# How Developers Copy

**In Proceedings of International Conference on Program Comprehension (ICPC 2006), pp. 56–65**

Mihai Balint[1], Tudor Gîrba[2] and Radu Marinescu[1]

[1]LOOSE Research Group, Politehnica University of Timişoara, Romania
[2]Software Composition Group, University of Berne, Switzerland

## Abstract

*Copy-paste programming is dangerous as it may lead to hidden dependencies between different parts of the system. Modifying clones is not always straight forward, because we might not know all the places that need modification. This is even more of a problem when several developers need to know about how to change the clones. In this paper, we correlate the code clones with the time of the modification and with the developer that performed the modification to detect patterns of how developers copy from one another. We develop a visualization, named* Clone Evolution View[1]*, to represent the evolution of the duplicated code. We show the relevance of our approach on several large case studies and we distill our experience in forms of interesting copy patterns.*

**Keywords:** code duplication, reverse engineering, software evolution, visualization

## 1 Introduction

Code duplication appears because of several reasons: time pressure, shortcomings of the programming languages, inexperienced developers *etc.* No matter the reason, duplicating the code introduces implicit dependencies between the duplicated parts (named clones hereafter) because changing one part requires changing the other parts as well.

Detecting code clones has long been in the center of the reverse engineering community [1, 6, 10, 12]. Most of the work in this area has been focused on the detection of problems in the code and in relating clones with their impact in terms of quality assurance.

Kim *et al.* made an interesting observation by experimentally showing that “refactorings may not always improve the software with respect to clones” [13]. Indeed, their empirical study showed that the short-lived clones are sometimes not worthwhile extensive refactoring, while the long living ones are often in the system due to shortcomings of the programming language. In the same line, Cordy observed that in special cases changing the clones is not advisable from a risk management point of view [4]. However, typically, extensive code duplication is clearly related to problems in maintenance [5, 11, 8].

One aspect that was neglected in the past when analyzing clones and their impact on quality is the *authorship of clones*; in other words, how developers create and maintain clones. As stated by Conway the shape of an organization influences the shape of the code structure [3]. Therefore, we believe that identifying and understanding the patterns by which a team of developers deals with duplicated code, will provide us with a better understanding of both a project’s structure and of its development team, and ultimately on the cloning phenomenon [9].

In this paper we aim to provide a novel view on clone detection, by focusing on how developers create and maintain clones, and integrating this knowledge with further information about the clones (*e.g.,* time of cloning, location of clones *etc.*). In order to detect who copies from whom we need the information of who changed each line of code involved in duplication. For that, instead of just analyzing the actual code, we analyze the information provided by versioning systems and recover for each line of code, the date and the developer that changed that line.

To get insight into what are the patterns of how the developers create and modify clones, we decompose the clone analysis along several axes: the developers that created the lines, the time of the modifications, the localization of the clones in the system, and the size of the clones.

To analyze the above variables we develop a visualization named *Clone Evolution View*. The visualization displays the details of how a code fragment is cloned into several places. Based on it we detect several copy-patterns. For example, we detect places where a developer forgets to change all the clones, clones that are changed by several developers, fresh clones *etc.*

[1] The visualization presented here makes heavy use of colors. Please obtain a color copy of this article for better understanding.

Typically in a software system we can have several hundreds such duplicated fragments. To help the reverse engineer cope with the large amount of data, we also introduce several measurements that can be used to detect interesting clones.

We apply our approach on three large case studies and we present the experience in the form of a list of copy patterns.

**Structure of the paper.** In the next section we briefly talk about the detection technique we use. Section 3 introduces our visualization of the clones. In Section 4 we present evidence of several copy patterns detected in two large case studies. Section 5 presents a set of metrics that we use to guide our search through the clone space. Section 6 discusses the different variation points of the approach. Section 7 presents the related work and Section 8 concludes the paper.

## 2 Detecting How Developers Copy

In this section we describe our approach of analyzing how developers copy. First, we describe how we automatically detect duplicated fragments in several places, and afterwards we describe how the developer information is added to the analysis.

### 2.1 Detecting Clones

The most popular approach to comprehend clones is using a scatterplot: it displays a matrix where the element $(i, j)$ is black if the line on the row $i$ is the same as the line on the column $j$ [6, 10, 12]. Figure 1 shows the details of such a scatter plot.

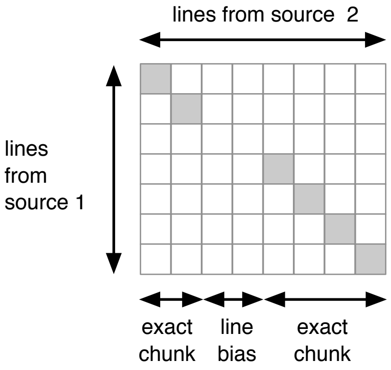


**Figure 1. Detecting clones on the scatterplot.**

To find duplicated fragments of text using the scatterplot we need to look through the scatterplot for patterns of diagonal rows of dots. If such a row of dots can be found it means that the text represented by the rows is the same (or very simillar) to the text represented by the columns.

For our analysis, we needed an automated approach. The approach is based on the quantification of the visual detection using thresholds. Three thresholds are used based on the work of Wettel [22, 23]:

- *Minimum clone length* defines the minimum amount of lines present in a clone. In our experiments we used 7.
- *Maximum line bias* defines the maximum amount of lines in between two exact chunks. In our experiments we used 2.
- *Minimum chunk size* defines the minimum amount of lines of an exact chunk. In our experiments we used 3.

The problem with the scatterplot approach is that once we find a diagonal pattern we are only looking at two pieces of text at once (*i.e.,* a *duplication*). For some systems, the number of such duplications between two fragments of text can be up to several thousands. However, in such large systems it is common to find the same fragment of text copied over and over a few times.

In our approach we have chosen to aggregate these duplicated text fragments into sets which we call *multiplications* or *clone families* [19]. Unlike duplications which describe a cloning relationship only between two fragments, a multiplication describes the cloning relationship between multiple fragments of text.

Because we automatically detect duplications by analyzing the scatterplot, we can also detect the multiplications. In a nutshell, if A, B and C name three fragments of text and A is duplicated by B, and B is duplicated by C then (A,B,C) are part of the same multiplication. With this aggregation technique we are able to lower (sometimes by more than half) the number entities we have to analyze [19].

For example, using the above mentioned values for the thresholds and this aggregation technique, the result of the automated detection is a list of multiplications where the detected clones will be at least 7 lines long, will have at most two lines between exact chunks and the exact chunks will be at least 3 lines long.

We also strive to reduce the number of detected duplications by eliminating some of the false positives generated by source code: duplications in programming language syntax (*e.g.,* java import statements), comments, text formatting. This is done by a simple parser that eliminates certain strings that match regular expressions [18].

### 2.2 Improving the Clone Detection with Developer Information

To identify how the developers create the clones, we make use of cvs annotate. In Figure 2 we give an exam-

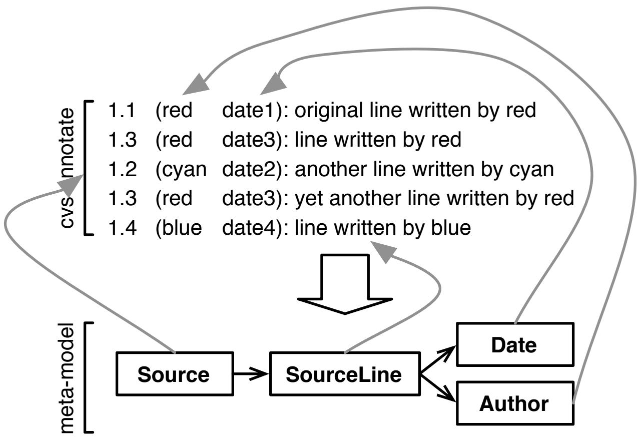

cvs annotate
1.1 (red date1): original line written by red
1.3 (red date3): line written by red
1.2 (cyan date2): another line written by cyan
1.3 (red date3): yet another line written by red
1.4 (blue date4): line written by blue
meta-model
Source
SourceLine
Date
Author

**Figure 2. Example of a result of a** cvs annotate **and the meta-model that describes it.**

ple of a fictive result of running cvs annotate on a file. Each line is annotated with the version information (*e.g.,* 1.1), the name of the user (*e.g.,* red), the date of the modification (*e.g.,* date 1).

Prior to applying the clone detection, we first process the annotations and create a model that conforms to the meta-model in the lower part of the figure. With this information at hand we can analyze how developers copy.

## 3 Visualizing How Developers Copy

There are myriads of questions that pop-up while trying to identify patterns that occur during the process of cloning. In our view most of these questions address one of the following four categories of problems:

- *Quantity e.g.,* How large are the clones? And how many clones are there in a multiplication?
- *Time e.g.,* When were the changes made to the clones? Where there inconsistent modifications?
- *Developers e.g.,* Who created the original fragment? Who copied from the original? How many developers are involved in cloning?
- *Localization e.g.,* Are clones in the same file, or in separate files?

### 3.1 Clone Evolution View

Because our aim is to explore an unexplored territory, we want to be exposed to as many variables as possible to answer the above questions. That is the reason why we design a novel visualization: the *Clone Evolution View*. On the one hand we need to give indication for when changes were made and by whom, and on the other hand we need to give an indication of what is the text of the clones. Figure 3 shows an example of our visualization.

Each line represents a source line. We attach a unique color to each developer present in the duplication, and we color each line with the color of the developer that changed it last. We can safely use a distinct color for each developer because usually, the number of developers participating in a multiplication is low (in our experiments we encountered at most 5 developers).

The visualization has two parts: on the left, we show a time representation of the modification of the line, and on the right we show the text of the line as obtained from the last version. The date of the modification of the line is shown with a vertical line annotated with the actual date.

### 3.2 How to Read the Clone Evolution View

In Figure 3 we display 3 fictive clones. Red wrote the original line of the multiplication at date 1 in the first two clones. At a later date (*i.e.,* date 2) Blue added another line in the two clones. At date 3 Red added two more lines, and created another complete clone in source 2. At date 4, Pink added another line, but only in 2 clones, and very soon after, Orange intervened and added the missing line to the third clone.

Based on the visualization we develop a vocabulary to describe the *activities* that may occur while cloning. Each activity can be further accompanied by several *attributes* that characterize in more detail the general context (and impact) of the activity.

**Typical Developer Activities.** We identified three main activities that in our experience are the most encountered ones, namely:

- *Line Cloning* denotes the act of introducing one new line to the multiplication. In *Clone Evolution View* we see that this activity happened if we see a line starting in one of the clones, and by *looking left* in the visualization (*i.e.,* earlier in time) at all the other clones, there is not another clone where the corresponding line has already been started. For example, in Figure 3, at date 4, the action of the Pink author is a *Line Cloning* as in none of the other clones that lines started by Pink has not been started at an earlier date.
- *Block Cloning* denotes a simultaneous creation of a significant number of lines from a clone, or even of an entire clone. In our example, Red created the entire

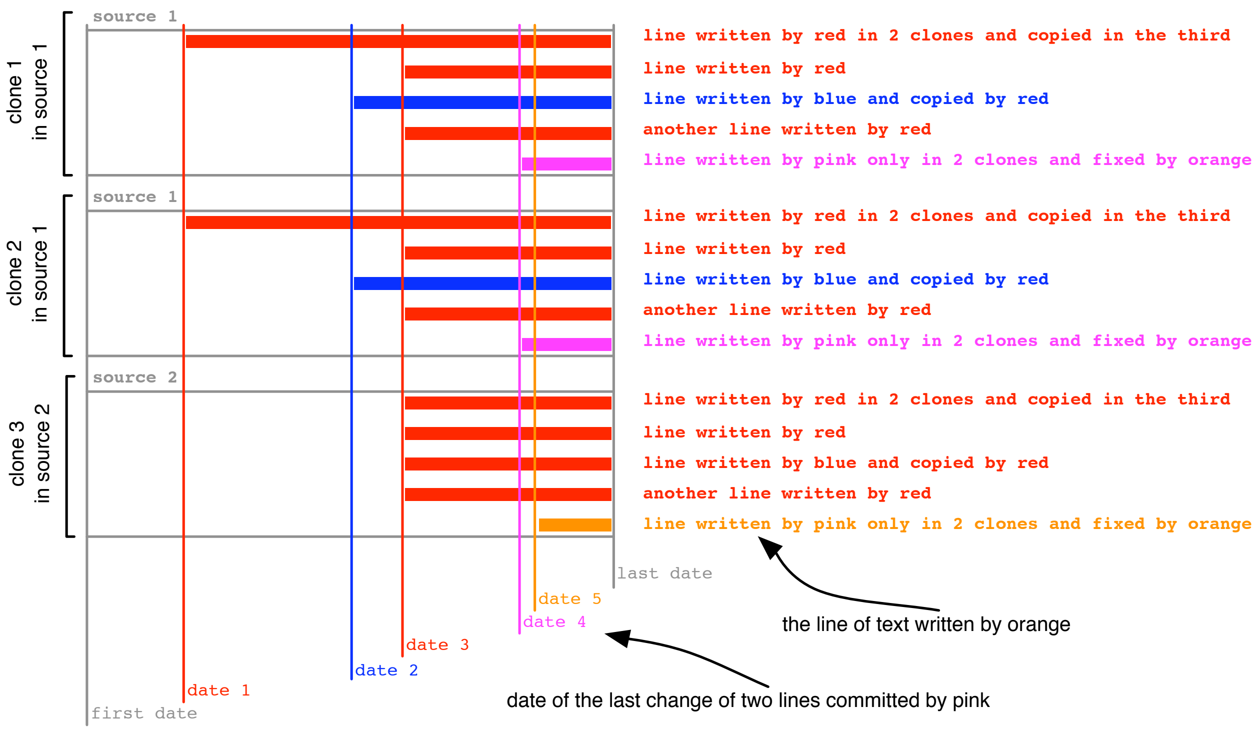

source 1
clone 1 in source 1
line written by red in 2 clones and copied in the third
line written by red
line written by blue and copied by red
another line written by red
line written by pink only in 2 clones and fixed by orange
source 1
clone 2 in source 1
line written by red in 2 clones and copied in the third
line written by red
line written by blue and copied by red
another line written by red
line written by pink only in 2 clones and fixed by orange
source 2
clone 3 in source 2
line written by red in 2 clones and copied in the third
line written by red
line written by blue and copied by red
another line written by red
line written by pink only in 2 clones and fixed by orange
last date
date 5
date 4
date 3
date 2
date 1
first date
the line of text written by orange
date of the last change of two lines committed by pink

**Figure 3. Example of the Clone Evolution View of a fictive multiplication.**

third clone by copying the lines existent in the other two.

- *Line Fixing* denotes the act of cloning at a later time then the original line, and at a later time then the clone birth. In our example, the action of ORANGE (date 5) is a *Line Fixing* as it fixes the lines committed by PINK.

**Consistency Attributes.** For the above actions we can take into account several attributes. From our empirical observations the attribute that is most relevant from the point of view of the maintenance impact is the *consistency*. Thus, each activity is characterized by one of the two attributes:

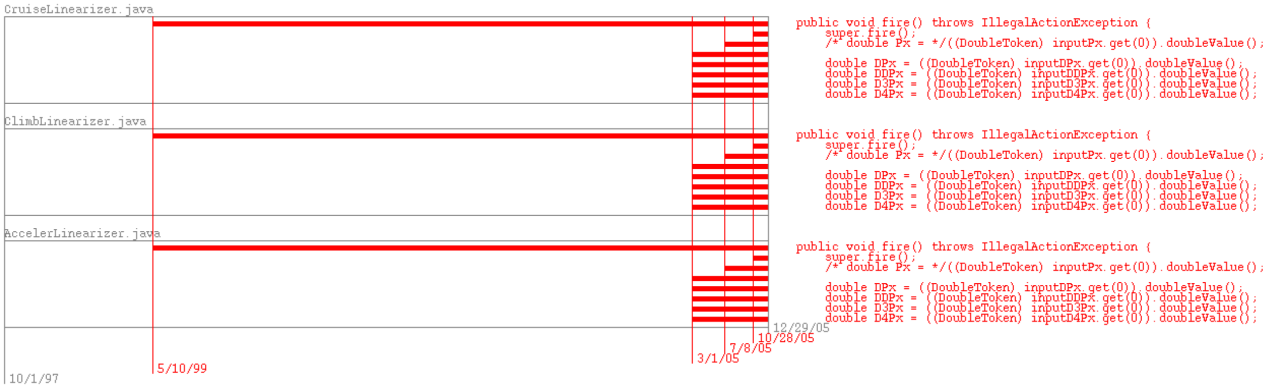
- *Consistent* – denotes a cloning (either of a line or of a block) that propagates the involved lines into all the clones that exist at the moment of cloning and additionally in all the clones that appear afterwards, at the very moment of the clone birth. For example, in Figure 3 BLUE makes a *Consistent Line Cloning* because at date 2 when he or she initiates a *Line Cloning* activity, the author propagates it in the other clone existing at that time, and also because at date 3, in the context of a *Block Cloning* activity that gives birth to the third clone, that line is also cloned, by RED.
- *Inconsistent* – denotes a cloning that does not get propagated to all the existing clones. Additionally, a cloning is considered to be inconsistent if the involved lines are not propagated in the clones that appear later at the very moment of their birth. In our example, PINK is responsible for an *Inconsistent Line Cloning* activity (date 4) as the cloning was not propagated in the third clone, and because of that at date 5 we notice a *Line Fixing* activity done by ORANGE.

Speaking about the consistency attributes we should note that the *Line Fixing* activity is always the result of an earlier *inconsistent* cloning activity.

Having an explicit vocabulary has a twofold advantage: on the one hand it offers the basic terms in which to express the reasoning, and on the other hand it provides a communication means. The vocabulary presented here comes from our experience, and it covers the interesting cases we discovered.

## 4 Patterns of Cloning Activity

In this section we present the results of our experiments on the file system of three case studies: Ant, ArgoUML and Ptolemy2. The aim of our experimental study was to empirically detect *cloning activity patterns*. Table 1 shows



```
public void fire() throws IllegalActionException {
    super.fire();
    /* double Px = */((DoubleToken) inputPx.get(0)).doubleValue();

    double DPx = ((DoubleToken) inputDPx.get(0)).doubleValue();
    double DDPx = ((DoubleToken) inputDDPx.get(0)).doubleValue();
    double D3Px = ((DoubleToken) inputD3Px.get(0)).doubleValue();
    double D4Px = ((DoubleToken) inputD4Px.get(0)).doubleValue();

public void fire() throws IllegalActionException {
    super.fire();
    /* double Px = */((DoubleToken) inputPx.get(0)).doubleValue();

    double DPx = ((DoubleToken) inputDPx.get(0)).doubleValue();
    double DDPx = ((DoubleToken) inputDDPx.get(0)).doubleValue();
    double D3Px = ((DoubleToken) inputD3Px.get(0)).doubleValue();
    double D4Px = ((DoubleToken) inputD4Px.get(0)).doubleValue();

public void fire() throws IllegalActionException {
    super.fire();
    /* double Px = */((DoubleToken) inputPx.get(0)).doubleValue();

    double DPx = ((DoubleToken) inputDPx.get(0)).doubleValue();
    double DDPx = ((DoubleToken) inputDDPx.get(0)).doubleValue();
    double D3Px = ((DoubleToken) inputD3Px.get(0)).doubleValue();
    double D4Px = ((DoubleToken) inputD4Px.get(0)).doubleValue();
```

**Figure 4. Case I: Consistent Block Cloning with Unique Author (Ptolemy2)**

the size of the case studies (cloner denotes a developer that copied at least on line of code).

| System | Files | Multiplications | Developers | Cloners |
|---|---|---|---|---|
| Ant | 1115 | 156 | 30 | 11 |
| ArgoUML | 1446 | 326 | 30 | 12 |
| Ptolemy2 | 2715 | 902 | 63 | 22 |

**Table 1. Case studies**

Next we are going to discuss the most interesting patterns discovered so far. We considered a pattern to be interesting, and selected it for presentation if it was either very frequently encountered, or if we considered it to have a particular negative impact from the maintenance point of view.

### 4.1 Case I: Consistent Line/Block Cloning with Unique Author

In this case one single developer is creating and consistently maintaining a number of clones. Usually the number of clones in the multiplication is not too high (usually between 3 and 6). This pattern may appear in several forms: either the first clone (*i.e.,* the original code) was developed at some point in time, and then is later multiplied in several clones (either simultaneously or one by one) by *Consistent Block Clonings*; or, all the clones are created synchronously (in this case we cannot say which one the original is), as seen in Figure 4. In both cases, one common characteristic is that after the clones are created, all the modifications are permanently propagated in all the clones.

Due to the consistency of change propagation and to the fact that all the clones are one-minded this is the least harmful case encountered.

### 4.2 Case II: Creation of Clones by Multiple Authors using Consistent Block Cloning

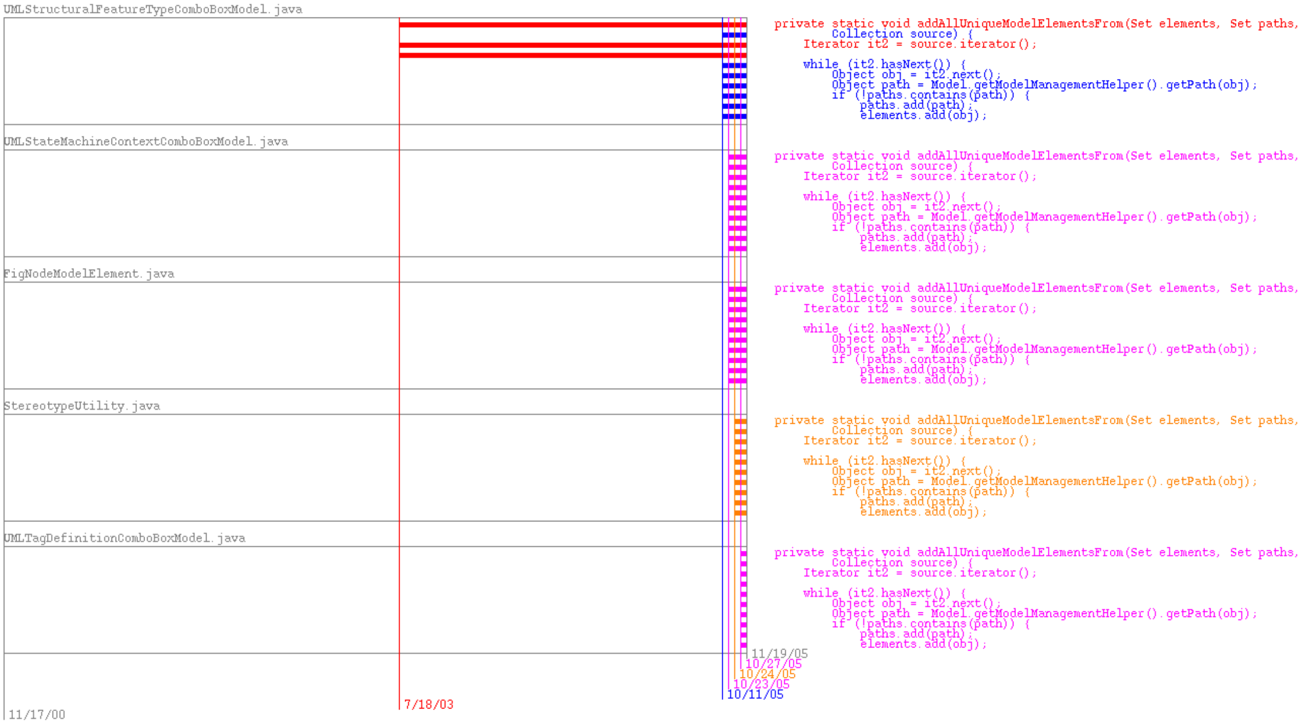
In this second pattern, we encounter at least two developers (but usually the number of involved developers is 3 or 4). The other main characteristic of this case is that the *Consistent Block Cloning* activity leads to the birth of a new clone that is created by a single developer, who is usually different from the authors of the original piece of code, just like in Figure 5. In this case the number of clones belonging to the same multiplication is rather high (usually between 4 and 7) and the size of the cloned block is also significant (higher that 7 lines). Another common characteristic is the fact that there is almost no further change in the multiplication after the *block cloning creation* of the other clones.

The consistency of the cloning activity makes this case to look harmless, yet the risk of a future inconsistent modification is strongly increased by the large number of distinct developers involved, by the tendency to have larger clones and by the number of clones created.

### 4.3 Case III: Consistent Line/Block Cloning with Multiple Authors

In Figure 6 we see the illustration of this third case of consistent line cloning. In this case there is again more than one developer involved in creating and keeping consistent the multiplication. But in this case, the clones tend appear and to be modified at the same time, while each developer takes responsibility for the creation and consistent propagation of a subset of lines. In other words, the multiplication involves several developers (usually either two or three), but each of them propagates the cloning activity in all the clones.

A further interesting example is depicted in Figure 7 where after several step-wise creation and modifications of the first three clones, made by the BLUE, RED and PINK



```
private static void addAllUniqueModelElementsFrom(Set elements, Set paths,
        Collection source) {
    Iterator it2 = source.iterator();

    while (it2.hasNext()) {
        Object obj = it2.next();
        Object path = Model.getModelManagementHelper().getPath(obj);
        if (!paths.contains(path)) {
            paths.add(path);
            elements.add(obj);
```

```
private static void addAllUniqueModelElementsFrom(Set elements, Set paths,
        Collection source) {
    Iterator it2 = source.iterator();

    while (it2.hasNext()) {
        Object obj = it2.next();
        Object path = Model.getModelManagementHelper().getPath(obj);
        if (!paths.contains(path)) {
            paths.add(path);
            elements.add(obj);
```

```
private static void addAllUniqueModelElementsFrom(Set elements, Set paths,
        Collection source) {
    Iterator it2 = source.iterator();

    while (it2.hasNext()) {
        Object obj = it2.next();
        Object path = Model.getModelManagementHelper().getPath(obj);
        if (!paths.contains(path)) {
            paths.add(path);
            elements.add(obj);
```

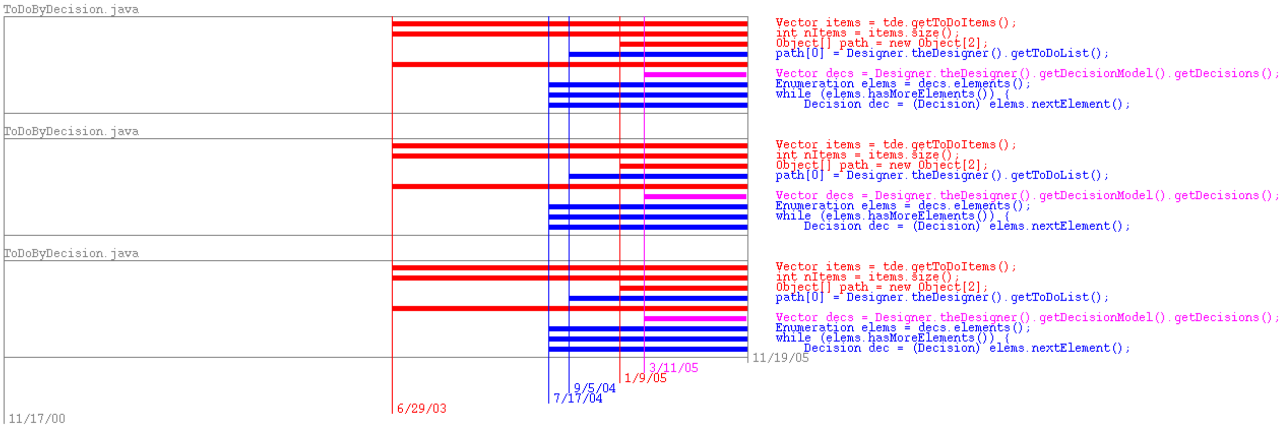
```
private static void addAllUniqueModelElementsFrom(Set elements, Set paths,
        Collection source) {
    Iterator it2 = source.iterator();

    while (it2.hasNext()) {
        Object obj = it2.next();
        Object path = Model.getModelManagementHelper().getPath(obj);
        if (!paths.contains(path)) {
            paths.add(path);
            elements.add(obj);
```

```
private static void addAllUniqueModelElementsFrom(Set elements, Set paths,
        Collection source) {
    Iterator it2 = source.iterator();

    while (it2.hasNext()) {
        Object obj = it2.next();
        Object path = Model.getModelManagementHelper().getPath(obj);
        if (!paths.contains(path)) {
            paths.add(path);
            elements.add(obj);
```

**Figure 5. Case II: Creation of Clones by Multiple Authors using Consistent Block Cloning (ArgoUML)**



```
Vector items = tde.getToDoItems();
int nItems = items.size();
Object[] path = new Object[2];
path[0] = Designer.theDesigner().getToDoList();

Vector decs = Designer.theDesigner().getDecisionModel().getDecisions();
Enumeration elems = decs.elements();
while (elems.hasMoreElements()) {
    Decision dec = (Decision) elems.nextElement();
```

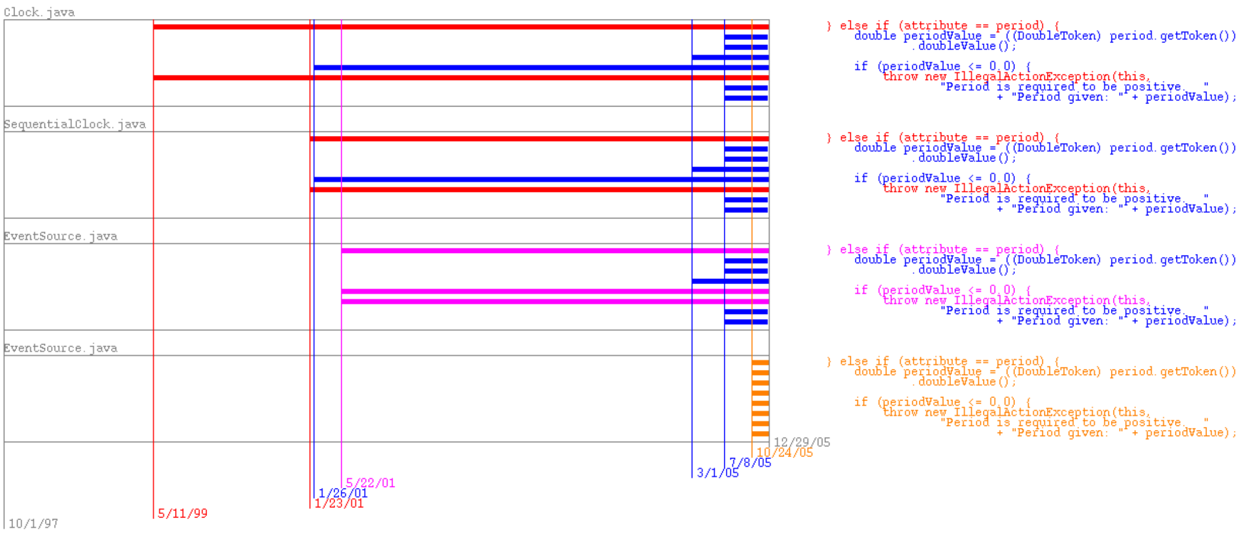
```
Vector items = tde.getToDoItems();
int nItems = items.size();
Object[] path = new Object[2];
path[0] = Designer.theDesigner().getToDoList();

Vector decs = Designer.theDesigner().getDecisionModel().getDecisions();
Enumeration elems = decs.elements();
while (elems.hasMoreElements()) {
    Decision dec = (Decision) elems.nextElement();
```

```
Vector items = tde.getToDoItems();
int nItems = items.size();
Object[] path = new Object[2];
path[0] = Designer.theDesigner().getToDoList();

Vector decs = Designer.theDesigner().getDecisionModel().getDecisions();
Enumeration elems = decs.elements();
while (elems.hasMoreElements()) {
    Decision dec = (Decision) elems.nextElement();
```

**Figure 6. Consistent Line/Block Cloning with Multiple Authors (ArgoUML)**



```
} else if (attribute == period) {
    double periodValue = ((DoubleToken) period.getToken())
            .doubleValue();

    if (periodValue <= 0.0) {
        throw new IllegalActionException(this,
                "Period is required to be positive.  "
                + "Period given: " + periodValue);
```

```
} else if (attribute == period) {
    double periodValue = ((DoubleToken) period.getToken())
            .doubleValue();

    if (periodValue <= 0.0) {
        throw new IllegalActionException(this,
                "Period is required to be positive.  "
                + "Period given: " + periodValue);
```

```
} else if (attribute == period) {
    double periodValue = ((DoubleToken) period.getToken())
            .doubleValue();

    if (periodValue <= 0.0) {
        throw new IllegalActionException(this,
                "Period is required to be positive.  "
                + "Period given: " + periodValue);
```

```
} else if (attribute == period) {
    double periodValue = ((DoubleToken) period.getToken())
            .doubleValue();

    if (periodValue <= 0.0) {
        throw new IllegalActionException(this,
                "Period is required to be positive.  "
                + "Period given: " + periodValue);
```

**Figure 7. An example of a complex cloning activity pattern, involving Case III, followed by Case II (Ptolemy2)**

authors, using the currently described pattern, we see the ORANGE author eventually acting in conformity with the previous pattern (*i.e., Creation of Clones by Multiple Authors using Consistent Block Cloning*) by adding suddenly a completely new clone to the multiplication. This example reveals that beyond each of the identified patterns, in ”real-life” cloning activities tend to appear as complex combinations of these (or other patterns).

Concluding, we may say that this pattern is less harmful than the previous one, due to the fact that all the clones tend to live their lives in a perfectly synchronized manner. Furthermore, the fact that only a small number of developers are involved makes the process of keeping the clones consistent manageable.

**Remark:** We believe that beyond its utility for the understanding of cloning activities, this pattern has also a remarkable side effect: it might reveal within an organization small cohesive teams of developers that work together in a one-minded manner.

### 4.4 Case IV: Inconsistent Line Cloning Fixed by the Same Author

The fact that a cloned line controlled by a single author is not a guarantee that inconsistencies will not appear during cloning activities. This pattern captures the case where one or more lines cloned by a single author are not consistently propagated in all the other existing clones, and it is only after a while that the initial author remembers (or finds out) about the inconsistency and reestablishes himself the consistency of the clones at that later moment in time. For example, in Figure 8 we see that the BLUE author keeps the consistency of the clones during his/her first two cloning activities, but fails to do so the third time (6/24/04) when he fails to update the upper clone. We see that it is only almost a year later (5/11/05) that the inconsistency is removed by a *Fix Line* activity.

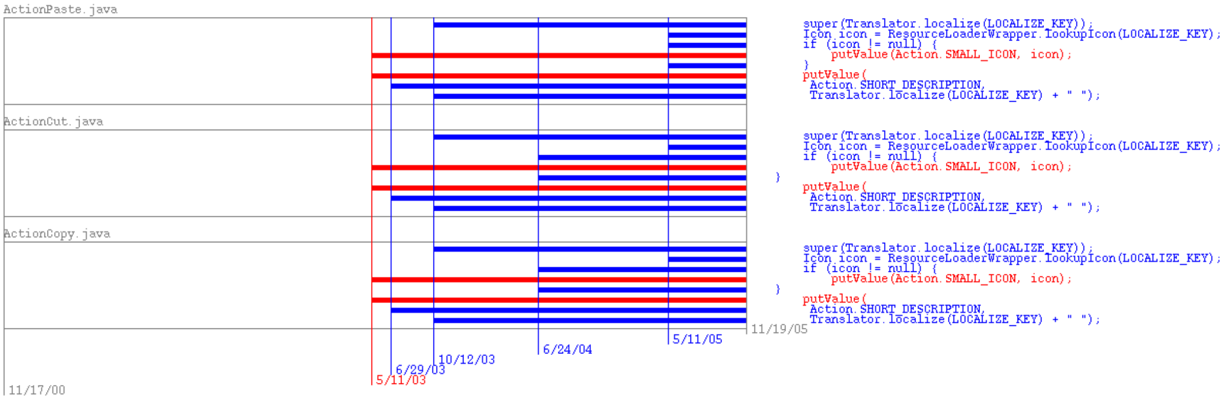
### 4.5 Case V: Inconsistent Line Cloning Fixed by Different Authors

In Figure 9 we encounter several developers, where at least one developer makes an inconsistent line cloning, and another developer intervenes at a later time to fix the inconsistency. In our example, PINK creates two clones but without lines 2, 3 and 4 (or at least the lines were not identical), and at a later time BLUE intervenes and fixes the 3 lines in both clones.

We found very few cases of this pattern. This is an indication that authors are rarely able to fix the cloning inconsistencies of others.

## 5 Guiding the Detection of Copy Patterns

The *Clone Evolution View* is useful for providing insights into one multiplication. However, as Table 1 shows, in a software system we might find hundreds of such multiplications, and not all of them are interesting from a reverse engineering point of view. As such, we need an automated

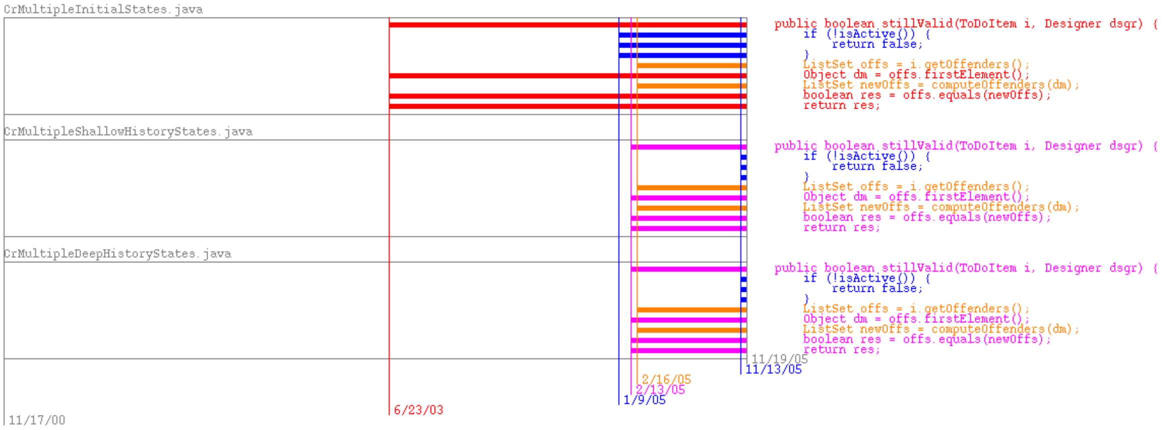


```
    super(Translator.localize(LOCALIZE_KEY));
    Icon icon = ResourceLoaderWrapper.lookupIcon(LOCALIZE_KEY);
    if (icon != null) {
        putValue(Action.SMALL_ICON, icon);
    }
    putValue(
     Action.SHORT_DESCRIPTION,
     Translator.localize(LOCALIZE_KEY) + " ");

    super(Translator.localize(LOCALIZE_KEY));
    Icon icon = ResourceLoaderWrapper.lookupIcon(LOCALIZE_KEY);
    if (icon != null) {
        putValue(Action.SMALL_ICON, icon);
}
    putValue(
     Action.SHORT_DESCRIPTION,
     Translator.localize(LOCALIZE_KEY) + " ");

    super(Translator.localize(LOCALIZE_KEY));
    Icon icon = ResourceLoaderWrapper.lookupIcon(LOCALIZE_KEY);
    if (icon != null) {
        putValue(Action.SMALL_ICON, icon);
}
    putValue(
     Action.SHORT_DESCRIPTION,
     Translator.localize(LOCALIZE_KEY) + " ");
```

**Figure 8. Case IV: Inconsistent Line Cloning Fixed by the Same Author (ArgoUML)**



```
public boolean stillValid(ToDoItem i, Designer dsgr) {
    if (!isActive()) {
        return false;
    }
    ListSet offs = i.getOffenders();
    Object dm = offs.firstElement();
    ListSet newOffs = computeOffenders(dm);
    boolean res = offs.equals(newOffs);
    return res;

public boolean stillValid(ToDoItem i, Designer dsgr) {
    if (!isActive()) {
        return false;
    }
    ListSet offs = i.getOffenders();
    Object dm = offs.firstElement();
    ListSet newOffs = computeOffenders(dm);
    boolean res = offs.equals(newOffs);
    return res;

public boolean stillValid(ToDoItem i, Designer dsgr) {
    if (!isActive()) {
        return false;
    }
    ListSet offs = i.getOffenders();
    Object dm = offs.firstElement();
    ListSet newOffs = computeOffenders(dm);
    boolean res = offs.equals(newOffs);
    return res;
```

**Figure 9. Case V: Inconsistent Line Cloning Fixed by Different Authors (ArgoUML)**

tool support to guide the reverse engineer through the maze of multiplications.

In this section we distill the experience we gained while performing the experiments. In the process, (see Figure 2) we defined several metrics, and found the following to reveal interesting insights:

- *Number of developers involved.* We use this measurement to detect multiplications which are done by one or more developers.

- *Number of clones.* We noticed that the most interesting cases are in multiplications with more than 2 clones.

- *Number of line fixes.* We use this to identify the cases of inconsistent cloning.

- *Knowledge extent over developers.* This measurement returns the number of lines cloned by more than one developer. We use this measurement to discriminate the cases in which several authors participate in an inconsistent cloning.

- *Number of modifications.* We use this measurement as activity indicator. A variation is to count the consistent modifications versus the inconsistent ones.

## 6 Discussion

**On the usage of cvs annotate.** Our current approach relies on the usage of cvs annotate, instead of analyzing the entire evolution. In this way, we are actually analyzing only the clones present in the analyzed version, and not their history. However, the initial results reported here are promising. In the future we plan to investigate in more details how developers copy by taking the clone genealogy approach of Kim *et al.* [13].

**On the visualization decisions.** The time is represented in the visualization on a linear scale. However, if two modification lines are too close from one another, it is difficult to distinguish the different moments. That is the reason why we imposed a minimum distance between two modification lines. An example of the effectiveness of this approach can be seen in Figure 5 (the last four modifications).

Another variable of the visualization was the choice of colors. In our experiments, the multiplications were developed by at most 5 developers. The limited amount of required colors allowed us to explore several color schemes trying to enhance the contrast both between the lines and against the white background, while also avoiding to combine red with green.

As we wanted to focus on how developers copy, we chose to represent the duplicated text detached from its context. In this way the attention is entirely dedicated to the cloning actions. However, after understanding the cloning phenomenon, we need to know if the clones are in the same context or not. That is why we chose to put the name of the context in a non disturbing color on the left side of the visualization.

To ease the interpretation of the visualization, we also arranged the clones in a meaningful way, trying to put the older clones on top of the newer ones. This follows our culture of reading from top to bottom [17], while it also provides more information. The colors inside the clone are also ordered by the the date of the first apparition in the multiplication. The order is: RED, BLUE, PINK and ORANGE.

**On the interactive nature of the implementation.** We implemented the approach in our tool called SmallDude on top of the Moose reengineering environment [16]. The implementation consists in extending the meta-model with the notion of multiplication, in building the visualization and in defining the measurements. Due to the capabilities of Moose we can easily query the multiplication space. This allowed us to explore different detection variations.

## 7 Related Work

There have been several implementations of text based approaches to detect clones. These approaches use either individual lines [1, 23, 6] or tokens as units for comparison [12]. Another way of finding duplications is by comparing the abstract syntax trees of methods (or parts of trees), this was implemented in a tool called CloneDR by Baxter *et al.* [2]. Mayrand *et al.* developed a tool for detecting duplication based on source code measurements [15]. They define a fingerprinting function which they then use to compare different entities from the source (*e.g.,* classes, methods).

Another related area is the use of visualization to display duplicated code. The most common way to visualize code clones is with a scatterplot [10, 12, 6]. The main drawback of this approach is that we can only see the duplications and not multiplications.

Rieger *et al.* aimed to visualize duplications at the level of the system using polymetric views to show the duplications as a relations between the parts involved [18]. For example, the showed classes as boxes and duplication as edges between them. However, they did not provide evidence of how multiplications are spread over the system. The difference with our visualization is that we proposed the visualization of a multiplication, and we related it to the the location in the source and with the author and the date that changed it.

The changes in clones have first been studied by Laguë *et al.* [14]. They use a metric based detection to identify the clones and then studied the number of clones added or removed. A much improved approach to study the evolution of clones was introduced by Kim *et al.* with their notion of clone genealogy [13]. They analyzed in detail several versions of the code, aiming to detect how clones get refactored. As opposed to the above ones, we aimed to analyze the clones in relation with the developers.

The analysis of developers actions has also attracted research over time. Girba *et al.* proposed an approach to analyze CVS repositories to detect how developers drive software evolution [9]. They define the owner of a file as a the developer that wrote the most lines of code and they develop a visualization to analyze how the authors have changed the different parts of the system.

Čubranić and Murphy bridged information from several sources to form what they call a "group memory" [20]. The sources used were: source code, versioning information, mailing lists, documentation.

Eick *et al.* proposed multiple visualizations to show changes using colors to denote the developer and width to map the module size [7]. They also discuss different types of views (matrix, 3D, bar and pie charts, etc.) about their usability and use this views to show change metrics. These change metrics include change count by developer, severity and status of the change.

Voinea et al. implemented a tool called CVSscan to analyze files from CVS repositories by showing all versions of the text lines [21]. They use colors to encode attributes like "author", "construct", and "line status" on each of these line versions.

The novelty of our approach as compared to the above ones is that we focused on providing information on how the developers copy.

## 8 Conclusions

The detection of clones have long been studied, and recently, their evolution has also attracted attention. However, to our knowledge, no attempt has been made to relate the developer information into the study of clones.

In this paper, we propose the usage of cvs annotate as a simple way of relating the developer and evolution information with the duplication information. Based on the collected data, we designed the *Clone Evolution View* to get insights into how developers copy.

We analyzed the clones of three large case studies and we identified several recurring patterns. We first browsed the systems to assess if interesting cases appear. The detection was made by looking just at the visualization to see striking cases from a reverse engineering point of view. In a second step we started to classify our findings and develop a vocabulary to describe the detected cases and then we checked (also assisted by automated filtering) the recurrence of the cases. In a third step, we tried to automate the detection of such interesting cases by developing measurements. As a result we provided a number of recommendations.

In this way, we detected that inconsistent changes are detected when more developers are involved in the multiplication. This enforces our original perception that the developers are relevant variables when analyzing code cloning. The experiments presented in this paper were more explorative, yet we believe the results are promising. In the future, we plan to undertake a more rigorous empirical study to classify a larger base of case studies.

**Acknowledgments.** Gîrba gratefully acknowledge the financial support of the Swiss National Science Foundation for the project Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1).

## References

[1] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

[2] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 368–377. IEEE Computer Society, Washington, DC, USA, 1998.

[3] M. E. Conway. How do committees invent ? *Datamation*, 14(4):28–31, Apr. 1968.

[4] J. R. Cordy. Comprehending reality—practical barriers to industrial adoption of software maintenance automation. In *Proc. 11th Int. Workshop on Program Comprehension (IWPC'03)*, pages 196–205, Portland, Oregon, USA, May 2003. IEEE.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[6] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings of theInternational Conference on Software Maintenance (ICSM '99)*, pages 109–118. IEEE Computer Society, Sept. 1999.

[7] S. Eick, T. Graves, A. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, 2002.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[9] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE)*, pages 113–122. IEEE Computer Society Press, 2005.

[10] J. I. Helfman. Dotplot patterns: a literal look at pattern languages. *TAPOS*, 2(1):31–41, 1995.

[11] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintence (ICSM 94)*, pages 120–126, 1994.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[13] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proceedings of European Software Engineering Conference (ESEC/FSE 2005)*, pages 187–196, New York NY, 2005. ACM Press.

[14] B. Laguë, D. Proulx, E. M. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of ICSM (International Conference on Software Maintenance)*. IEEE, 1997.

[15] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance (ICSM)*, pages 244–253, 1996.

[16] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[17] D. A. Norman. *The Design of Everyday Things*. The MIT Press, 1988.

[18] M. Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, University of Berne, June 2005.

[19] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *Proceedings of WCRE 2004 (11th Working Conference on Reverse Engineering)*, pages 100–109. IEEE Computer Society Press, Nov. 2004.

[20] D. Čubranić and G. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, New York NY, 2003. ACM Press.

[21] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: visualization of code evolution. In *Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005)*, pages 47–56, St. Louis, Missouri, USA, May 2005.

[22] R. Wettel. Automated detection of code duplication clusters. Master's thesis, Faculty of Automatics and Computer Science, "Politehnica" University of Timişoara, June 2004.

[23] R. Wettel and R. Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005)*, pages ??–??, 2005.