RC23217 (W0405-125) May 21, 2004
Computer Science

IBM Research Report

An Ethnographic Study of Copy and Paste Programming Practices in OOPL

Miryung Kim[1], Lawrence Bergman[2], Tessa Lau[2], David Notkin[1]

[1]Department of Computer Science and Engineering
University of Washington

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

An Ethnographic Study of Copy and Paste Programming Practices in OOPL

Miryung Kim[1] Lawrence Bergman[2] Tessa Lau[2] David Notkin[1]

Department of Computer Science & Engineering
University of Washington[1]
{miryung, notkin}@cs.washington.edu

IBM T. J. Watson Research Center[2]
{bergmanl, tessalau}@us.ibm.com

Abstract

Although programmers frequently copy and paste code when they develop software, implications of common copy and paste (C&P) usage patterns have not been studied previously. We have conducted an ethnographic study in order to understand programmers' C&P programming practices and discover opportunities to assist common C&P usage patterns. We observed programmers using an instrumented Eclipse IDE and then analyzed why and how they use C&P operations. Based on our analysis, we constructed a taxonomy of C&P usage patterns.

This paper presents our taxonomy of C&P usage patterns and discusses our insights with examples drawn from our observations. From our insights, we propose a set of tools that both can reduce software maintenance problems incurred by C&P and can better support the intents of commonly used C&P scenarios.

1. Introduction

Programmers often copy and paste code from various locations: documentation, someone else's code, or their own code. However, the use of copy and paste (C&P) as a programming practice has bad connotations because this practice has the potential to create unnecessary duplicates in a code base. Researchers have recommended that programmers should avoid creating code duplicates [7][9] – which are often created by C&P – because such duplicates can be difficult to maintain. For example, a bug can be propagated to scattered places when the code is copied. The software engineering community has made a significant effort to tackle the problem of code duplication. A number of clone detection tools have been developed to help programmers automatically locate code duplicates and refactor existing duplications to a unit of programming language abstraction [1][2][3][6][12][14][15]. However, in practice, a substantial amount of duplicated code is still present in many software systems [6][12]. Our understanding of how and why code clones are created is very limited.

Earlier studies have formed a few informal hypotheses about how C&P is performed by programmers to reuse code [17][18]. However, existing work has not focused specifically on solving the possible problems that can be incurred by C&P during software evolution.

The main purpose of our work is to investigate common C&P usage patterns and associated implications as a first step toward understanding and solving such problems. We believe that understanding when and how C&P is used will also reveal limitations in programming language designs and the lack of software engineering tool support to cope with common usage patterns.

In our investigation we have conducted an ethnographic study by observing programmers' C&P behavior. We developed a logger that records editing operations, enabling us to observe programmers in a non-intrusive manner. In addition, we built a replayer that can play back the editing logs captured by the logger. Then we analyzed how and why C&P operations were used and created a taxonomy of C&P usage patterns based on our analysis.

We have identified a number of interesting findings about common C&P patterns. Not only does C&P save typing, it also captures important design decisions made by programmers. Dependencies created by C&P are useful for program understanding. In fact, programmers employ their memory of C&P history as they make changes to code or decide when to restructure code. However, a programmer's recollection of C&P history can be short-lived,

somewhat inaccurate, and difficult to transfer from person to person. The lack of tool support for recording and using C&P editing information may cause problems during software evolution. Specifically we have made the following observations:

- Limitations of programming language designs may result in unavoidable duplicates in a code base.
- Programmers often delay code restructuring until they have copied and pasted several times.
- C&P dependencies often reflect important underlying design decisions, such as aspects or crosscutting concerns.
- Copied text is often reused as a template and is customized in the pasted context. Current software engineering tools have poor support for identifying reusable code templates or maintaining them during software evolution.

Based on our insights about C&P usage patterns, we propose tools to reduce software maintenance problems caused by C&P and that would allow programmers' intent to be expressed in a safe and efficient manner.

The rest of the paper is organized as follows. Section 2 presents the ethnographic study that we conducted. Sections 3-5 describe our taxonomy of C&P patterns that we observed from three different perspectives. Section 6 summarizes our insights. In Section 7, we propose tools based on our insights. In Section 8, we discuss possible threats to the validity of our study results and share our conjecture about C&P patterns in different study settings. Section 9 discusses related work and Section 10 summarizes our contributions.

2. Ethnographic Study

We conducted a study to observe programmers performing coding tasks either by watching them directly or by having them use an instrumented editor that logs their editing operations. In the latter case, we conducted follow-up interviews to understand programmers' tasks at a high level and to confirm our interpretation of their actions. Using the collected data, we built a taxonomy of C&P operations.

In Section 2.1 we present the two different methods of observations and discuss their pros and cons. In Section 2.2, we describe the functionality of the logger and the replayer that we developed. Section 2.3 describes our method for analyzing C&P operations.

Section 2.4 presents some statistics about C&P behaviors that we observed.

2.1. Observation

Our study involves two types of observations. First, we watched programmers over the shoulder as they write programs, and we took notes during the observation. In general, it was extremely difficult to manually log editing operations performed by the subjects; we could not identify the exact code fragments that were copied and pasted. Therefore, we interrupted the subjects' programming flow and asked them to explain what and why they were copying and pasting. Because the subjects were aware that we were analyzing the intention of each C&P operation, they did not copy and paste unless they thought they had a valid reason. Our presence in the room also seemed to put pressure on the subjects to write code continuously, which was not natural for them. One advantage of direct observation, however, was that it was easier for us to identify the intention of copying and pasting because most participants voluntarily and clearly explained their intentions.

In order to enable subjects to write programs in a more natural setting and to log editing operations with greater precision, we used a logger and a replayer (described in the next section). Using the logger, we recorded coding sessions and then observed the participants' actions off-line by replaying the captured editing operations.

For both types of observation, the subjects were researchers at IBM T. J. Watson Research Center. They were expert programmers in Java and were involved in small team research projects. In total, nine subjects participated in our study, and we observed about 60 hours of coding in object-oriented programming languages, mainly in Java. Observational study settings are summarized in Table 1.

Table 1. Observational study setting

	Direct Observation	**Observation using a logger and a replayer.**
Subjects	Researchers at IBM T. J. Watson	
Number of Subjects	4	5
Total Coding Hours	About 10 hrs	About 50 hrs.
Interviews	Questions asked during observation	Twice after analysis (30 mins ~1 hour/each)
Programming Languages	Java, C++, and Jython	Java

2.2. Logger and Replayer

The logger efficiently records the minimal information required to reconstruct document changes performed by a programmer. We developed the logger by extending the text editor of the Eclipse IDE [5] and instrumenting text editing operations. It records the initial contents of all documents opened in the workbench and logs changes in the documents. It records the type of editing operations, the file names of edited documents, the range of selected text, and the length and offset of text entries, as well as editing operations such as copy, cut, paste, delete, undo, and redo. It also captures document changes triggered by other automated operations such as refactoring and organizing import statements.

The replayer plays back the changes made to the document using the low level editing events captured by the logger. It displays documents and highlights document changes and selected text. It has a few controls such as play, stop, and jump. Whereas videotape analysis of coding behavior normally takes 10 times as long as the actual coding,[1] we only spent 0.5 to 1 times as long as the actual coding to analyze the data by using the instrumented text editor and the replayer.

2.3. Analysis

By replaying the editing logs, we documented individual instances of C&P operations. An instance consists of one copy (or cut) operation followed by one or more paste operations of the copied (or cut) text. It also includes modifications performed on the original text or the cloned text. We categorized each instance with a focus on the procedural steps and the syntactic units of copied (or cut) content, such as types, identifiers, blocks, and methods.

Since we observed multiple C&P instances that share similar editing steps, we generalized the editing procedures to identify C&P usage patterns. For example, one frequent C&P pattern was to change the name of a variable repeatedly. The renaming procedure consists of selecting a variable, copying the variable, optionally searching for the variable n times, and pasting the variable n times (where n is the number of appearances of the variable within its scope).

For each generalized C&P procedure, we inferred the associated programmer's intention. Inferring a programmer's intention was often straightforward. For example, "changing the name of a variable consistently" is the intention associated with the renaming pattern described earlier.

For each C&P instance, we also noted the relationship between a copied code snippet and code elsewhere in the code base. In addition, we analyzed the evolutionary aspect of C&P instances by observing how duplicated code fragments were maintained and changed during our study.

After producing detailed notes for each C&P instance, we met with subjects to confirm our interpretation of their C&P tasks. Then we built a taxonomy of C&P operations by grouping related C&P instances together and hypothesizing C&P usage patterns from the grouped notes using an affinity process [4]. In total, 460 C&P instances were analyzed.

2.4. Statistics

In this section, we present simple statistics about C&P usage patterns that we observed. With the instrumented editor, we observed 460 C&P instances. We measured the frequency of C&P instances for each observation session (i.e. the number of C&P instances per hour). The average number of C&P instances per hour is 16 instances per hour and the median is 12 instances per hour.

In order to understand how often C&P operations of different size occurred, we grouped C&P instances into four different syntactical units and counted them (Figure 1). About 74% of C&P instances fall into the category of copying text less than a single line such as a variable name, a type name or a method name. In these cases, we believe that copying was performed to save typing. However, about 25% of C&P instances involved copying and pasting a block or a method. We believe that copying in this category often creates structural clones and reflects design decisions in a program. When we multiply this percentage (25%) by the average 16 instances per hour, it means that a programmer produces four non-trivial C&P dependencies per hour on average.

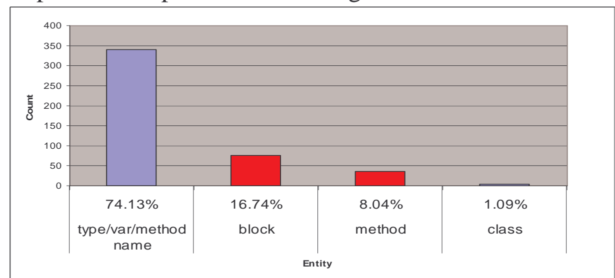

Figure 1. Distribution of C&P instances by different syntactic units

[1] Personal communication with J. Karat, a user study expert.

The following three sections present the resulting taxonomy, focusing respectively on the intentional, design, and evolutionary perspectives on C&P operations. Section 3 (Intention view) describes the categorization of programmers' intentions involved in C&P operations. Section 4 (Design view) describes the categorization of design decisions that induce programmers to copy and paste in particular patterns. In Section 5 (Maintenance view) we discuss maintenance tasks associated with C&P operations.

3. Intention View

We constructed the categorization of programmers' intentions by inferring intentions associated with common C&P patterns and by directly asking questions of the subjects.

One use of C&P is to **relocate, regroup, or reorganize** code from one place to another according to the programmers' mental model of the program's structure. Programmers also use C&P to **reorder** code fragments. For example, a Boolean expression (A||B||C) could be reordered as the equivalent expression (B||C||A) to improve performance, or several if-blocks could be reordered so that negated if-statements return earlier. Programmers also use C&P to **restructure** (or **refactor**) their code manually.

The most common C&P intention in our study was to use a copied code snippet as a **structural template** for another code snippet. Programmers often copied the entire code snippet and removed code that was irrelevant to the pasted context. The structural templates can be either reusable syntactic elements of code snippets (syntactic templates) or reusable programming logic (semantic templates).

Figure 2 shows an example of a syntactic template. The statement `protectedClasses.add("java.lang.Object")` was copied multiple times. The duplicates were modified after they were pasted. We deduced that the programmer intended to reuse `protectedXXX.add("java.lang.YYY")` as a template for other statements in the static method initialization. We conjecture that the lack of functionality in today's IDE and (/or) limitations in language constructs increase the need for copying syntactic templates. For example, the absence of repetitive text editing support in an IDE or the lack of the `"enum"` construct in Java causes programmers to copy and paste a particular phrase frequently.

```
static{
  protectedClasses.add("java.lang.Object");
  protectedClasses.add("java.lang.ref.Reference$ReferenceHandler");
  protectedClasses.add("java.lang.ref.Reference");
  protectedMethods.add("java.lang.Thread.getThreadGroup");
}
```

Figure 2. Example of a syntactic template

In this paper, copied text is represented as copied text (with dotted underline), pasted text is represented as *pasted text (italic)*, deleted text is represented as ~~deleted text (with double strikethrough),~~ and cut text is represented as ~~cut text (with single strikethrough)~~. Modifications performed on top of pasted text are represented as modified text (with solid underline).

The other category of structural template reuse is semantic templates. The following four paragraphs categorize the use of semantic templates.

Design Pattern

In our study, one programmer told us explicitly that what he copied was the instantiation of the Strategy pattern [8]. We suspect that the programmer used a concrete instantiation of the Strategy pattern as a template because it was easier than writing code from an abstract description of that design pattern.

Usage of a Module (Class)

Programmers often copy a code snippet to reuse the usage protocol of a target module [19]. We observed many cases where a code snippet was copied because it contained logic for accessing a frequently used data structure. In Java, programmers are required to know the usage protocol for library data structures that they intend to use. For example, in order to traverse keys in a `Hashtable`, a programmer needs to get a reference for a key set by invoking the `keySet`() method on the hashtable object and then obtain an iterator for the key set. We observed a number of similar cases in our study. One example is shown in Figure 3. The code snippet was copied because it contains frequently used code for traversing over `Element` nodes in a DOM Document in C++.

Implementation of a Module

Programmers may copy a code snippet because it contains a definition of particular behavior that they want to reuse. For example, a programmer copies the signature and partial implementation of a module when they intend to reuse part of the module's behavior. Although inheriting abstract classes or interfaces can be an alternative for this case, in our study, programmers sometimes did not choose this alternative.

```
DOMNodeList *children = doc->getChildNodes();
int numChildren = children->getLength();

for (int i=0; i<numChildren; ++i)
  {
    DOMNode *child = (children->item(i));
    if (child->getNodeType() ==
      DOMNode.ELEMENT_NODE)
    {
      DOMElement *element = (DOMElement*)child;
```

Figure 3. Semantic template: traversing over element nodes in a DOM document in C++

Control Structure

Programmers frequently reuse complicated control structures, such as a nested `if then else` or a loop construct. When programmers intend to write code that has the same control structure but different operations inside the control structure, they tend to copy the code with the outer control structure and modify its inner logic. For example, in Figure 4 a programmer copied a loop construct and modified the inner logic after pasting.

```
For (Iterator it=messages.iterator();it.hasNext();) {
    Message curr= (Message) it.next();
    IFile markFile=
    WorkspaceUtils.getFile(curr.getFirstLocation().getCla
    ssName());
.........}
for (Iterator it=messages.iterator();it.hasNext();) {
    Message curr= (Message) it.next();
    MessageLocation loc = curr.getLastLocation();
    IFile markFile=
    WorkspaceUtils.getFile(loc.getFirstLocation().getCla
    ssName());
........}
```

Figure 4. Semantic template: Copying a loop construct and modifying the inner logic

4. Design View

The Aspect Oriented Programming community has observed that primary design decisions that are already embedded in a system sometimes do not allow secondary design decisions to be modularized in a small module when they are added to the system [13][19]. We believe that the lack of modularity leads programmers to insert similar code snippets across a code base, which is often done through copying and pasting.

We examined underlying design decisions that induce programmers to copy and paste in particular patterns. Unlike the intention view, where we analyzed code snippets involved in each C&P instance in isolation, in the design view, we analyzed the code snippets in relation to other code snippets in the system. We raised several questions to understand the architectural (or design) context of C&P operations. Each of the following three sub-sections discusses why we chose each question and describes the categorization of answers to the question.

4.1. Why is text copied and pasted repeatedly in multiple places?

We observed that particular code snippets may be copied and pasted repeatedly in scattered places. We raised this question to understand why programmers chose to duplicate a code snippet rather than to refactor it.

Our answer to this question is that some concerns are difficult to separate from the execution context because these concerns require accessing the execution context [19]. For example, the code of a logging concern in Figure 5 was copied and pasted four times within one file and many more times across the code base. Because it is difficult to generalize the list of arguments for the factored logging function, refactoring this code snippet is often less preferable than copying the code snippet. In addition, even if the programmer chooses to refactor it, the dependencies between the logging module and the other modules would remain entangled.

```
if (logAllOperations) {
        try {
                PrintWriter w = getOutput();
                w.write("$$$$$");
..
        } catch (IOException e) {
        }
```

Figure 5. Duplicated code: logging concern

For the same reason, adding a software feature sometimes requires making changes in scattered places across a code base. In one project that we observed, a programmer added a feature to display a user-friendly type for internal objects instead of the internally used XML type for the objects in his code. First, he wrote the body of `getFriendlyTypeName()` and duplicated it in four different classes. When he realized that it was better to refactor the code as a separate method, he copied the body of `getFriendlyTypeName()` and pasted it into the `MiscOps` class. Then he copied and

pasted the invocation statement of `MiscOps.getFriendlyTypeName()` four times to call the refactored method.

We suspect that when the secondary design decision such as the logging concern needs to compromise its modularity with the primary design decision, programmers are required to duplicate some code because the dependencies between primary modules and secondary modules would still exist.

4.2. Why are blocks of text copied together?

We observed that when a code snippet is copied from A and pasted to B, related code snippets are also copied from A and pasted to B. We believe that code snippets are often copied together because they belong to the same functionality or concern. Some examples of code snippets that are copied together are described as follows:

Comments

A comment is copied when its related code is copied.

Referenced Fields/Constants

Programmers copy referenced fields and constants when they copy a method that refers to them.

Caller Method and Callee Method

Programmers copy a referenced method when they copy a method or a class that invokes the method. Similarly, a caller method is copied when its called method (callee) is copied. In one case that we observed, a programmer copied the contents of the `sender.cpp` file to `heartbeat.cpp` in order to create a `heartbeat` thread that has similar behavior to the `sender` thread. After he finished modifying `heartbeat.cpp`, he copied the invocation statement of `start_sender()` and pasted it as the invocation statement of `start_heartbeat()` in the test driver file. He also copied the invocation of `shutdown_sender()` and pasted it as the invocation of `shutdown_heartbeat()`.

Paired Operations

Programmers copy and paste paired operations together. For example, when a programmer copies `writeToFile()`, he also copies `openFile()` and `closeFile()`. Likewise, when `enterCriticalSection()` is copied, `leaveCriticalSection()` is copied as well.

4.3. What is the relationship between copied and pasted text?

We raise this question to understand why programmers choose a code fragment as a template. In other words, we are interested in understanding the relationship of the copied text and the pasted text.

```
public void updateFrom  (Class c ) {
    String cType =
    Util.makeType(c.getName());
    if (seenClasses.contains(cType)) {
        return;
    }
    seenClasses.add(cType);
    if (hierarchy != null) {
        addToHierarchyViaReflection(c);
    }
    if (methods != null) {
        Method[] ms = c.getDeclaredMethods();
        for (int i = 0; i < ms.length; i++) {
            Method m = ms[i];
            methods.addMethod(cType,
            m.getName(),
            Util.computeSignature
            (m.getParameterTypes(),
            m.getReturnType()),
            m.getModifiers());
        }
    }
}
```

Figure 6. Populating the same data structure: updateFrom(Class c)

```
public void updateFrom  (ClassReader cr )  {
    String cType =
    CTDecoder.convertClassToType
    (c.getName());
    if (seenClasses.contains(cType)) {
        return;
    }
    seenClasses.add(cType);
    if (hierarchy != null) {
        CTUtils.addClassToHierarchy(hierarchy,
        cr);
    }
    if (methods != null) {
    int count = cr.getMethodCount();
    for (int I = 0; i < count; i++) {
            Method m = ms[i];
            methods.addMethod(cType,
            cr.getMethodName(i),
            cr.getMethodType(i),
            cr.getMethodAccessFlags(i));
    }
}
```

Figure 7. Populating the same data structure: updateFrom (ClassReader cr)

Similar Operations but Different Data Sources

This category is a special case of semantic templates where the duplicated code snippets manipulate different data sources. In one application that we observed, error messages were sent from one stage to the next stage by calling method A. The same error messages are also sent to a user by invoking method B. A is copied and used as a template for B because A and B contain logic for reading the same header, only differing in the targets to which they direct error messages.

As another example, in Figures 6 and 7, the `updateFrom (Class c)` method is used as a template for the `updateFrom (ClassReader cr)`. Both methods contain logic for populating the same data structure. While one method reads from a class object that is obtained through Java reflection, the other reads from Java byte code.

Semantically Parallel Concerns

We define semantically parallel concerns as design decisions that crosscut a system in a similar way. For example, we say that supporting an integer operation and supporting a floating point operation in a compiler are semantically parallel concerns because they crosscut each component of a compiler's pipeline architecture in a similar way. We observed one project that involves extending a compiler to support XML DOM objects. At the time of the observation, the compiler already had code related to the `serialize` concern and the subject wanted to insert code related to the `appendChildren` concern. The programmer identified all the code related to the `serialize` concern by exploiting the fact that information transparent modules [10] are often encoded with the same signature, such as the use of particular variables, data structures, or language features. The programmer then copied the identified code snippets and modified them as necessary for the `appendChildren` concern. When we asked the programmer why he programmed in such way, he answered that those concerns crosscut the same places in the compiler architecture and it helped him to keep track of which part of the system to extend. A similar case was observed in [10] when the C-Star was retargeted to Ada. The pipeline architecture of C-Star guided the programmer to identify all the code related to C syntax specific support and convert it to Ada syntax specific support.

Paired Operations

In Section 4.2, we mentioned that paired operations are copied together frequently. But in this section we discuss paired operations as a special case of sharing the usage of the same data structure (discussed in Section 3).

```
public void addMethod () {                      WRITE
    // retrieve a map
    if (map == null) {
        // create a map
    }
    // get an entry o
    if (o == null) {
        // add that method into a map and return
    }
    if (o instanceof ArrayList) {
        // cast
    } else {
        // create an array list and add it to a map
    }
    // add a method to the array list
}

public MethodInfo getClassMethod() {            READ
    // retrieve map
    if (map == null) {
        // return null
    }
    // get an entry o
    if (o == null) {
        // return null
    }
    if (o instanceof ArrayList) {
        // traverse each method "m" in the array list,
        and if matches, return "m"
    } else {
        // if signature matches, return that method
}
```

Figure 8. Paired operations: write / read logic

For example, in Figure 8 the `addMethod()` method is copied and used as a template for the `getClassMethod()` method because the `addMethod()` and the `getClassMethod()` access a hashmap where each value of (key,value) pairs can be either a single object or an array list of multiple objects. `getClassMethod()` contains read logic that pairs with write logic in `addMethod()`.

Inheritance

We observed several cases where a programmer copied a superclass and used as a template for subclasses and copied a sibling class as a template for other sibling classes.

To summarize, in the design view, we examined various kinds of C&P dependencies, such as the relationship between a copied code snippet and a pasted code snippet, the relationship of code snippets that are copied together, and the relationship of code snippets that are duplicated repeatedly. Based on our analysis, we conclude that observing and maintaining

C&P dependencies is worthwhile because these dependencies reflect important design decisions, such as crosscutting concerns, feature extensions, paired operations, semantically parallel concerns, and type dependencies (inheritance).

5. Maintenance View

We investigated maintenance tasks for duplicated code because failing to perform such tasks may lead to software defects. Although this ethnographic study was not a longitudinal study, we addressed maintenance problems associated with C&P by examining what programmers did immediately after a C&P operation and how programmers modified code duplicates created by C&P.

Short Term Maintenance Tasks

We noticed that cautious programmers modify the portion of pasted code that is specific to the intended use immediately after they copy and paste. For example, they modify the name of a variable to prevent identifier naming conflicts or remove the portion of the pasted code that is not part of the structural template.

Long Term Maintenance Tasks

Programmers restructure (or refactor) their code after frequent C&P of large texts. For example, after one code snippet is copied and pasted multiple times, the code snippet may be refactored as a separate method. Another example is that after frequently defining an anonymous class and instantiating objects of the class on the fly, a programmer might define an inner class and create a member variable that holds the object.

By observing how programmers handle code duplicates created by C&P, we noted that programmers tend to apply consistent changes to code from the same origin. In other words, after they create structural clones, they modify the structural template embedded in the clones consistently when the template evolves. This observation is symmetric to the information transparency principle [10] that code elements that change together must look similar.

6. Insights

In this section, we summarize our insights about C&P.

- **Limitations of particular programming languages produce unavoidable duplicates in a code base.**

For example, the lack of multiple inheritance in Java makes it difficult to impose a particular behavior or an aspect without creating duplicates. As another example, the lack of the `"enum"` construct in Java makes programmers copy the phrase `"public static final String"` frequently.

In some cases programmers do not remove code duplicates even if it is possible to refactor them, because the organization of the code does not match the programmers' conceptual organization of the code.

- **Programmers use their memory of C&P history to determine when to restructure code.**

A few programmers told us that they deliberately delay code restructuring on purpose until they copy and paste several times because such reuse helps them discover the right level of abstraction. We suspect that larger or frequently copied code fragments are good candidates for refactoring.

- **C&P dependencies are worth observing and maintaining.**

The examples in Section 4 demonstrate that C&P dependencies reflect design decisions such as aspects, parallel cross-cutting concerns, paired operations and so on. We believe that C&P dependencies are worth preserving because they can supplement the static understanding of a code base that can be extracted from the code itself.

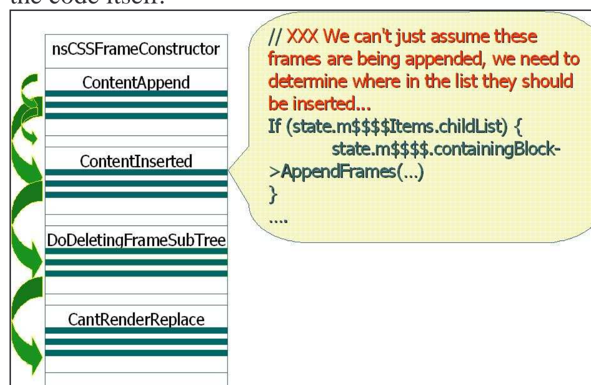

Figure 9. Example of a bug propagation: Mozilla bug id 217604

In addition, programmers rely on their memory of C&P dependencies when they apply consistent changes to duplicated code. If programmers forget where structural clones are located and what the template of a set of structural clones is, then they may produce defects when making changes to the software. We found a motivating example from the Mozilla open source project. One bug in Mozilla required a programmer to fix bugs that had been propagated to 12 different places by C&P. A bug was introduced to the code snippet in Figure 9 by invoking the `appendFrames()` method instead of the `insert Frames()` method. The code snippet was copied

twice within the same method and the method itself was copied three times. Ultimately, 12 structural clones containing that faulty code snippet were produced. The programmer who fixed the bug had to lexically search the code base for comments starting with "xxx" in order to apply the appropriate modifications consistently. If "xxx" had not existed in the copied comment, or if the signature of the structural template had evolved very differently in individual code fragments, then lexical search may not have been adequate to locate the faulty code snippets.

- **Programmers copy an entire code snippet because it contains the structural template that they intend to reuse.**

Thus we conclude that it is desirable to provide software development environments that learn structural code templates and support reuse of the learned templates. We also believe that identifying frequently used structural templates will provide input for better programming language design.

7. Proposed Tools

Based on our insights from our study results, we propose tools that can minimize the software maintenance problems that may be incurred by C&P, as well as support common C&P programming practices.

Visualization

We propose a tool that visualizes copied and pasted contents and explicitly maintains and represents the C&P dependencies. We believe that this tool can increase traceability of a code snippet when programmers intend to apply the same change to the duplicates of the code snippet. Programmers can also communicate the intention behind the duplication to other programmers by annotating the duplicated code snippets with their intention.

Extraction of Structural Templates

We propose a tool that learns the relevant structural template of a code snippet by observing repetitive duplication of the code snippet followed by modifications to it. When a programmer intends to duplicate this structural template via C&P, the tool can provide advanced sentence (or block) completion and assist in removing the code that is irrelevant to the pasted context.

Warning / Notification

From the examples of semantic templates in Section 3, we conjecture that structural templates may indicate protocols or agreements on the usage of a module. Using the two proceeding proposed tools, we could warn programmers when they attempt to change a structural template of code fragments. We could also notify other programmers or propagate changes to other uses of the structural template automatically when there is a change in the structural template. This tool could prevent inconsistent changes in a code base.

Refactoring recommendations

Although the Eclipse IDE provides a number of automatic restructuring mechanisms, Eclipse IDE does not suggest where to restructure or which refactoring mechanism to use. We believe that by identifying a structural template of copied code snippets and by monitoring the frequency and size of copied text, we could automatically suggest when and how to restructure the copied text.

8. Threats to Validity

The scope of our study is confined to C&P programming practices in object oriented programming languages (OOPL). Thus some results that involve OOPL-specific features may not apply to other programming languages. For example, programmers who use functional programming languages may not need to copy a code snippet that contains a complicated control structure because higher order functions in the language allow such control structure to be passed as parameters. However, we believe that OOPLs are widely used and our study results provide valuable insights for the design of software engineering tools for them.

Participants in our study were researchers at the IBM T. J. Watson Research Center. They were expert programmers in Java and were involved in small team research projects. Our results may not be applicable to larger projects or projects that involve programmers with different levels of expertise in programming. We conjecture that novice programmers may copy and paste to learn the syntax of programming languages or employ less of their knowledge about C&P history when they maintain software.

9. Related Work

Various types of clone detectors have been developed to cope with the problem of maintaining code duplicates during software evolution [1][2][3][6][12][14][15]. In addition, these clone detection tools have found a large proportion of cloned code in popular software systems. However, these research projects did not address how code clones are entered into a system or why programmers duplicate code.

In order to understand code reuse strategies in object oriented programming, Lange *et al.* [17]

conducted a week long observation of a single subject writing programs in Objective-C. The investigators observed that the subject often copied a super-class or a sibling-class as a template for a new class and then edited the copied class. Rosson *et al.* [18] observed four subjects programming in Smalltalk. In her study, she observed that when the subjects were interested in reusing a particular target class, they copied the usage protocol of the target class and used it as an example code snippet without deeply comprehending the behavior of the target class. Although they considered C&P as one strategy of source code reuse, they did not focus on the implications of C&P. In our study, we not only observed the same code reuse behavior, but also analyzed why programmers chose specific code snippets as templates.

Lagüe *et al.* [16] studied the evolution of code clones in six versions of a large telecommunications system. They found that the overall number of clones in the system grew even though a significant number of clones were removed from the system. They also found that only half of clones in each version were modified in the same way in the next version. Their argument supports the benefits of our proposed tools.

Data collection via logging has been used to observe programming practices in the software engineering community. For example, Thomas *et al.* collected students' programming data by capturing keystroke, mouse, and window focus events generated by the Windows OS [11]. In our study, we efficiently logged and analyzed large quantities of coding data using our logger and replayer. We envision that our tools will serve as a basis for an infrastructure that captures editing history and provides an API to software engineering applications that employ editing process information.

10. Conclusion

Common wisdom dictates that good programmers do not use C&P operations because they tend to produce maintenance problems. Our ethnographic study has shown that programmers nevertheless use C&P very frequently, producing up to four architecturally significant C&P instances per hour. Rather than viewing this as a drawback, we instead take this as an opportunity to identify and develop software engineering tool support for existing practices. Specifically we discovered that C&P editing information is useful for program understanding and that programmers actively make use of the history of C&P operations as they make changes to software or decide when to restructure code. We have identified software maintenance problems that may be induced by common C&P usage patterns and proposed a set of tools to solve such problems.

11. References

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and L. Kontogiannis, "Advanced Code Analysis to Support Object-Oriented System Refactoring", *WCRE,* 2002.

[2] B. S. Baker, "A Program for Identifying Duplicated Code", *Computer Science and Statistics*, 1992.

[3] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees", *ICSM*, 1998.

[4] Beyer, H. and K. Holtzblatt, *Contextual design: defining customer-centered systems,* Morgan Kaufmann Publishers, San Francisco, California, 1998.

[5] Eclipse IDE. http://www.eclipse.org

[6] S. Ducasse, M. Rieger, S. Demeyer, "A language Independent Approach for Detecting Duplicated Code", *ICSM*, 1999.

[7] Fowler, Martin *Refactoring: Improving the Design of Existing Code,* Addison Wesley, 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[9] Hunt, A. and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master,* Addison-Wesley, 2000.

[10] W. Griswold, "Coping with Software Change Using Information Transparency", *ICSE*, 1999.

[11] GRUMPS project, http://grumps.dcs.gla.ac.uk

[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 System*, 2002.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented programming", ECOOP, 1997.

[14] R. Komondoor, and S. Horwitz, "Using Slicing to Identify Duplication in Source Code", *ISSA*, 2001.

[15] J. Krinke, "Identifying Similar Code with Program Dependence Graphs", *WCRE,* 2001.

[16] B. Lagüe, D. Proulx, E. M. Merlo, J. Mayrand, and J. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process", *ICSM,* 1997.

[17] B. Lange and T. Moher, "Some Strategies of Reuse in an Object-Oriented Programming Environment", *CHI,* 1989.

[18] M. B. Rosson and J. M. Carroll, "Active Programming Strategies in Reuse", *ECOOP*, 1993.

[19] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr., "N degrees of Separation: Multidimensional separation of concerns", *ICSE,* 1999.