tools able to improve the speed and effectiveness of this process are desired. Detecting the clones helps the maintenance activity by pointing suspect zones that maybe should be refactored, reducing by that the costs of maintenance.

This is impossible to be done by hand when dealing with industrial-size projects. The fact that all we start with is a set of source files and no *a priori* information about possible locations of code duplicates, suggests that exhaustive searching has to be performed, which desperately calls for powerful tools support.

## 1.2 Contribution

The idea of implementing such a tool was born inside LOOSE Research Group in Timişoara, a small group of enthusiasts concerned with quality analysis of object-oriented software and reverse engineering. The benefits of using such a tool were already known, from our experience of working with a similar tool called Duploc [RD98] in analyzing systems from the industry. Duploc is a Smalltalk program made in the Software Composition Group (SCG) from the University of Bern, a group with whom we have a very fruitful collaboration since several years.

We needed a similar tool, more adapted to our needs, and capable of integrating with other existent tools developed in our group. Some good ideas to improve the tool have been distilled out of those experiences.

So, we wanted a configurable tool capable of detecting code duplication, that could be later integrated in the reengineering platform developed by LRG (Insider). The tool should be accurate, scalable to industrial-size projects, and should stay language independent. With these in mind, a new tool was born and the name was DuDe (Duplication Detector). The program, written in Java has been developed in order to help during the code analysis activities.

The tool can be used either as a stand-alone program or as part of a reengineering platform developed in LRG called Insider. The duplication data provided by DuDe are used by Insider in its own static analysis process, which consists - among others - of metrics computation for quality measurement of software design. The duplication data can also be used to define new *detection strategies* [Mar02a] based on these duplication-related metrics.

The tool uses textual comparison to detect **chains** of duplicated code and it can be parameterized, offering extra flexibility to the searching process. As 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.

Often, copying a fragment of code is accompanied by small modifications to that code, aimed to adapt it to the current problem (by modifying, inserting or deleting lines of code), which makes these duplications harder to find by simple textual comparison. In spite of that, DuDe has been conceived to be able to find duplication chains that include these modifications. The tool's detection engine is parameterizable and it can also cover other fine changes to the code (renaming of variables, condensing of code into a single line, changing indentation, comments).

The searching parameters help the user to filter the results: minimum length (in LOC) of the duplication chains, whether to analyze comments or not (very useful to track a copy & paste activity made under pressure, where some of the code was adapted, but not the comments), maximum line bias (in LOC) which limits the number of lines of modified, inserted or deleted code within a duplication chain and minimum length of the exact duplicated portions of code (in LOC).

Thus Dude can find duplication chains that can be composed of a number of exact duplicated sequences (that I will further refer as *exact chunks*), separated by fragments of modified, deleted or inserted lines of code.

Duplications are full objects that have references to the entities they belong to (files or method bodies), the line indexes of the beginning and of the end of the duplication chain, length, type (exact, modified, insert, delete or composed) and *signature*, a feature that embeds the precise form of the chain, conceived to retain the pattern that a visual tool would offer by displaying the scatter-plot.

The tool presented in this work is appropriate for analyzing software systems written in any language (because the matching is done by textual comparison) and is scalable (it has been successfully applied on software systems from 100 KB to 30 MB).

## 1.3 Outline

Chapter 2 describes today's software engineering fundament. We will make a short review on the mechanisms offered by modern object-oriented programming languages, of the principles that help when looking for a good software design. There is a section dedicated to design patterns, some refactorings to use when eliminating the bad smells of code related mainly to code duplication. And there is a short section on the practices that make extreme programming so interesting.

Chapter 3 discusses the state of the art in the field of code duplication detection, related to object-oriented design. There has been some 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 prepared a list with the most desirable features that a clone detector should have.

After setting up the environment, the problem and other authors' work, chapter 4 presents this work's approach on detecting the duplicated code. It cuts through directly to the principles on which algorithm for clone detecting relies on. Then, a short description of the system architecture is presented, along with UML class diagrams, for a deeper understanding of the approach. Chapter 4 ends with a list of specific terms used when describing the algorithm.

Chapter 5 does an evaluation of the work, along with reports on the results of applying the clone detector to a suite of 8 running software systems. It starts with a brief presentation of the tool's features (graphical user interface, configuring the detection process through parameters). Then we get a glimpse of the experience of integrating DuDe in an integrated analysis platform (Insider) and the way they interact. Next, DuDe is subject of an experiment that should prove its industrial strength (8 C/C++ and Java projects, the same that were used when testing the tools on the First International Workshop on Detection of Software Clones). Then, I tried to look deeper into the results reported by the tool and to conclude some more. The case study is java swing, with an interesting result. The last section of the chapter is confronting the list from chapter 3.

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

# Chapter 2

# Fundaments

This chapter's goal is to introduce some of the idioms and principles that guide nowadays the object oriented design and programming.

## 2.1 Object-Oriented Programming

Object-oriented methods provide a set of techniques for analyzing, decomposing, and modularizing software system architectures. The object-oriented programming is concerned with implementation issues and is highly dependent on the object-oriented programming languages.

The main mechanisms provided by the modern object-oriented programming languages are:

1. abstract data types (classes)
2. encapsulation
3. inheritance
4. polymorphism

**Encapsulation** is basically described as hiding data. Objects generally do not expose their internal data members to the outside world (that is, their visibility is protected or private). But encapsulation refers to more than hiding data.

The advantage of using encapsulation is that more we make our objects responsible for their own behaviors, the less the controlling programs have to be responsible for. Encapsulation makes changes to an objects internal behavior transparent to other objects. Encapsulation helps to prevent unwanted side effects. With encapsulation the data structure of a class is hidden behind an interface of operations.

**Inheritance** is another vital mechanism of object-oriented programming. Instead of defining every time new types from scratch, we can use types (classes) that already exist and specialize them. This is the support for the *is-a* relationship: having one class be a special kind of another class. The base class (called the superclass) can be extended by any number of new classes and this is how class hierarchies appear.

**Polymorphism** is the ability of related objects to implement methods that are specialized to their type. We are able to refer to different derivations of a class in the same way, but getting the behavior appropriate to the derived class being referred to. This way, it offers basis for flexible architectures and designs. The high-level logic is defined in terms of abstract interfaces (the "program to an interface not to an implementation" heuristic), and relies on the specific implementation provided by the subclasses. What we apparently refer are objects with one type of reference that is an abstract class type. However, what we are actually referring to are specific instances of classes derived from their abstract classes. The subclasses can be added without changing high-level logic. Objects of the subclasses can be dynamically interchanged without affecting their clients.

## 2.2 Object-Oriented Design

The object-oriented design is a method for decomposing software architectures, based on the objects every system or subsystem manipulates. It stays relatively independent of the programming language used.

### 2.2.1 OCP

One of the most important principle in software engineering is the **Open-Closed Principle** ([Mar02b] citing Bertrand Meyer). This principle states that *software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.* It says that you should design modules that never change. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

Modules that conform to the open-closed principle have two characteristics:

- They are **Open** For Extension. This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.

- They are **Closed** for Modification. The source code of such a module is inviolate. No one is allowed to make source code changes to it. It would seem that these two attributes are at odds with each other. The normal way to extend the behavior of a module is to make changes to that module. A module that cannot be changed is normally thought to have a fixed behavior. How can these two opposing attributes be resolved?

Abstraction is the key: the module has to depend on an abstract class, not on a concrete class (Fig. 2.1). In order to adapt to changes in the requirements, all we need to do is to create a new class that inherits from the base abstract class and to specialize the method affected by the changes.

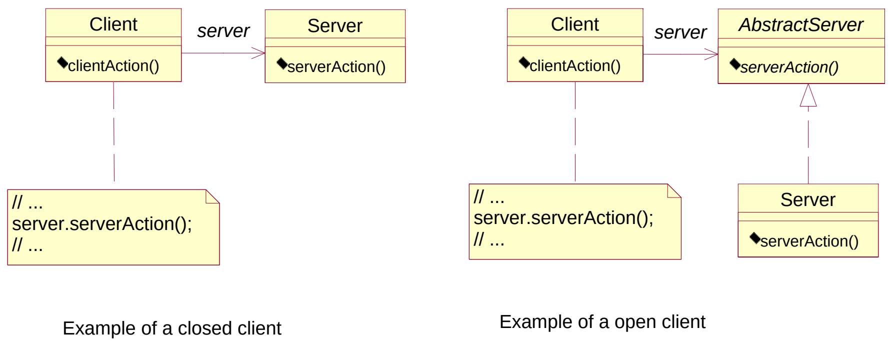


Figure 2.1: Open-closed principle

### 2.2.2 DIP

Martin's **Dependency Inversion Principle** provides the means to respect the Open-Closed Principle. This principle gives theoretical support to avoid bad design. Martin's symptoms of bad design are:

- *rigidity* which is described by the difficulty to change the software, because it would affect too many other parts of the system
- *fragility* is the property of a software system that would break in unexpected parts if one would make a change somewhere in the system.
- *immobility* is the impossibility to reuse system's code because of the strong coupling between the subsystems.

The principle states that *high level modules should not depend upon low level modules. both should depend upon abstractions.* or, in other words *abstractions should not depend upon details. details should depend upon abstractions.*

The traditional software development methods, such as structured programming tend to create software structures in which high level modules depend upon low level modules, and in which abstractions depend upon details.

### 2.2.3 LSP

The Liskov Substitution Principle states that *"Subclasses should be substitutable for their base classes*. This is the main idea behind the mechanism of polymorphism. This principle's concept is that a user of a base class should continue to function properly if a derivative of that base class is passed to it.

This may seem obvious, but there are subtleties that need to be considered. The canonical example is the Circle/Ellipse dilemma, depicted in fig. 2.2.

The Circle/Ellipse Dilemma. Most of us learn, in high school math, that

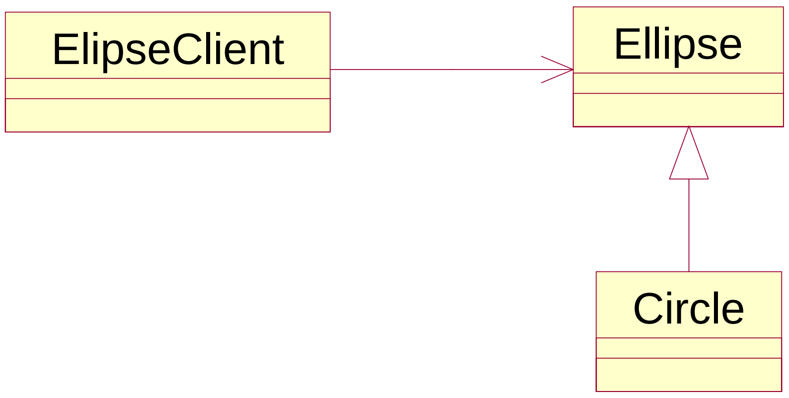


Figure 2.2: Circle/Ellipse dillema

a circle is just a degenerate form of an ellipse. All circles are ellipses with coincident foci. This *is-a* relationship tempts us to model circles and ellipses using inheritance. While this satisfies our conceptual model, there are certain difficulties. Ellipse modelled as a software entity does not correspond to that *is-a* relationship. While the Circle really only needs two data elements, a center point and a radius, the Ellipse needs are slightly different (two are the foci and the the length of the major axis).

Still, if we ignore the slight overhead in space, we can make Circle behave properly by overriding its *SetFoci* method to ensure that both foci are kept at the same value.

```
void SetFoci(Point a, Point b)
{
    itsFocusA = a;
    itsFocusB = a;
}
```

This way we will have an Ellipse entity that will change both its foci at one time. The problem that we have is when clients of the ellipse will work with an instance of a Circle. If, for example the client will change first the first focus and than the second he will expect (and possibly could make a unit test case) that the major axis corresponds to the value mathematically calculated. Instead, it will get another value, because he actually changed the radius of the circle two times, so the first value is gone. If we were to make the contract of Ellipse explicit, we would see a postcondition on the *SetFoci* that guaranteed that the input values got copied to the member variables, and that the major axis variable was left unchanged. Clearly *Circle* violates this guarantee because it ignores the second input variable of *SetFoci*.

**Design by Contract**. Restating the LSP, we can say that, in order to be substitutable, the contract of the base class must be honored by the derived class. Since *Circle* does not honor the implied contract of Ellipse, it is not substitutable and violates the LSP.

To state the contract of a method, we declare what must be true before the method is called. This is called the **precondition**. If the precondition fails, the results of the method are undefined, and the method ought not be called. We also declare what the method guarantees will be true once it has completed. This is called the *postcondition*. A method that fails its postcondition should not return.

Restating the LSP once again, this time in terms of the contracts, a derived class is substitutable for its base class if:

1. Its preconditions are no stronger than the base class method.

2. Its postconditions are no weaker than the base class method.

Or, in other words, *derived methods should expect no more and provide no less.*

As a conclusion, it proves that other than what we know from the world that we live in, when it comes to software, an *IS-A* relationship always refers to the **behavior** of the class.

### 2.2.4 Putting it all together

The relation between the 3 principles enounced by Robert Martin ([Mar02b] is: Open-Close states the goal, Dependency-Inversion provides the mechanism to accomplish that goal, while Liskov's Substitution Principle is the insurance for the mechanism.

## 2.3 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 2.1 (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 understanding 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.

### 2.3.1 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. 2.3. There may be many observers and the

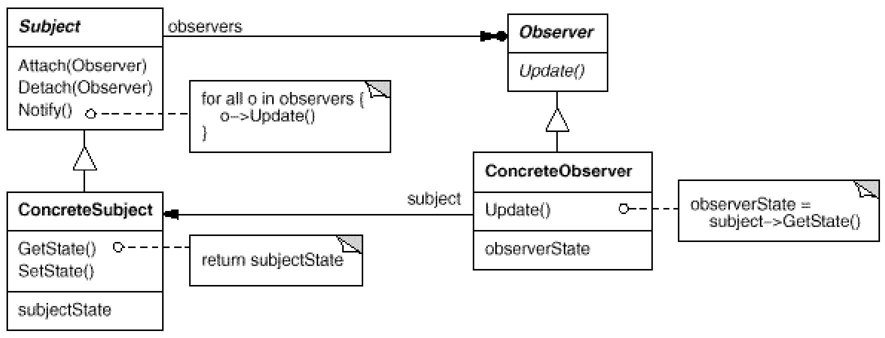


Figure 2.3: Observer

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. 2.4.

The consequences of applying the Observer pattern are:

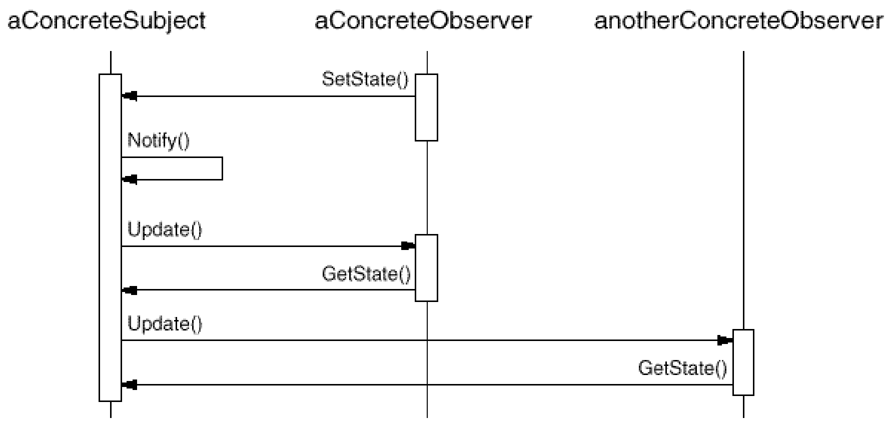


Figure 2.4: Sequence diagram for Observer

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.

### 2.3.2 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.2.5) 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. 2.6. After building 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 button 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 2.7 sequence diagram.
The problem here is that the object that ultimately provides the help isn't

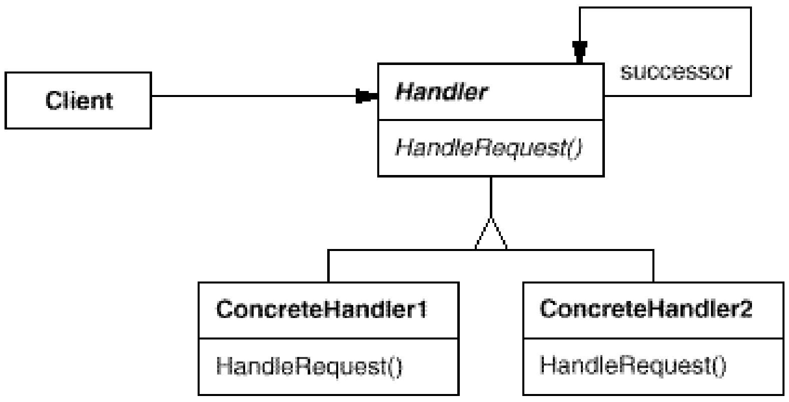


Figure 2.5: Class diagram for Chain Of Responsibility

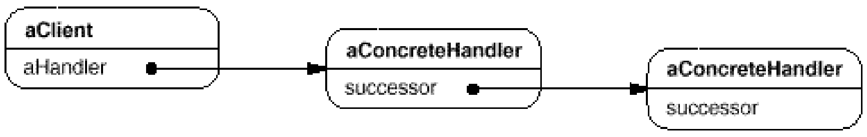


Figure 2.6: Building the chain

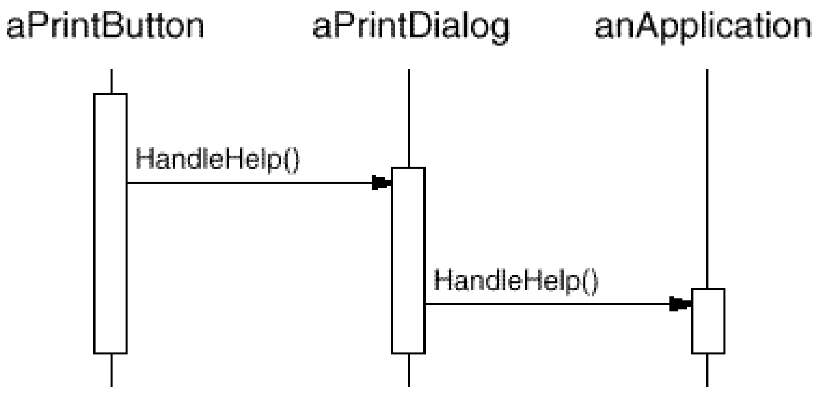


Figure 2.7: Interaction within the chain

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.

### 2.3.3 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. 2.8 class diagram.

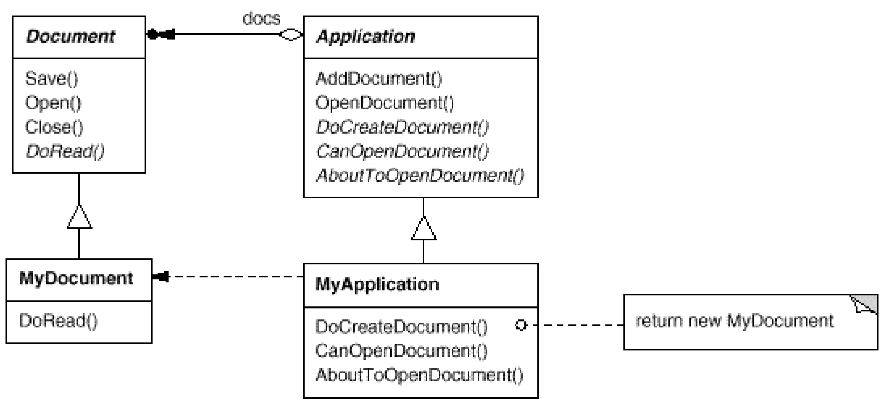


Figure 2.8: Example of applying TemplateMethod

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))
    {
        // 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.

### 2.3.4 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. 2.9).

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

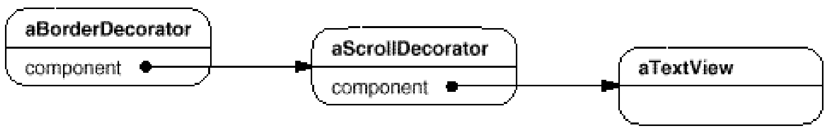


Figure 2.9: Example of composing

an abstract class for visual components that decorate other visual components (fig. 2.10.

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

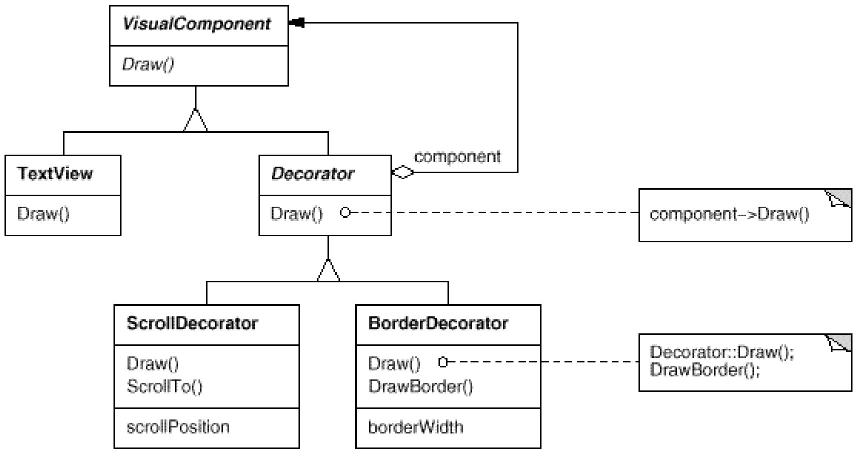


Figure 2.10: Example of applying Decorator

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. 2.11).

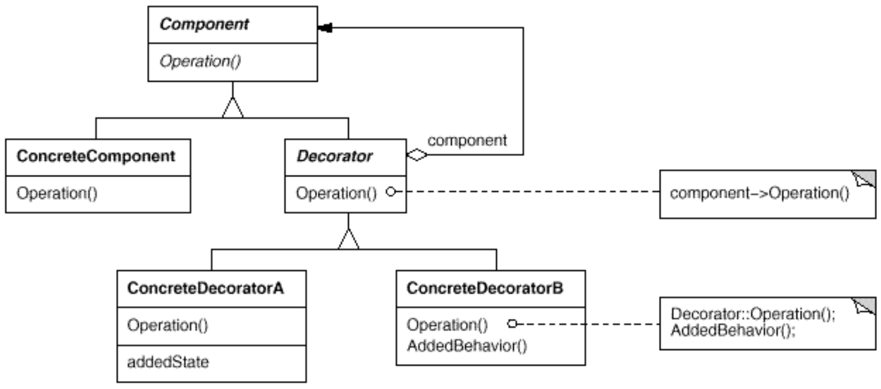


Figure 2.11: Structure of Decorator

## 2.4 eXtreme Programming

XP is a lightweight methodology for small to medium-sized teams developing software in the face of vague or rapidly changing requirements [Bec00]. XP takes commonsense principles and practices to extreme levels:

- *If code reviews are good, we'll review code all the time (pair programming).*
- *If testing is good, everybody will test all the time (unit testing), even the customers (functional testing).*
- *If design is good, we'll make it part of everybody's daily business (refactoring).*
- *If simplicity is good, we'll always leave the system with the simplest design that supports its current functionality (the simplest thing that could possibly work).*
- *If architecture is important, everybody will work defining and refining the architecture all the time (metaphor).*
- *If integration testing is important, then we'll integrate and test several times a day (continuous integration).*
- *If short iterations are good, we'll make the iterations really, really short-seconds and minutes and hours, not weeks and months and years (the Planning Game).*

The XP philosophy is best described by:

- early, concrete, and continuing feedback from
- short iterations, incremental planning approach
- reliance on automated tests written by programmers and customers to monitor the progress of development, to allow the system to evolve, and to catch defects early.
- the ability to flexibly schedule the implementation of functionality, responding to changing requirements

The XP practices include:

- small releases: first, the programmer will quickly write a simple program, then release new versions on a very short cycle
- testing: programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished
- refactoring: programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility

On the 2nd International Workshop On Detection Of Software Clones (IWDSC'2003), Eric Nickell and Ian Smith [NS03] describe the effect of XP on the use of code duplication (Chapter 3).

Beck presents a model of software development from the perspective of a system of control variables. In this model, there are four variables in software development: cost, time, quality and scope. The way the software development game is played in this model is that external forces (customers, managers) get to pick the values of any three of the variables. The development team gets to pick the resultant value of the fourth variable.

There is a strange relationship between internal and external quality. External quality is quality as measured by the customer. Internal quality is quality as measured by the programmers. Temporarily sacrificing internal quality to reduce time to market in hopes that external quality won't suffer too much is a tempting short-term play. And you can often get away with making a mess for a matter of weeks or months. Eventually, though, internal quality problems will catch up with you and make your software prohibitively expensive to maintain, or unable to reach a competitive level of external quality.

## 2.5 What is Refactoring?

**Definition 2.2 (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 2.3 (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**

.

### 2.5.1 Solutions based on refactorings

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.

### 2.5.2 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].

# Chapter 3

# 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 (c.f. Baker's 1992 paper [Bak92]), 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 (e.g., [Bak92], [MLM96], [BYM$^+$98b]). 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]).

## 3.1 Pioneers in clones field

Early interest in this field has been showed in the early '90s. 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.*

## 3.2 Actual concerns on software clones

The last years, researchers form all over the world spent increasing effort in the field of software clone (or duplicated code) detection.

### 3.2.1 First International Workshop

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+98a], [KKI02], [Kri01] and [DRD99]) that were compared and the whole experiments, along with the results was presented in another work [Bel02].

#### Evaluating the competition

Before presenting the results, we need to introduce a set of terms related to the clone detection evaluation process.

1. **Definition 3.1 (Reference)** *A* reference *is a clone in the given project, that has a known type.*

2. **Definition 3.2 (Candidate)** *A* candidate *is a clone reported by a detecting tool, to which the tool can assign a clone type.*

3. **Definition 3.3 (Recall)** *The ratio between the number of found real clones covered by the clone candidates found by a tool and the total number of clones in the system is called* **recall**. *It is a measure of how many of the existent clones can a tool detect.*

$$Recall(P,T) = \frac{References(P,T)}{References(P)}$$

4. **Definition 3.4 (Precision)** *The* **precision** *of a clone detecting tool is the ratio between the number of clones detected by the tool and the clone candidates. Precision is a measure of the number of clones that make*

*sense found by the tool.*

$$Precision(P,T) = \frac{References(P,T)}{Candidates(P,T)}$$

Baxter *et. al* [BYM+98a] 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. As they stated, *...program scale is a problem for any clone detection scheme. For our experiment, we were limited to 100,000 line chunks in 600Mb RAM because of artificially large memory-based data structures in our prototype DMS.* 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. The underlying concepts for their tool were:

- The tool should be industrial strength, and be applicable to a million-line size system within affordable computation time and memory usage.
- A clone detection system should have ability to select clones or to report only helpful information for user to examine clones, since large number of clones is expected to be found in large software systems. In other words, the code portions, such as short ones inside single lines and sequence of numbers for table initialization, may be clones, but they would not be useful for the users. A clone detection system that removes such clones with heuristic knowledge improves effectiveness of clone analysis process.
- Renaming variables or editing pasted code after copy-and-paste makes a slightly different pair of code portions. These code portions have to be effectively detected.
- The language dependent parts of the tool should be limited to a small size, and the tool has to be easily adaptable to many other languages.

The 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 this thesis. Even the concept of duplication chain is inspired by the scatter-plot visualizing. 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, the degree of attention or fatigue of the user (some of the duplications are easy to overlook).

### Experiment's Results

Bellon ran these tools on a set of software projects and analyzed the way the tools dealt with every project (he also analyzed Baker's DUP [Bak95] and Merlo's CLAN [LPM$^+$97]). The results were discussed in his paper [Bel02].

He divided the duplications into 3 categories:

- Exact copy
- Copy affected by renaming (of variables, methods)
- Modified copy (more than renaming, inserted code, deleted code)

The case studies subjects were 8 projects, 4 written in Java and 4 in C and C++, with sizes ranging from 11 SLOC (number of lines of code ,except null or comment lines) to 350 SLOC.

Some of the tools were configurable: number of processors (Baxter), clone's length (Baker, Baxter, Kamyia, Krinke, Merlo), percentage of similarity (Baxter), number of variable parameters of a Type-2 clone (renamed elements), metrics used (Merlo), some C++ specific parameters (eliminating namespace information, template parameters, etc.).

Some of the tools encountered problems with part of the study cases. Krinke's Duplix could not analyze the *weltab* project, for it had more than one *main()* function and he also could not analyze the *postgresql*, and for the *cook* and *snns* projects, for some unknown reason the times for processing were unacceptable high. Rieger's Duploc had problems in analyzing the 2 big systems: *postgresql* and *j2sdk1.4.0-javax-swing*, because it consumed too much memory and/or time.

After analyzing the results, Bellon could not name a winner, he could just admit that every method has its advantages and disadvantages. He divided the tools in two categories: the ones with high *recall* but lower *precision* and the ones with a lower recall but with higher precision. Baxter and Merlo belong to the ones with higher precision while Baker, Kamiya and Rieger got more recall, with loss of precision. Krinke managed high recall only with clones of type-3.

Bakers's DUP, with its token-based approach found a lot of clone candidates, was a good choice to search for exact copies (not for parameterized ones). It also proved to be less precise in delimiting the clones (often the start and end of clones were inexactly reported).

Baxter's CloneDR reported fewer candidates, but the most were qualitative (high precision). His tool was better at finding exact copies than the parameterized ones (with renamed elements). Type-3 clones were rarely found. Some of the type-1 clones (exact) were wrongly reported as type-2.

Kamiya's CCFinder uses also a token-based approach with some other transformations. Similarly to CloneDr, it found many candidates with a big ratio of not-qualitative clones. Type-1 clones were better recognized than type-2. It managed though to detect some type-3 clones. But the main problem is that it cannot recognize the candidates types at all.

Krinke's Duplix is not appropriate for Java projects and cannot analyze big Systems. Its precision is low and the recall proved to be good only for type-3 clones. Krinke claims this would be the only type his tool could find.

Merlo's CLAN with the metrics-based approach showed some good precision, but with the cost of low recall. It proved to find all three types of clones. The power of the tool is that it can detect complete copied function and methods, so the delimitations of the candidates are, most of the time, precise.

Rieger chose for Duploc a line-based approach and pattern recognition. The tool seemed to have problems with big projects, which it could not analyze. Duploc found many candidates, but the percentage of real clones low enough (high recall and low precision). The tool does not divide the candidates into types.

As a conclusion of that experiment was stated that there is no evident winner and there is no optimal solution to the clones detecting issue. Depending on the goal of the analysis is one tool or the other the appropriate tool.

### 3.2.2 Second International Workshop

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.

The papers presented with this occasion have discussed various issues around the clone problem. Some authors tried to characterize why, how and when clones occur in industrial software systems [KG03a], for they considered that for this issue there are only a few empirical studies.

1. Merlo *et. al* emphasized on *the need of a substantial research effort to define new similarity measures and new approaches aiming at reducing the computational cost, while, at the same time, fully considering the substantial individuality of objects together with their properties* [EM03].

2. Eric Nickell and Ian Smith [NS03] tried to study the effect of XP (eXtreme Programming) on the use of code duplication. They ran a near-clone detector over software that has been developed while making heavy, light, or no use of extreme programming (XP) practices. The results of their study was that the XP projects produced significantly lower scores on cloning than either the non-XP or semi-XP projects, and the project with the lowest score was written by the team with the longest experience and highest commitment to XP. The semi-XP and non-XP projects produced a wide range of score. There is evidence that the XP projects gain some of their advantage in lower scores in part because XP developers tend to produce smaller methods. With these intermediate results in mind, they further analyzed the two projects that produced the lowest and highest scores.

   Nearly all (95%) of the detections for donquixote (the non-XP project with the highest rate of clones) were typical near-clones: someone has done cut-and-paste with minimal changes, for methods up to 150 lines, while ardor (the full-XP developed project with the lowest score) had 91% of its duplications in the test code. The conclusion was that as they expected, XP-developed software is more *cloneresistant* than software developed in a more traditional manner.

3. Kienle et. al moved the discussion of software clones in the Web sites domain, with emphasis on the issued of generated code. *In order to be of use to a software maintainer, the clones detected by a tool should provide meaningful information. However, generated code can diminish the usefulness of clone detection significantly because generated "clones" are reported that are of no interest to the maintainer* [KHAM03]. Clone detection techniques have been developed for procedural as well as object-oriented software. More recently, this work has been expanded to include Web sites. Web sites can contain clones in HTML, scripts such as JavaScript and Perl, XSLT, XML, DTD, XSchema, etc.

   Automatic detection of duplicated code can be a useful aid for software maintenance and reverse engineering. For example, the detection of clusters of similarly structured HTML pages is a good starting point to refactor a Web site to a design that separates content and navigational structure. However, to be effective, the detected clones have to be sufficiently precise and meaningful in the eyes of the person inspecting the proposed clones. In this context, a particular problem arises for software that is partly handwritten and partly generated automatically. Software systems that consist of generated components tend to generate highly regular-sometimes even identical-pieces of code.

4. Walenstein and Lakhotia proposed some ideas for information retrieval meant to improve the evaluation of clone detectors [WL03]. Current evaluation techniques based on simple performance measures borrowed from information retrieval (IR) research could be enriched with additional IR evaluation measures.

   By now, the evaluation of clone detectors has been made mainly in terms of *recall* (the ratio between the number of clones that a tool can find and the total number of clones in the project) and *precision* (the ratio between the number of real clones found by a tool and the total number of clones candidates it found). Perfect recall, in this context, means that every clone is found, while perfect precision means that there is no false clone reported. Effectively there is but a single query being evaluated, namely: "find all clones". Normally the result is in the form of a simple set of results. That is, the order of the clone candidates is considered unimportant. This basic evaluation template has been followed by several researchers, and precision and recall values have become the *de facto* basis for empirical evaluation of automated clone detecting systems. Although this has proven to be a useful tactic, it limits what can be said about the results and how they may be compared. The authors feel that it is worthwhile to expand on these evaluation techniques, and that exploring other IR-based measures techniques is a good first step. IR has developed an arsenal, and from examining them several useful directions for CD evaluation might be derived.

The authors propose some other measures like *fallout* (the fraction of false-positive results) and even composite measures for the cases where a tool has higher recall and lower precision than other tool. Then a composite measure like the ratio of precision to recall, or precision to fallout would probably make the difference.

Another way of performing the comparison is by so called *precision/ recall curves*, which can show how a detector responds to changes in its tunable parameters (only clone detectors that can be tuned up by means of searching parameters). For instance, how does a detector's precision differ from another detector's precision when the recall is set to 80%? 50%? 30%?

One vigorous area of research in IR has been the investigation of ranked query result sets. Anyone using WWW search tools knows about ranked results: the most interesting results are presented first. Many existing clone detecting tools are easily modified so that they generate ranked outputs instead of amorphous result sets: merely rank the clone candidates. The ranking generally needs to reflect the user's goals.

## 3.3 More tools

*DOTPLOT* [Hel95] is a tool for displaying large scatter-plots, used to compare source code, but also filenames and literary texts.

*DATRIX* [MLM96] a tool that finds similar functions by comparing vectors of source metrics.

*SimScan* is a tool dedicated to finding duplications in Java. It is based on ANTLR, a tool for building parsers. It can be parameterized by: volume, similarity, speed/quality and can save reports. It is also provided as a plugin for IDEA.

*SimianUI* is a plugin for Eclipse that finds only exact duplications. It is not very flexible. By double-clicking on a reported duplication, you can jump straight to the relevant source files, with the duplicated regions highlighted.

*Dupman* is a plugin for Eclipse. It is hard to configure, it offers few details about the results (a degree of similarity).

One can use Unix's *diff* program to show differences between two files, or each corresponding file in two directories. diff outputs differences between files line by line in any of several formats, selectable by command line options. It has several options on ignoring white spaces, some options dedicated to C language source code analysis, input file filtering.

## 3.4 Success of a clone detector

After analyzing most of the existent tool, we made a list of attributes that we would like to see with any duplicate detection tool.

1. **Scalability** is a very important factor in the industrial context. The tools should be able scale up to large (hundreds of K LOC) and even very large software systems. The assistance such a tool offers is essential for the engineer that analyzes legacy systems, because such an exhaustive search on a huge system is impossible to do *by hand*.

2. Another important issue is the **speed** or the **processor time** for analyzing software systems. Often a software consultant gets mostly a few days to pronounce itself on the quality of a given system. If the clone detection tools cannot provide results in an acceptable amount of time, it is useless.

3. The **memory issue** is not one that we could overlook: dealing with huge systems (up to millions of LOC) could cause some problems if the tools do not manage the memory in an efficient way.

4. In order to get a **language independent** clone detector, the parts of the tool that are language dependent should be localized and reduced to a minimum. This way, the tool can be applied to any programming language, so it remains open for future languages

5. A useful feature of the tool is to be able to find so-called **parameterized** clones (Type 2 [Bel02]), because there is a habit of copying & pasting code and then modifying it by only renaming some of the participating elements (variables, methods, etc.).

6. The ideal tool in terms of recall and precision is a tool that provides results with **high recall and high precision**. It seems that it is hard to obtain both of them at the same time. The ones that have a high recall obtain it with the price of lower precision and a higher precision belongs together with lower recall.

7. The tool should not suffer of the **splitted-duplicates** symptom (a simple modification in a duplicate caused a detection of two independent duplicates, one before the modifications and another after them).

8. The detector should be able to categorize the duplicates into **types**, in order to get a better comprehension of the system, of the programmers that work on it, and of their copying habits.

9. Some of the tools were not able to **correctly delimit the duplications**. This is an important issue when it comes to validate the clones, because the one that makes the validation wants to get directly to the affected code, without searching it (the tool already did that!).

# Chapter 4

# The Approach

## 4.1 Algorithm's Principles

The approach chosen for this tool is an **enhanced scatter-plot approach**. The scatter-plot approach is not new to software research and it is based mainly on:

1. bringing the code to a brute state (non-indented, comments off)
2. building a matrix that will store the results of matching between the lines of code
3. populating it, by marking every match
4. presenting it to the specialist, for further visual studying

The *enhancement* we propose is to try to unite copied sequence of code that are close enough to each other and merging them into a cluster of code duplication. The tool will report a list of clusters (chains).

When introducing code clones, programmers often change white spacing (blanks, tabs, newlines) and comments, which will disable recognition based purely on strings. In order to combat this problem, the presented tool transforms the code lines by "cleaning" the code.

Clone detection is a process in which the input is a list of source files (or method bodies) and the output is a list of duplication chains. The entire process of our string comparison clone detecting technique consists of the following steps:

1. Code cleaning

   After reading the source code lines, the first thing to do is bringing the code to a raw state, in order to avoid situations where identic lines which are differently indented or having comments added are not reported as duplicates. This preprocessing phase cleans the code, by:

- striping the comments (optionally, only Java and C,C++)
- removing any whitespaces (including nice indentation)
- removing noise (specified in a file), i.e. lines of code containing only a keyword (*else*) or some other syntactic element (an open or a closed brace), which can be considered as less relevant to the duplication issue. It would not make us very happy if the tool would report us a duplication length of 5 LOC, which would consist of a closed braces on every line.

2. Building the matrix

   The lines of code the cleaning this phase (further referred to as *relevant lines*) will be stored in the exact order they were read: all the relevant lines of the first file, followed by the ones of the second file, and so on. The next step consists of building a two-dimensional matrix NxN, where N is the total number of relevant lines in the system. An element of the matrix Element[i,j] will store the result of matching the relevant lines i and j. This way, every line will be compared with every other line in the system (exhaustive approach). Only the hits of the matching will be marked in the matrix (Fig. 4.1).

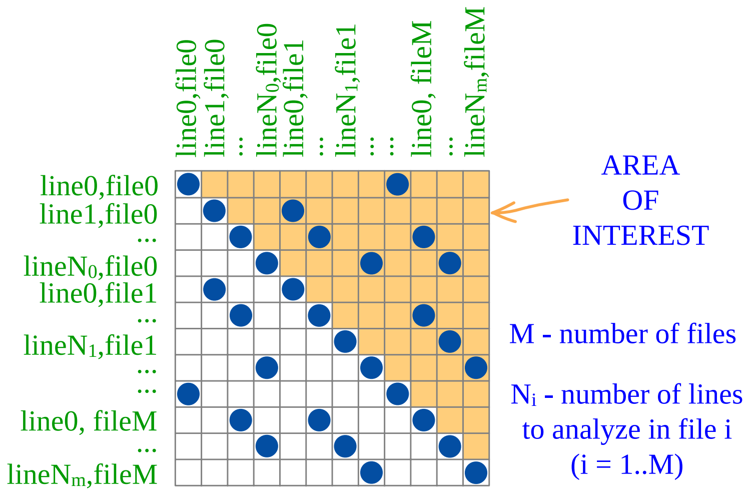


Figure 4.1: Building the matrix

The main diagonal is completely marked, each of its elements is actually the result of matching a line of code with itself. There is an obvious symmetry towards the main diagonal (the marks above it are the mirrored image of the marks below it).

**Definition 4.1 (Area of interest)** *Based on these reasons, I will define the zone above the main diagonal (excluding the diagonal itself) as* *the* **area of interest** *for the duplication issue, the rest of the matrix containing only redundant information.*

Due to the huge amount of memory needed to store the matrix when dealing with middle size to big software systems (a system with 370.000 lines of relevant code involves a matrix with 137 billions elements), an ingenious storing solution was needed. On one hand, it should offer good performance for searching and extracting operations and on the other hand this data structure should efficiently manage the memory in order to store the sparse matrix. Based on this reasoning, the 2-dimensional Array (Matrix) was replaced by a data structure with quick access to information and also very economical: a list (Java's ArrayList) which associates every line of code with a map (Java's HashMap) containing key - value pairs (key is the line index and value is the type of mark). The set of keys for such a HashMap puts together the indexes of all the lines that match the line associated to that specific HashMap.

Even with this new solution, by increasing the size of the analyzed code (5MB), the memory proved to be insufficient. Then again, a new solution had to optimize the performance and scalability. And because after extracting the duplication data (chains), the content of the matrix was no more useful, I divided the matrix into areas, each of these zones representing the intersection between the lines of code belonging to 2 files (Fig. 4.2).

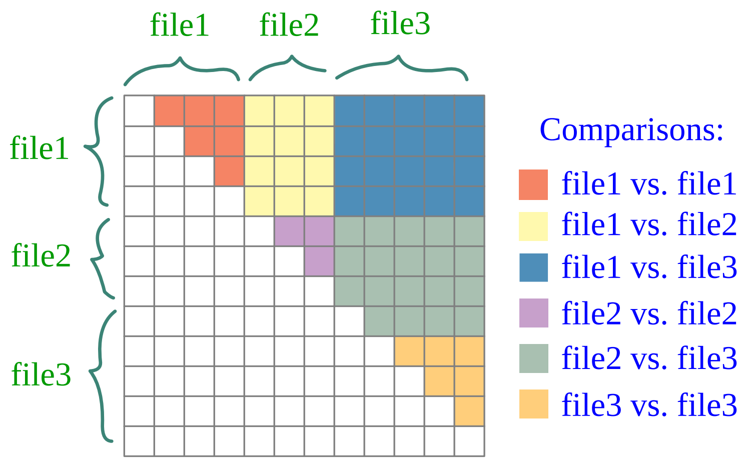


Figure 4.2: Area division in the matrix

**Definition 4.2** *A* **slice** *is the portion of the matrix containing the results* *of the comparison between the lines of code of two files: one spreading on horizontal direction and the other on vertical (the two files are not necessarily distinct files).*

This way, every area will be passed 2 times: one for marking the matches between each 2 lines of code and a second one for building the chains of duplicated code, based on the marks made in the first passage. Then, the memory occupied by that area is released and the algorithm continues with the next area in the matrix. By that, I am assured that the memory hosts at most one area of the matrix (the current area).

3. Tracking the duplication chains

For simplity reasons, the algorithm will be described in pseudo-code:

```
for (*every entity E1 (file) in the project)
    for (*every entity E2 that has not been fully processed)
        *set currentArea as the area corresponding to the
            intersection of the lines of code belonging to E1 and E2
        completeScatterPlot(currentArea);
        findDuplicationChains(currentArea);
        *add traced duplications to the list of clones;
        *free the memory occupied by currentArea;
    end
end
```

The procedure called *completeScatterPlot(area)* is best described by the next pseudo-code description of the algorithm:

```
for (*every line L in the area)
    for (*every column C in the area, starting with column L+1)
        if (code(L) = code(C))
            *mark element[L,C] with the value unused;
    end
end
```

At the beginning of the *completeScatterPlot* procedure, all the elements in the specified area are unmarked (null value). Then, every line of the matrix is compared to every column of the matrix, with respect to the area of interest (only cells above the main diagonal). Every cell found as a match between the lines of code that intersect in that specific point is marked with the value unused (false), as a sign that it has not been part of a duplication cell.

The procedure *findDuplicationChains(area)* can be described as follows:

```
for (*every line L in the area)
    for (*every element Elem[L,C] marked as unused)
        *create a list of coordinates;
        *add (L,C) coordinates to the list of coordinates;
        while *there is a next coordinate available
            if (*Elem[L+1,C+1] is marked)
                *add Elem[L+1,C+1] to the list of coordinates;
                *mark Elem[L+1,C+1] as used;
            end
            else
                if (*size of current exact chunk > minExactChunk)
                    *search for a marked element in Elem[L+1,C+1]'s
                        proximity, with respect to maxLineBias;
        end
        if (*the duplication chain that could be built based
            on the list of coordinates is longer than minLength)
                *add the clone to the duplications list
    end
end
```

Due to the structure of the *VirtualMatrix*, in this phase, the program has direct access to the marked cells (through an Iterator). This is an improvement by almost 150% of the program's speed performance, because there is no need to search every element on a line (we get a set of marked elements).

Once we found an unmarked cell, that is a potential start of a duplication chain, we start the tracing. The first place to look for a marked cell is the next cell on the diagonal direction (this searches for exact chunks). If there is a match, that cell is added to the chain and the algorithm continues the same way and then the element is marked as used. This way, we avoid detecting duplication chains that are overlapped or even contained ones. Else, it starts searching in the proximity of that cell (fig. 4.3), in order to find a continuation of the chain. The exact chunk is over, but there may be another exact chunk close enough (the distance limit between them is maxLineBias) to be linked together. Also the sizes of the exact chunks that compose the chain have to be at least minExactChunk.

The chain is considered closed when the line bias reaches the maxLineBias value without succeeding in extending the chain. The algorithm continues with the next unused mark on the current row of the matrix, until the end of the row. A concrete example of building a chain is shown in fig. 4.4.

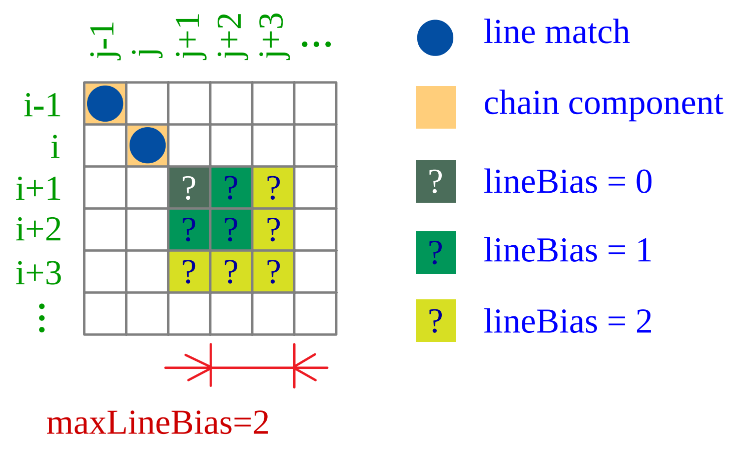


Figure 4.3: Duplication chain expansion

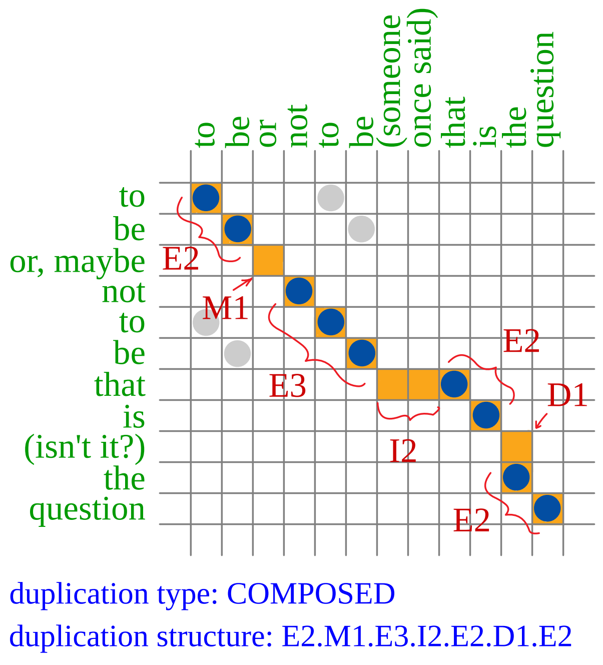


Figure 4.4: Duplication chain example

## 4.2 System architecture

The architecture of the clone detecting engine is described in the UML class diagram (fig. 4.5). In the middle of the diagram stands the *Processor* class,

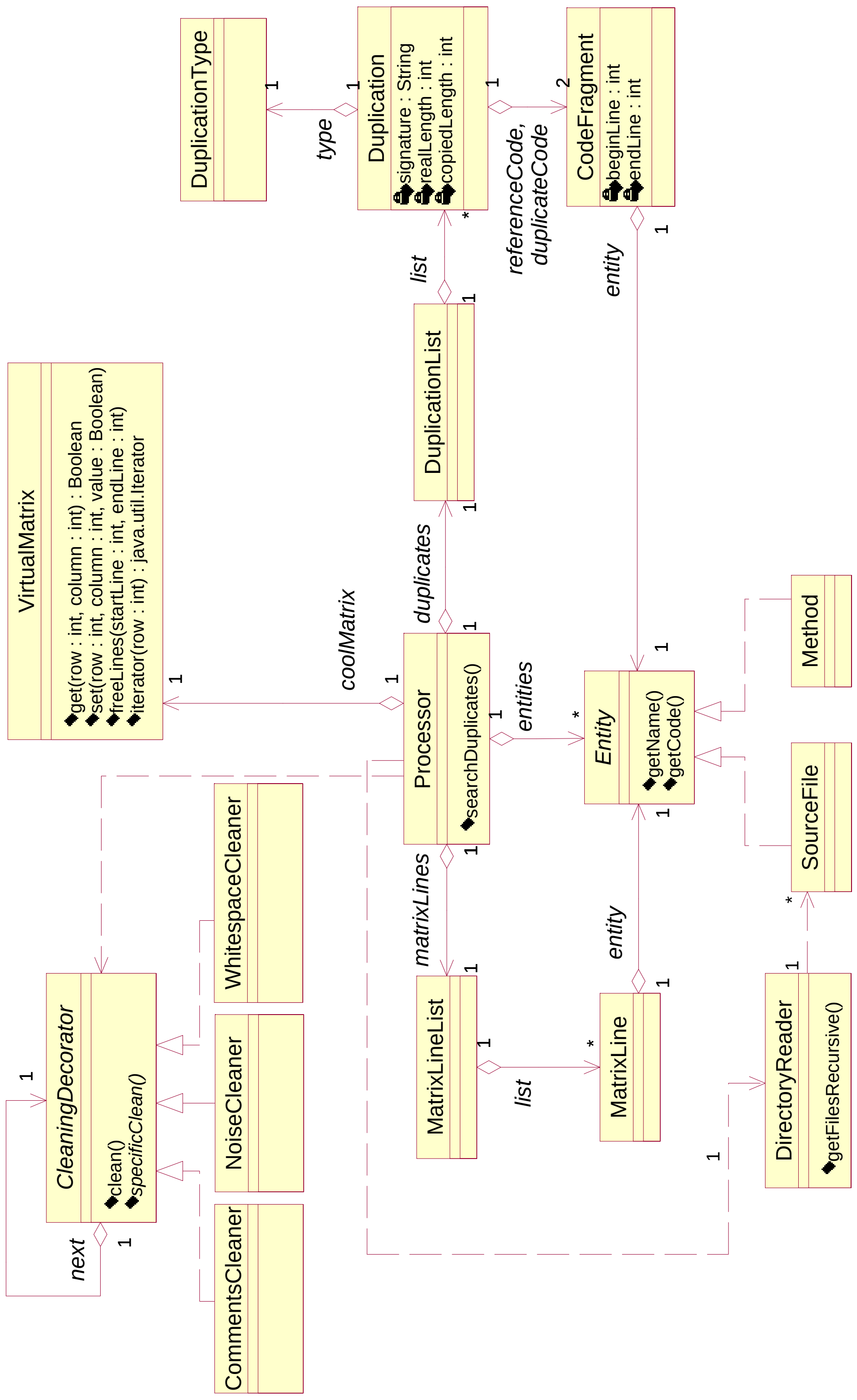


Figure 4.5: System architecture (UML)

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 *SourceFile*, which is the input for the stand-alone version of DuDe and *Method*, which is the input provided by the Insider reengineering platform, in DuDe's integrated version.

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

In the process of detection, some more classes play some roles. 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], that implements a class that cleans somehow the code. 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. If the commentCleaner option is ON, than the processor uses a chain of cleaners made of: a comments cleaner, a whitespace cleaner and a noise cleaner. The sequence diagram (fig.4.6) describes the code cleaning process in terms of time.

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. If commentCleaning is also wanted, the cleaner will be actually a chain of cleaners in the following order: CommentCleaner, WhitespaceCleaner and NoiseCleaner. 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 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 WhitespaceCleaner cleans the whitespaces and then further delegates the next cleaner to clean the code by calling it's *clean* method. The last cleaner (NoiseCleaner) cleans the lines considered noise (which can be specified in a file) 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 better manage the change of information between the *Processor* and the graphical user interface (not present on the UML class diagram), I used

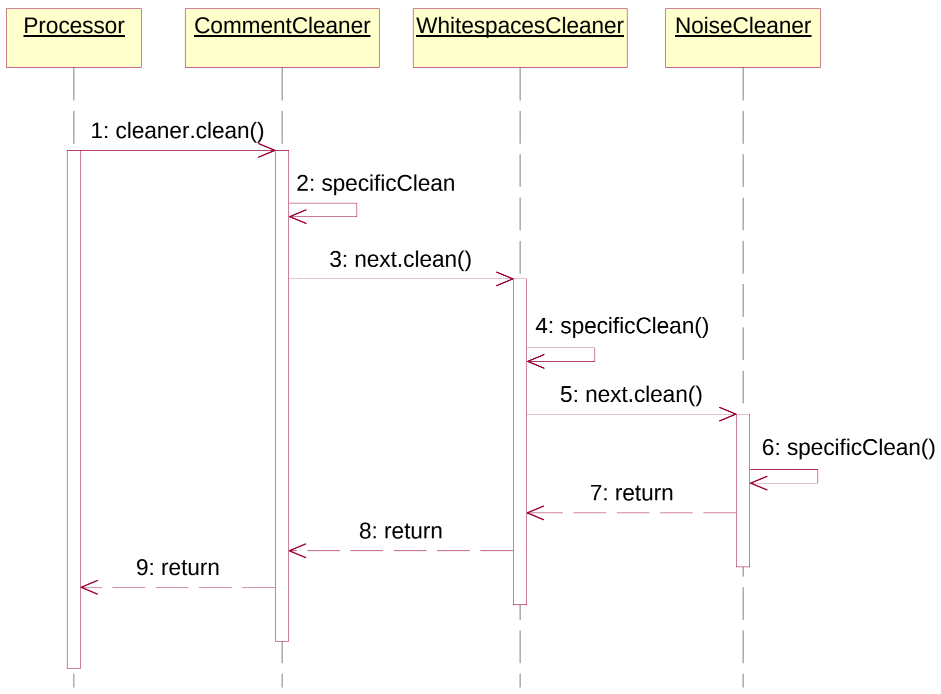


Figure 4.6: Code cleaning process (sequence diagram)

the **Observer** design pattern (presented in the Chapter 2).

## 4.3 Specific terms

A **Chain of code duplication** object is composed of:

- a pair of fragments of code (CodeFragment class in the 4.5 class diagram) that store the localization information. Each of these CodeFragment objects are described by:
  - a reference to the entity involved (source-file or method)
  - the index of the first line of the duplicate
  - the index of the last line in the duplicate
- the duplicate's type (basic types: exact, insert, delete and modified, plus a composed type)
- its signature, an original concept that captures the chain's configuration the way it would look on the scatter-plot representation. It contains in-

formation on both files and the way the code sequences evolved after the duplicating act took place.

- the copied length, which is often smaller than the length covered within the files, that is due to the cleaning process. The code in files still contains the white spaces (indentation), comments and some other "noise"

**Definition 4.3 (Exact Chunk)** *An* **exact chunk** *is a part of a duplication chain made of the maximum (greedy approach) number of consecutive copied lines of code. In other words it is the length of a copied code sequence. In a scatter-plot representation, an exact chunk is a continuous diagonal configuration of dots. The exact chunks are the raw material out of which the duplications chains are made of. In terms of exact chunks, a duplication chain is a set of exact chunks, with every two consecutive chunks linked by a number of modified (or inserted/deleted) lines (fig. 4.7).*

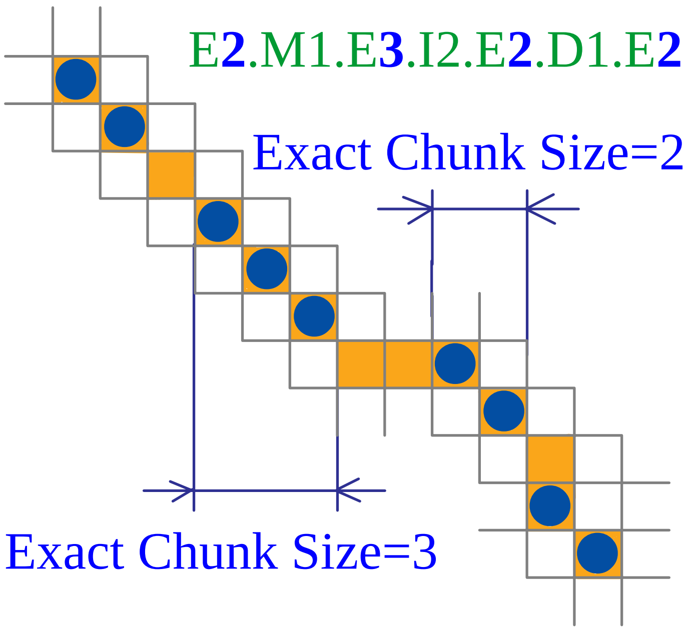


Figure 4.7: Parametrul ExactChunk

The term is directly related to the minExactChunk parameter, which filters the results, by finding only the duplication chains that contain exact chunks whose sizes are larger than the minExactChunk parameter. This parameter was introduced out of the need to avoid situation that came up on the early versions of DuDe, like: duplication chains made of two exact chunks size 2 (or even 1), linked by a number of 4 lines of code. In this case the solution is to consider it as two distinct chains of type EXACT. The exact chunks were in this case too far away from each other reported to their sizes.

**Definition 4.4 (Line Bias)** *The* **line bias** *is the number of lines of code (LOC) that link two consecutive exact chunks within a chain (Fig. 4.8). In other words, it is the distance covered in the search for a next exact chunk before giving up. This parameter was introduced because we needed to consider a duplication chain in the case that a code fragment was copied and partially modified still as a single duplication chain and not as 2 shorter clones: one before and one after the modified lines.*

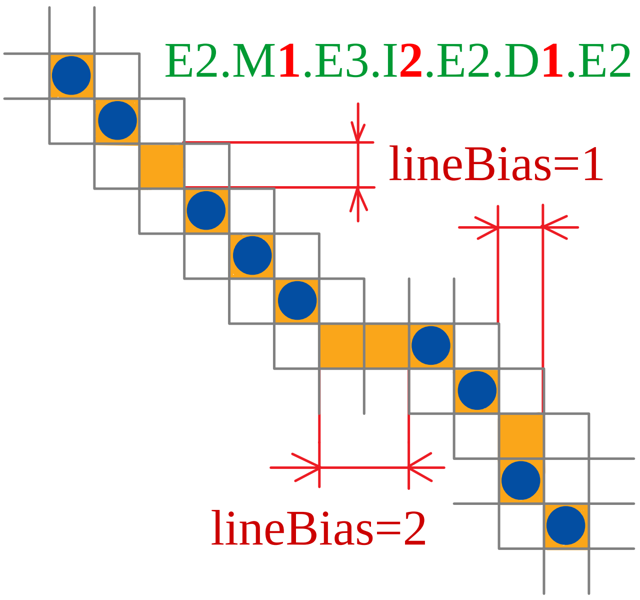


Figure 4.8: Parametrul LineBias

Its corresponding search parameter maxLineBias is the one that offers the flexibility of the searching process. By setting maxLineBias to 0, the tool will find only duplication of type EXACT.

The duplication's **Type** can be one of the following types:

1. EXACT, which represents the class of a duplication chain that is an identical duplicated code sequence, which has not suffered any modifications (fig. 4.9):

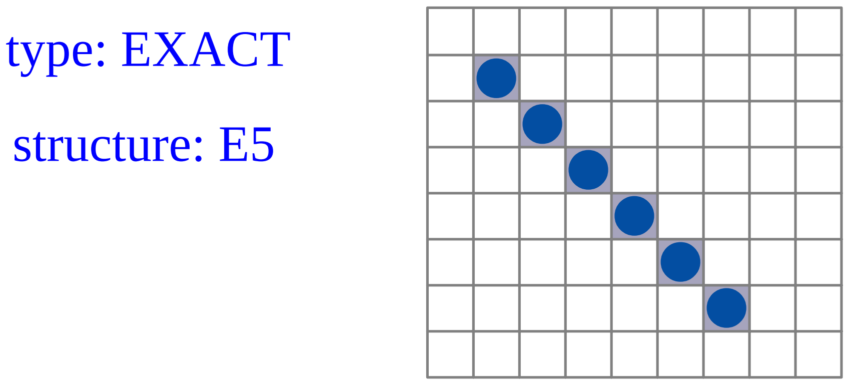


Figure 4.9: Type EXACT

2. MODIFIED, contains identical code sequence interrupted by modified lines of code (fig. 4.13):

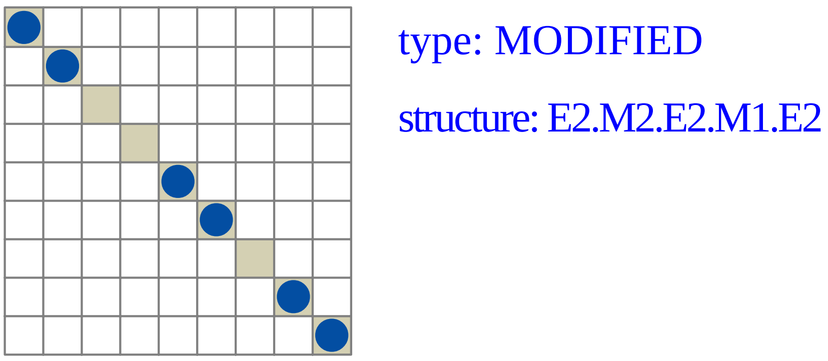


Figure 4.10: Type MODIFIED

3. INSERT, when the exact chunks within the duplicate chain are bound only by lines of code inserted in the second file (fig. 4.11):

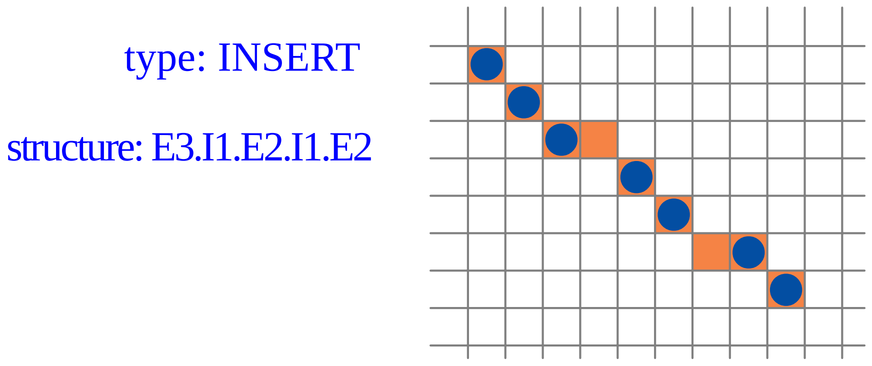


Figure 4.11: Type INSERT

4. DELETE, when its exact chunks are linked only by lines of code that don't come up in the second file, who were deleted from the first file (fig. 4.12):

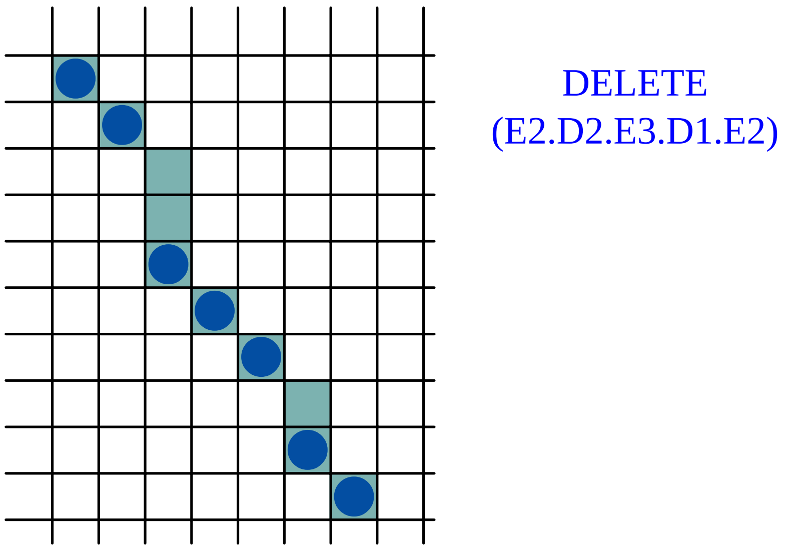


Figure 4.12: Type DELETE

5. COMPOSED, which is a combination of the other types (fig. **??**):

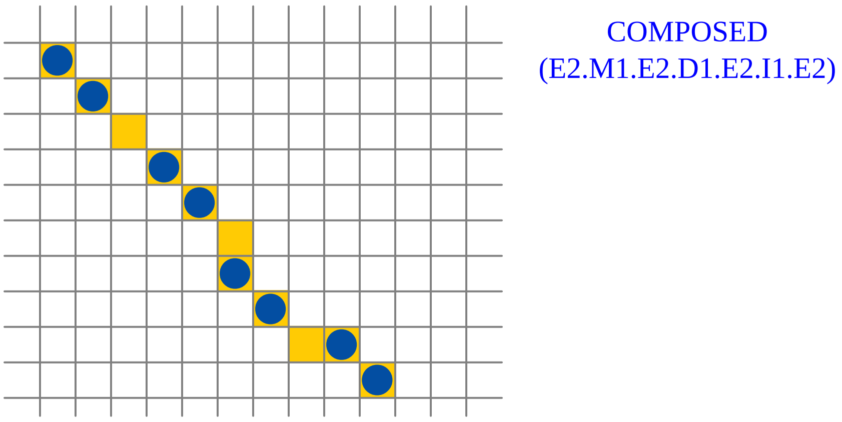


Figure 4.13: Type COMPOSED

**Definition 4.5 (Duplication Signature)** *The* **signature** *of a duplicate is the structure (dot configuration) of the chain the way we would see it if we represented the scatter-plot.*

From a duplication signature, one can deduce the type and the length.

A duplication's **length** stands for the number of relevant lines of code that compose the chain and it is calculated as the minimum between the length on the horizontal direction (length of the chain in the first entity) and the length on the vertical direction (length of the chain in the second entity). This two lengths can vary if the chain's signature contains deleted or inserted portions of code (Fig. 4.14).

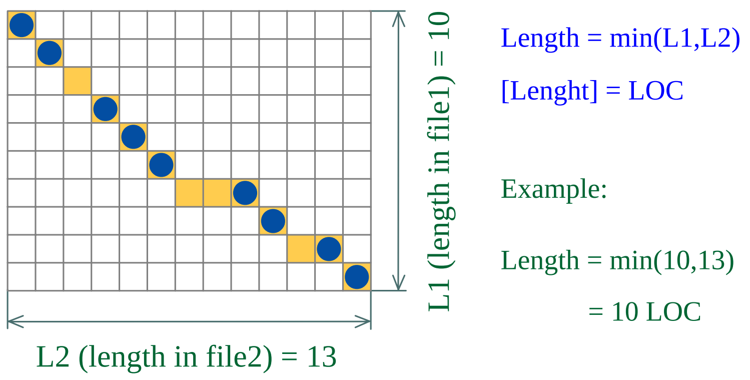


Figure 4.14: The length of a duplication chain

# Chapter 5

# Evaluation of the tool

## 5.1 Features

The graphical user interface (fig. 5.1) offers a simple, yet powerful access to the duplication chains detecting engine. The all-in-one-window integrated workspace is composed of:

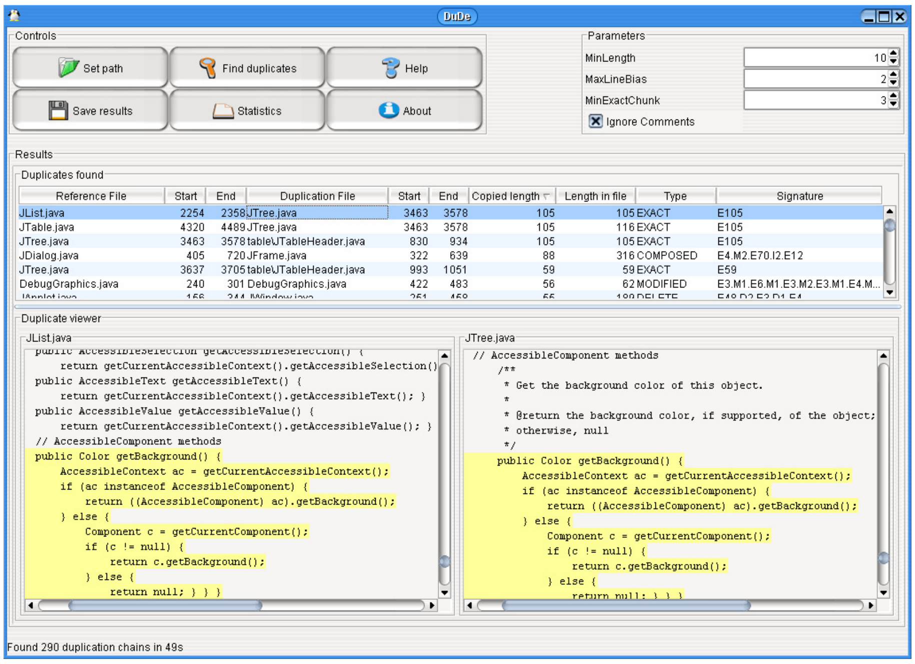
- control panel
- parameters panel
- results panel (a list of found chains)
- visualization panel, for visual analysis of the duplicated code involved in a chain
- status bar

In order to analyze a project, first you have to set the starting path (current directory) where the source files of the project are located. Then, you can modify the searching parameters and hit the Search button. The status bar contains a progress bar, visible only during a search process.

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:

- Reference File, Duplication File are the 2 files that share the duplicated code
- startLine and endLine are the line indexes in the 2 files where the duplicated code starts and ends



Parameters

| | |
|---|---|
| MinLength | 10 |
| MaxLineBias | 2 |
| MinExactChunk | 3 |
| ☒ Ignore Comments | |

| Reference File | Start | End | Duplication File | Start | End | Copied length | Length in file | Type | Signature |
|---|---|---|---|---|---|---|---|---|---|
| JList.java | 2254 | 2358 | JTree.java | 3463 | 3578 | 105 | 105 | EXACT | E105 |
| JTable.java | 4320 | 4489 | JTree.java | 3463 | 3578 | 105 | 116 | EXACT | E105 |
| JTree.java | 3463 | 3578 | table\JTableHeader.java | 830 | 934 | 105 | 105 | EXACT | E105 |
| JDialog.java | 405 | 720 | JFrame.java | 322 | 639 | 88 | 316 | COMPOSED | E4.M2.E70.I2.E12 |
| JTree.java | 3637 | 3705 | table\JTableHeader.java | 993 | 1051 | 59 | 59 | EXACT | E59 |
| DebugGraphics.java | 240 | 301 | DebugGraphics.java | 422 | 483 | 56 | 62 | MODIFIED | E3.M1.E6.M1.E3.M2.E3.M1.E4.M... |

JList.java

```
    public AccessibleSelection getAccessibleSelection() {
        return getCurrentAccessibleContext().getAccessibleSelection()
    public AccessibleText getAccessibleText() {
        return getCurrentAccessibleContext().getAccessibleText(); }
    public AccessibleValue getAccessibleValue() {
        return getCurrentAccessibleContext().getAccessibleValue(); }
    // AccessibleComponent methods
    public Color getBackground() {
        AccessibleContext ac = getCurrentAccessibleContext();
        if (ac instanceof AccessibleComponent) {
            return ((AccessibleComponent) ac).getBackground();
        } else {
            Component c = getCurrentComponent();
            if (c != null) {
                return c.getBackground();
            } else {
                return null; } } }
```

JTree.java

```
 // AccessibleComponent methods
     /**
      * Get the background color of this object.
      *
      * @return the background color, if supported, of the object;
      * otherwise, null
      */
     public Color getBackground() {
         AccessibleContext ac = getCurrentAccessibleContext();
         if (ac instanceof AccessibleComponent) {
             return ((AccessibleComponent) ac).getBackground();
         } else {
             Component c = getCurrentComponent();
             if (c != null) {
                 return c.getBackground();
             } else {
                 return null; } } }
```

Found 290 duplication chains in 49s

Figure 5.1: DuDe's Graphical User Interface

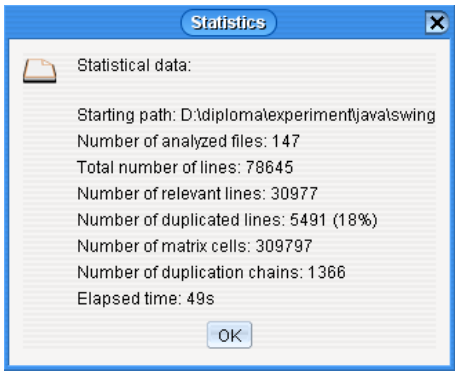
- Copied length is the length of the duplication chain
- Length in file is the number of duplicated lines of code between the start line and the end line of the duplication. it is usually greater than the Copied length, because it may contain the noise and blank lines that were rejected in the code cleaning phase, just before the analysis
- Type of the duplication chain is the chain's type (previously discussed)
- structure is a chain of <symbol><size> elements (separated by '.'), where size represents the number of lines and the symbols can be: E (exact), M (modified), I (insert), D (delete).

If you would like to save the results for subsequent analysis, you can ask DuDe to generate a report (the **Save Results** button), which stores the list of results in a specified file, with respect to the current sorting of the list.

I 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. 5.2) gathered during the last searching operation:

- number of analyzed entities,
- total number of lines of code, number of analyzed lines of code (relevant lines),
- number of duplicated lines (the ones that are part of at least one duplication chain),
- the percentage of duplicated lines
- elapsed time



```
Statistical data:

Starting path: D:\diploma\experiment\java\swing
Number of analyzed files: 147
Total number of lines: 78645
Number of relevant lines: 30977
Number of duplicated lines: 5491 (18%)
Number of matrix cells: 309797
Number of duplication chains: 1366
Elapsed time: 49s
```

Figure 5.2: 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).

During an operation, which can take between a few milliseconds and several hours (20 minutes for a 15.3 MB code, comments ignored and 2h15minutes for the same 15.3 MB project, this time analyzing comments, too), at the right bottom end of the screen there is a progress bar, which gives a hint about the how much of the whole operation has been left behind.

### 5.1.1 Parameters

- 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 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)

### 5.1.2 Controls

- Set path - sets the path where DuDe is starting its search
- Save results - saves the list of duplication chains in a specified file
- Find duplicates - starts searching for the duplicates according to the specified path and parameters
- Statistics - gives some statistical information about the results of the last search operation
- Help - shows this file
- About - version, author, date

## 5.2 Running from Insider

From Insider, DuDe searches duplications in the method bodies provided by Insider, and attaches the resulting chains to the corresponding method bodies as properties. Insider can further use these data to compute software metrics based on the duplication issue.

## 5.3 Experiment

I choose the 8 projects written in Java and C that were study cases in Bellon's paper on evaluation of clone detecting tools [Bel02]. The purpose of this experiment is to prove the applicability of DuDe towards various projects.

The 8 projects, covering the size range from up to 10 MB, are presented in fig. 5.3. The C projects have an orange background and the Java projects have a blue background, 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). K LOC column is the number of lines of code, while K SLOC is the number of lines of code excluding the comments and the blank lines (I refer them as relevant lines of code) and this is actually the set of lines of code that DuDe analyzed. The configuration for the searching was set to: minLength = 10, maxLineBias = 2, minExactChunk = 3, ignoreComments = on.

| Project Name | Language | Size (MB) | No. of Files | K LOC | K SLOC |
|---|---|---|---|---|---|
| weltab | C | 0,43 | 65 | 11 | 10 |
| cook | C | 2,68 | 590 | 80 | 50 |
| snns | C | 4,82 | 420 | 115 | 77 |
| postgresql | C | 9,52 | 612 | 235 | 153 |
| netbeans-javadoc | Java | 0,68 | 101 | 14 | 9 |
| eclipse-ant | Java | 1,43 | 178 | 35 | 16 |
| eclipse-jdtcore | Java | 6,90 | 741 | 148 | 96 |
| j2sdj1.4.0-javax-swing | Java | 8,39 | 538 | 204 | 102 |

Figure 5.3: Test projects

The second table (5.4) shows the way the different types of duplication chains were distributed among the results for every project. The first column is the total number of found duplication clusters (Total(#)), then the number of duplication chains of type exact (E), the percentage of this type, and then the number of duplication chains and the percentage for the types: modified (M), insert or delete (I/D) and finally composed (C). I grouped together the types Insert and Delete, because they are actually the same type. The only thing that makes the difference is which of the 2 files involved is considered the reference file and which one is the file where the duplication occurred.

| Project Name | Total(#) | E(#) | E(%) | M(#) | M(%) | I/D(#) | I/D(%) | C(#) | C(%) |
|---|---|---|---|---|---|---|---|---|---|
| weltab | 305 | 83 | 27,2 | 178 | 58,4 | 16 | 5,3 | 28 | 9,2 |
| cook | 894 | 677 | 75,7 | 189 | 21,1 | 12 | 1,3 | 16 | 1,8 |
| snns | 652 | 386 | 59,2 | 211 | 32,4 | 35 | 5,4 | 20 | 3,1 |
| postgresql | 485 | 117 | 24,1 | 300 | 61,9 | 26 | 5,4 | 42 | 8,7 |
| netbeans-javadoc | 29 | 10 | 34,5 | 19 | 65,5 | 0 | 0,0 | 0 | 0,0 |
| eclipse-ant | 5 | 2 | 40,0 | 2 | 40,0 | 1 | 20,0 | 0 | 0,0 |
| eclipse-jdtcore | 361 | 153 | 42,4 | 159 | 44,0 | 30 | 8,3 | 19 | 5,3 |
| j2sdj1.4.0-javax-swing | 841 | 512 | 60,9 | 312 | 37,1 | 11 | 1,3 | 6 | 0,7 |

Figure 5.4: The distribution of types

From observing the total number of duplication clusters for all the projects, we can infer that there is not necessary a relation between the size of a project and the number of clones that the programmers introduced during the life of that system. On the other hand, looking at two smallest projects from both camps (Java and C/C++), it may look as the C projects generally tend to contain more clones that the Java projects, which can be explained by the fact that

Java is a younger programming language while C is older and at its time, the concept of reusing code was far away from the concept we know nowadays.

As I started to validate some of *cook*'s duplication chains (who seemed at the first sight to be the duplication champion) I saw that 604 exact chains out of 677 were made of generated code.

An interesting thing is the fact that the C/C++ projects contained every type of duplication chain, while some of the Java projects missed the Insert/Delete types and the Composed type.

The *eclipse-ant* project has the lowest number of duplicates, though it is not the smallest project. This may mean it is well designed, at least towards the code duplication.

It is also easy to observe that the types insert/delete and composed come up much less frequently than the types exact and modified.

A thing I also cared for was the longest duplication cluster for every type (fig. 5.5), for I wanted to dive in more in the habit of copy&pasting.

| Project Name | Type | Exact | Modified | Insert/Delete | Composed |
|---|---|---|---|---|---|
| weltab | Exact | 256 | 335 | 73 | 235 |
| cook | Modified | 40 | 71 | 42 | 67 |
| snns | Exact | 338 | 148 | 91 | 149 |
| postgresql | Exact | 1122 | 130 | 51 | 57 |
| netbeans-javadoc | Modified | 36 | 88 | N/A | N/A |
| eclipse-ant | Exact | 24 | 22 | 17 | N/A |
| eclipse-jdtcore | Exact | 190 | 57 | 70 | 69 |
| j2sdj1.4.0-javax-swing | Modified | 165 | 188 | 91 | 88 |

Figure 5.5: Longest clusters

The longest duplication out of all projects was one belonging to *postgresql*, which was rather an unusual one, so I looked at it in the duplication viewer and it proved to be a header file that appeared in two different directories (copied length was 1122). The next longest clone of that project was of length 299, so the first clone was atypical for that project. The unusual cases can be sometimes explained by looking at the code.

The type of the longest duplication cluster is either exact or modified. An explanation for this may be the much higher ratio of such duplications compared with the insert/delete or even with composed. Another sad (but true) explanation lays in the fact that the copying habits involve mainly copying and pasting the code (and sometimes modifying it to adapt it to the problem) as an unhealthy way of code reutilization.

The last table presented for this experiment is one that captures some statistical data (fig. 5.6).

| Project Name | Coverage (%) | Clones per 1 K SLOC | Elapsed time(mm:ss) |
|---|---|---|---|
| weltab | 72 | 27,73 | 00:06 |
| cook | 11 | 11,18 | 02:15 |
| snns | 15 | 5,67 | 05:14 |
| postgresql | 8 | 2,06 | 20:15 |
| netbeans-javadoc | 11 | 2,07 | 00:05 |
| eclipse-ant | 1 | 0,14 | 00:14 |
| eclipse-jdtcore | 11 | 2,44 | 07:56 |
| j2sdj1.4.0-javax-swing | 7 | 4,12 | 11:23 |

Figure 5.6: Statistical info

The first column presents a metric called *coverage of clone code*, which is defined for all clones in a whole source code, not for one particular clone pair.

**Definition 5.1 (Coverage)** *The percentage of lines that include any portion of clone (percentage of lines of code contained in at least one duplication chain) is a metric called* **coverage***.*

The second columns is reserved to a the number of clones per K LOC, which measures the density of code clones on every project. The third column is a measure for the time efficiency of DuDe towards every project.

From the coverage column it is obvious that the project that is the least "infected" by clones is *eclipse-ant* with 1%, and that the presented Java projects are much better in terms of coverage than the presented C/C++ projects. 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 (72% coverage means 3/4 of the lines of code have at least one replica in the project).

The second measurement (Clones per K LOC) is completing the picture. While the 27% gives *weltab* the gold medal, the second place goes to *cook* and the third keeps its distance with 5% *cnns*. Compared to the coverage podium, there is a strange thing going on: while *cook* is on the second place on the "clones per K LOC" discipline, he does not manage to qualify so bad on coverage. That means that, although it has a big average of duplication, the clones are not homogenously distributed, but there are less lines of code duplicated with more replicas for every clone.

The elapsed times for the analyzed projects show that the searching process is directly influenced by K SLOC factor (number of relevant lines). This is an expected fact, for the most computing power is needed to build the matrix and to compare the lines, one by one, with all the others.

### 5.3.1 Case study

Out of all the projects in this experiment, I chose to analyze some of the results I got on *j2sdk1.4.0-javax-swing*, because it is well-known in the Java world and there are some interesting results related to it, that shows us that copying & pasting happens in big software companies, too.

While browsing in the list of duplication chains, the first thing that hit me was a set of 6 clones, same type (EXACT), same length (105). As I looked closer it proved that these clones belonged to the same 4 classes, in different combinations. Than, I saw that this was the case of 1 multiple clone (the same clone, which appeared in more than 2 classes). So, after analyzing this *clone class* (containing 4 instances), the results are:

- we are talking of a multiple clone of type EXACT and length 105
- the clone has got instances in 4 classes: JList (covering lines 2254 to 2358), JTree(3463-3578), JTable(4320-4489) and table.JTableHeader(830-934)
- the copied code was composed of 11 public, identical methods, disposed in exactly the same order in all of the 4 classes:

  public Color getBackground()

  public void setBackground(Color c)

  public Color getForeground()

  public void setForeground(Color c)

  public Cursor getCursor()

  public void setCursor(Cursor c)

  public Font getFont()

  public void setFont(Font f)

  public FontMetrics getFontMetrics(Font f)

  public boolean isEnabled()

  public void setEnabled(boolean b)

This is an obvious proof of bad design. The simple solution to this problem could be: creating a class that implements all the common methods and make all of the involved classes (at least JTree, JTable and JList, which belong to the same package) inherit from that class. Than these methods are written once and only once, in the superclass. Of course, it is not that simple, because all those classes already have a superclass (JComponent) and Java offers no support for multiple inheritance, but there are plenty of other solutions to avoid this ugly code duplication (the newly created class could be a subclass of JComponent).

As I looked further for such multiple clones, I concentrated my attention on the classes involved in the last clone and I found another clone of type exact, smaller in size, but involving the same 4 classes. The second clone:

- another multiple clone of type EXACT and length 48
- the clone has got instances in 4 classes: JList (2417-2464), JTree(3648-3705), JTable(4607-4654) and table.JTableHeader(1004-1051).
- the copied code comprised 6 public, identical methods, disposed in exactly the same order in all of the 4 classes:

  public void setSize(Dimension d)

  public Accessible AccessibleAt(Point p)

  public boolean isFocusTraversable()

  public void requestFocus()

  public void addFocusListener(FocusListener l)

  public void removeFocusListener(FocusListener l)

I continued my investigation and discovered another clone, this time excluding the JTree class, still with 23 LOC duplicated. Judging by these results, the 17 methods that are common to those 4 classes suggests that a refactoring would be worth the effort.

Looking at the names of the authors, it proved that every file involved had other authors. So it was not one programmer who was reusing his code, it was rather a known issue and for some reason they left the duplication code. They are probably aware of this. Maybe it's time for a system revision.

The described case study showed one possible approach to analyze a system's design by looking for clones and to find possible start points for refactoring. It also showed that even software systems that we rely on in our development process can suffer from code duplication. Though, it is not that easy to prove when it comes to software systems that are other than open-source.

## 5.4 DuDe's Iterative Development

We started to plan the development of DuDe in an iterative, step-by-step manner. The early version of DuDe had no user interface and it was a non-configurable version. The starting directory could be specified in the command line. The results were really hard to validate; for every reported duplication, the same drill had to be executed: opening both of the involved files in a text editor and searching for the start and end lines, delimiting the duplicated code fragments for both files, and finally comparing the results.

The next generations of DuDe had a command interpreter and the possibility to configure the tool by entering commands: set the starting path, searching for duplicates, show the results sorted by various attributes, save the results in a file. These first really usable versions were the ones that started a row of testings. Some of the assistants from our faculty helped by providing feedback of the results DuDe offered. This way, we found and eliminated many bugs.

Still, the problem of validating the clones was not very user-friendly. And, finally, as I saw DuDe was getting better and better, and the duplicate detecting engine evolved fine, I decided it should have a graphical user interface that would help examining the list of duplication chains, by sorting it in many ways, and through the duplicates viewer to have means to validate the clones in a more pleasant and easy way. At this point I was able to test it against different projects and to benefit of the code's proximity: I could validate the duplications in no time.

Until now, the program was tested against projects written in Java, Visual Basic, C, C++ and even on some of Shakespeare's pieces.

Integrating DuDe in the Insider platform was a problem of 3 hours of pair programming, because of the fact that we knew from the start that the tool will be integrated later on, and because of Insider's extendability.

It was an interesting, unique experience, with permanent new requirements to adapt to, with performance problems when the need to scale up to industrial needs came up that were solved after hard brainstorming sessions, with disappointing days after a solution proved to be wrong, with enthusiastic moments as the tool proved its scalability and finally, with the joy of using a program that I spent so many time working on. With everything that spices up the software development work.

## 5.5 Summary of accomplished goals

After presenting all the issues that are important for such a tool 3.4, I would like to review the way DuDe addresses all of them.

1. Regarding **scalability**, the tool presented in this thesis proved some real industrial strength. DuDe evolved favorably, by analyzing a software project with over 600.000 SLOC (lines of code, excluding comments and blanks) in 32 MB of source code, 1786 source files. The result of the searching process (with a minLength of 7, a maxLineBias of 2 and minExactChunk of 2) was a list of 52611 duplication chains. The longest chain measured 401 lines of code, and there were 62 duplication chains with length over 100.

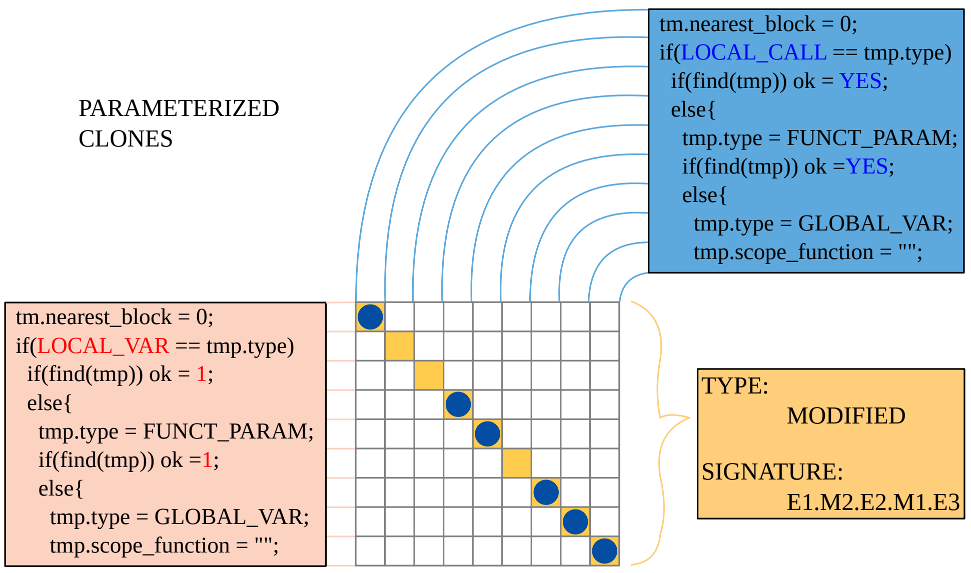
2. The **processor time** for the previously presented results was 8h:53m on a Celeron 1.7GHz processor and 2h:44m on a P4 2.8GHz processor, which is a totally acceptable processing time on a project that size.

3. And in terms of **memory**, we have the same situation: the 32 MB, 600 K SLOC project was analyzed on both computers with 512 MB RAM, which nowadays is an average value for the RAM memory.

4. The only language dependent part of the tool is reduced to the comment cleaner, so with the comment cleaning option turned off, DuDe can be applied to projects written in any programming language (even to any other text-based file). So, in this terms the tool is as **language-independent** as it gets.

5. The problem of catching duplicated code with renamed variables or method names (**parameterized clones** is also addressed (with less precision than a token-based approach, tough) by its ability to detect chains of exact copied code fragments separated by modified, inserted or deleted lines of code. With this in mind, a code fragment with renamed variables will be probably (depending of the number of occurrences and the searching parameters) detected by DuDe as a duplication chain with modified lines corresponding to the lines where the variables' names occur. An example of this case is described in fig. 5.7.

6. Evaluating the tool in terms of **recall and precision**, its recall is high and the precision is lower, when providing the list of candidates. The real clones can be further validate by visualizing the code in the duplicate viewer panel and deciding which are real clones and which are false positives. Some of the reported duplicates, although they are correct detected clones (they match) but they are not to be considered clones, for instance a sequence of *import* statements in java. The tool by itself cannot take this decision, but it offers aid (visualizing the involved code) for the human factor to decide whether a portion of duplicated code is a real clone or not, in this specific context.



```
tm.nearest_block = 0;
if(LOCAL_CALL == tmp.type)
  if(find(tmp)) ok = YES;
  else{
    tmp.type = FUNCT_PARAM;
    if(find(tmp)) ok =YES;
    else{
      tmp.type = GLOBAL_VAR;
      tmp.scope_function = "";
```

```
tm.nearest_block = 0;
if(LOCAL_VAR == tmp.type)
  if(find(tmp)) ok = 1;
  else{
    tmp.type = FUNCT_PARAM;
    if(find(tmp)) ok =1;
    else{
      tmp.type = GLOBAL_VAR;
      tmp.scope_function = "";
```

Figure 5.7: Catching a duplicate with renamed variables

7. The tool definitely does not show any signs of the **splitted duplicates symptom**. DuDe was created as a lightweight, fast tool but also configurable. DuDe has a parameter called maxLineBias that tunes the number of modified, deleted or inserted lines within a duplication chain, meant exactly to avoid that unpleasant described situation. It can be tuned to adapt the searching to the goal of the analysis. A threshold for the length of the duplication chains can also be set and the minimum exact chunk size within a chain is the parameter that tunes the balance between the distance between exact chunks and their lengths. In the code preprocessing phase, a set of lines (considered noise) specified in a file are also searched and eliminated. By that, the tool can adapt to other elements that might be considered unwanted, so it is open to modifications.

8. DuDe can not only detect the **type** of the duplication chain, but it has some more types added compared to the ones described in [Bel02]. In this thesis, I defined a number of types of duplication chains that serve the best for our purpose: exact type (type 1 in Bellon's categorizing), modified type (catches most of the time the type 2, depending of the frequency of the renamed element in the code), insert type (type 3), delete type(also type 3, depending on the file considered as reference), composed type (a type that was not described in that experiment). DuDe can find duplicates within the same class (or method body) or across-file duplicates.

9. Some of the tools did not make correct delimitation of the clones. DuDe makes **correct delimitation** of the duplicated code in the files, and this

can be verified by checking the beginning and the end of the duplicated chain to see if the lines match and if the signature is correct and then check the code portion before the beginning and the code after the end of the chain to see if indeed id does not match. These limits can be easily observed in the duplication viewer, because the duplicated code is highlighted.

Comparing to the tools based on syntax trees, the approach based on comparison of string of characters brings some immediate advantages: lower memory requirements, higher computing speed, and what seemed to be a real issue in Bellon's experiment [Bel02] was the fact that the tools that were using a language-dependent parser have encountered problems if there were syntactic errors or even some header file were missing. Because the clone-detecting tool is not meant to detect syntactic errors (there are enough IDEs to do that), the only thing that counts here are the duplications. DuDe does not care about syntactical errors, bugs or even if the analyzed project is a a whole project or just a part of it, it just finds the duplicate code.

# Chapter 6

# Conclusion

## 6.1 Review

Chapter 1 introduced the context in which the software duplications appear, with the permanently changing requirements and the high costs of the maintenance process. Then, a description of the way these clones appear in the software systems took us through wrong understanding of the "reuse" concept, wrong abstract data types use and compromising design to favor performance, even accidental clone introduction. Followed up by the problems associated to code duplication: difficulties in implementing changes, bugs copied along with the code. And the way various authors describe the duplication issue. Chapter 1 ends with solution to this problem offered by refactorings meant to eliminate the replicated code and the urgent need for tool aid in the process of analyzing software systems.

Chapter 2 makes a journey in the world of object-oriented technology with starts in the following stations: object-oriented programming and design, detection of design flaws, design patterns, extreme programming. The concepts of data abstraction, encapsulation, inheritance and polymorphism are described, because they offer means for the presented solutions to eliminate or prevent software clones.

Chapter 3 called State Of The Art starts revisiting the first papers on this issue, back in '92. Later on, we find out about the current concerns in the object-oriented software world towards the issue of code clones. The 2 workshops on this issue are introduced, along with the papers presented at these workshops. At the first workshop, a paper approaches a comparison between the clone detecting tools, a good opportunity to try to see the advantages and disadvantages brought by every solution. With these in mind, we wrote a list of desirable attributes that we would like to see on a good duplication detecting tool.

Chapter 4 makes a description of the approach taken in the current implementation. While explaining the way this tool finds the duplication chains (most of the algorithm is described in pseudo-code). There is also a short description of the evolution in terms of performance, as the need for scaling up rises. For a better understanding of the system, we presented the architecture in terms of class diagrams and sequence diagrams (UML). During this description, there were a few new terms, which are defined in this chapter.

Chapter 5 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, the integration of DuDe in a reengineering platform called Insider. Then we present the results of an experiment, conductes on 8 projects (Java and C/C++). And, finally, we review the way DuDe accomplished the goals specified as the list of desirable features from chapter 3.

This final chapter will be a conclusion to this paper, with pros and cons for the presented tool, with the evaluation of the authors contribution and some perspectives on future work.

## 6.2 Pros and Cons

### 6.2.1 Pros

- program portability (since the entire code is written in Java)
- language independency (the white spaces removing operations do not require a lexical analyzer)
- flexibility, through parameterizing the searches
- scalability, mainly obtained by dividing the matrix into areas
- performance - good speed performance and efficient memory management
- it can analyze a project even if it contains lexical errors, while a tool based on syntax tree will fail to build it
- the tools can take decisions by itself - no need for trained users and can detect duplication otherwise easy to overlook (doesn't require so much of the user's attention)

### 6.2.2 Cons

- simplicity (no complicated parsing algorithms)
- it's not capable of detecting variable renames or equivalent structures

- cannot detect replicated functionality, which doesn't reflect into replicated syntax (i. e. a for cycle equivalent to a while cycle).

## 6.3 Evaluation of Contributions

This work's apport of originality is brought by defining a number of new concepts:

1. Chain of duplication (or cluster)
2. Signature of a duplication chain

Besides that, the tool proved its ability to handle industrial-size systems, in acceptable processing times and memory needs. Its strength are:

- scalability
- integrability
- performance

## 6.4 Future work

Plug-in for a well-known IDE (IDEA or Eclipse). This way, the tool could gain in popularity.

Possible development on the direction of parsing the code and replacing the variable names with a generic name. This way, it could also detect parameterized clones (type 2 in Bellon's classification).

Code highlighting based on the structure and type of the duplication chain.

Studying correlation possibilities between duplication types or structures and solutions to eliminate the duplication, based on the refactoring mechanisms described in chapter 2.

# 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.

[Bak95] Brenda S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the second IEEE Working Conference on Reverse Engineering (WCRE)*, pages 86–95, July 1995.

[BBC$^+$99] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. *The FAMOOS Object-Oriented Reengineering Handbook.* European Union under the ESPRIT program Project no. 21975 (FAMOOS), 1999.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 2000.

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

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

[BYM$^+$98b] 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.

[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.

[Hel95] Jonathan Helfman. Dotplot Patterns: a Literal Look at Pattern Languages. *TAPOS*, 2(1):31–41, 1995.

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

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison Wesley/ACM Press, Reading, Mass., 1992.

[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.

[KG03a] Cory Kapser and Michael W. Godfrey. A taxonomy of clones in source code: The re-engineers mostwanted list. 2003.

[KG03b] 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.

[LPM$^+$97] Bruno Laguë, Daniel Proulx, Ettore M. Merlo, Jean Mayrand, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of ICSM (International Conference on Software Maintenance)*. IEEE, 1997.

[Mar02a] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. Ph.D. thesis, Department of Computer Science, "Politehnica" University of Timişoara, 2002.

[Mar02b] R.C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall: 1st Edition, 2002.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[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.

[RD98] Matthias Rieger and Stéphane Ducasse. Visual detection of duplicated code. In Stéphane Ducasse and Joachim Weisbrod, editors, *Proceedings ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, number 6/7/98 in FZI Report. Forschungszentrum Informatik Karlsruhe, 1998.

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