Ethnographic Study of Copy and Paste Programming Practices in OOPL

Miryung Kim

Abstract

When programmers develop and evolve software, they frequently copy and paste (C&P) code from an existing code base, or sources such as web pages or documentation. We believe that programmers follow a small number of well defined C&P usage patterns when they program, and understanding these patterns would enable us to design tools to improve the quality of software. We conducted an ethnographic study in order to understand programmers' copy and paste programming practices. We observed 5 subjects programming in Java using an instrumented *Eclipse* IDE for about 50 coding hours and conducted interviews with them. In addition, we also observed 4 programmers directly for about 10 hours while they programmed. Based on these observations, we have built a taxonomy of copy and paste usage patterns.
This paper presents our taxonomy of copy and paste usage patterns and discusses our insights with examples drawn from our observations. 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 copy and paste scenarios.

1 Introduction

Programmers often copy and paste code from documentation, someone else's code, or their own code. However, usage of copy and paste as a programming practice has a bad connotation, because this practice could create unnecessary duplicates in a code base. Some researchers have recommended that programmers should avoid creating code duplicates [Fowler00] - which are often created by C&P, because such duplicates can be difficult to maintain. For example, a bug introduced in copied code can be propagated to scattered places when the code is pasted. Earlier studies have formed a few informal hypotheses about how C&P is performed by programmers to reuse code [RC93, LM98]. However, existing work has not focused specifically on solving the possible problems that can be incurred by C&P during software evolution.

The purpose of this study is to investigate common C&P usage patterns and associated implications as a first step to understand and to solve such problems. In our investigation, we conducted an ethnographic study by observing programmers copy and paste code. We built a taxonomy of C&P usage patterns by analyzing C&P operations from multiple perspectives. In addition, we investigated the quantitative aspect of copy and paste operations by measuring how frequently C&P was used and how often C&P tasks of a certain type occurred.

We present a number of interesting findings about programmers' intentions involved in C&P and the relationships of copied code snippets to other code snippets in a program. Our study shows that copying and pasting not only saves typing but also reflects a programmer's mental model of design. However it creates textual clones and introduces dependencies that are significant in the structure of a program.

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

The rest of the paper is organized as follows. Section 2 discusses our approach towards identifying C&P usage patterns. Section 3 ~ 6 describe various aspects of our taxonomy of these patterns. We present simple statistics about C&P behaviors in Section 7. Section 8 summarizes our insights. In Section 9, we propose

tools based on our insights. In Section 10, we discuss possible threats to the validity of our study results. We conclude by summarizing our study.

2 Approach

Our study involves two types of observations. First, we observed participants by watching them program directly (Section 2.1). We then observed participants by replaying logs of editing operations captured by an Eclipse plug-in that we developed (Section 2.2). Each sub-section describes the user study formats, the analysis techniques that we used, and the tools we built.

2.1 Direct Observation

We observed programmers directly over the shoulder while they program. Because this type of observation is not limited by an instrumented editor that is designed for a particular programming language, we observed programming in Java, Jython, and C++. Each subject programmed for about 1-2 hours per session. In total, we observed 5 sessions involving 4 subjects.

In general, it was extremely difficult to manually log editing operations performed by the subjects: we could not identify the exact code snippets that were copied and pasted. Thus, we interrupted the subjects' programming flow and asked them to explain what they were copying and pasting and why. Because the subjects were aware that we were analyzing the intention of each copy and paste operation, they did not copy and paste unless they thought they had a valid reason. Most participants did not like having an observer next to them while they were coding, because they preferred to design, code, and test iteratively than to write code continuously.

One advantage of direct observation was that it was easier for us to identify the intention of copying and pasting, because most participants voluntarily explained their intentions clearly.

2.2 Observation with an Instrumented Eclipse IDE

In order to enable subjects to program in a more natural setting and to log editing operations with greater precision, we extended the text editor of a popular software development environment, *Eclipse*. By deploying the instrumented version of Eclipse IDE, we collected log files from participants. Because the Eclipse JDT is designed to support software development in Java, our observation was limited to Java. We observed the subjects offline by replaying the log files. After we analyzed the intention of each C&P task, we conducted interviews with the subject to confirm the result of our analysis. After the interviews, we built a taxonomy of C&P operations by constructing an affinity diagram [BH95].

Five subjects programmed for 50 hours. In total 17 log files were collected.

Eclipse Plug-In
The editor plug-in captures the initial contents of all documents opened in the workbench and logs changes in the documents. The editor generates a *typing* event when a programmer presses a regular key or backspace key, or when Eclipse IDE performs changes to a document triggered by other automated operations such as refactoring and organizing import statements. A typing event records the type of editing operation, the file name of the document, and the length and offset of text involved in the operation. A *Command* event is generated by editing operations, such as copy, cut, paste, delete, undo, and redo. A command event records a timestamp and selected text ranges in addition to the same information recorded in a typing event. All events are recorded in XML format in time order.
Replayer

The replayer that we built visualizes an XML tree of typing events and command events. It displays documents and highlights document changes and selected text. It has a few controls such as play, stop, and jump.

While a videotape analysis of coding behavior normally takes 10 times of the actual coding time [K03], we only spent 0.5 to 1 times of the actual coding time by using the instrumented text editor and building a replayer.

Analysis

Copy and paste operations are analyzed and documented in an *instance unit.* 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 other modifications performed on the original text or the cloned text.

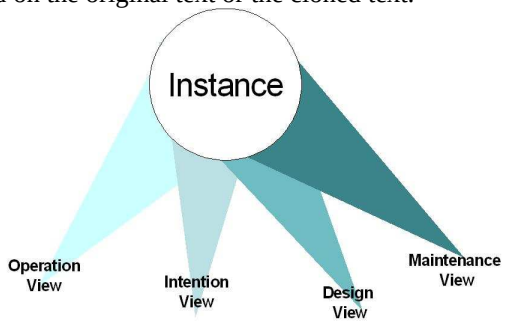

Figure 1 Analysis of an Instance from Multiple Perspectives

We analyzed each instance from multiple perspectives. Having multiple perspectives helped us avoid concentrating on a single aspect of copy and paste operations and constructing a taxonomy with more weight on that aspect. We chose four different views, which consist of an operation view, an intention view, a design view and a maintenance view. The operation view focuses on the procedural steps of editing operations (Section 3). The intention view focuses on the programmers' intention at a high level (Section 4). In the design view, we investigate the design decisions embedded in a system that induce programmers to copy and paste in particular patterns by analyzing copied code snippets in relation to other code snippets in a system (Section 5). In the maintenance view, we reason the evolutionary aspect of copy and paste by observing how duplicated code snippets are maintained during our study (Section 6). In Section 3~6, we discuss why each view was selected and present the taxonomy that we built by examining copy and paste operations in that view.

Interview

For each participant, we conducted two interviews to understand his/her tasks at a high level and to confirm the intentions that we inferred. We asked questions such as "Can you tell us about your project?" or "why did you choose this text to copy?'' or "why did you duplicate this text?'' or "why did you modify the pasted text in this way?" or "What is the relationship between A and B?" To help subjects to recall the editing context, we sometimes replayed a log file prior to questioning. Each interview lasted from 30 minutes to 1 hour.

Affinity Process

We built an *Affinity Diagram* to identify patterns of copy and paste operations[1]. The *Affinity Process* is a standard generalization procedure to construct a taxonomy. For each C&P instance, we wrote its description on a post-it. Then we put the post-it on the board and moved it or other post-its so that they could form a group and reveal a more general concept. When we drew the same interpretation from multiple C&P instances, we used only one post-it and annotated the number of occurrences on the post-it. Figure 2 shows a picture of a 8 feet by 6 feet white board with the final state of the affinity diagram we built. In total, 153 post-its were used.

[1] More information about how to build an affinity diagram is presented in [BH95].

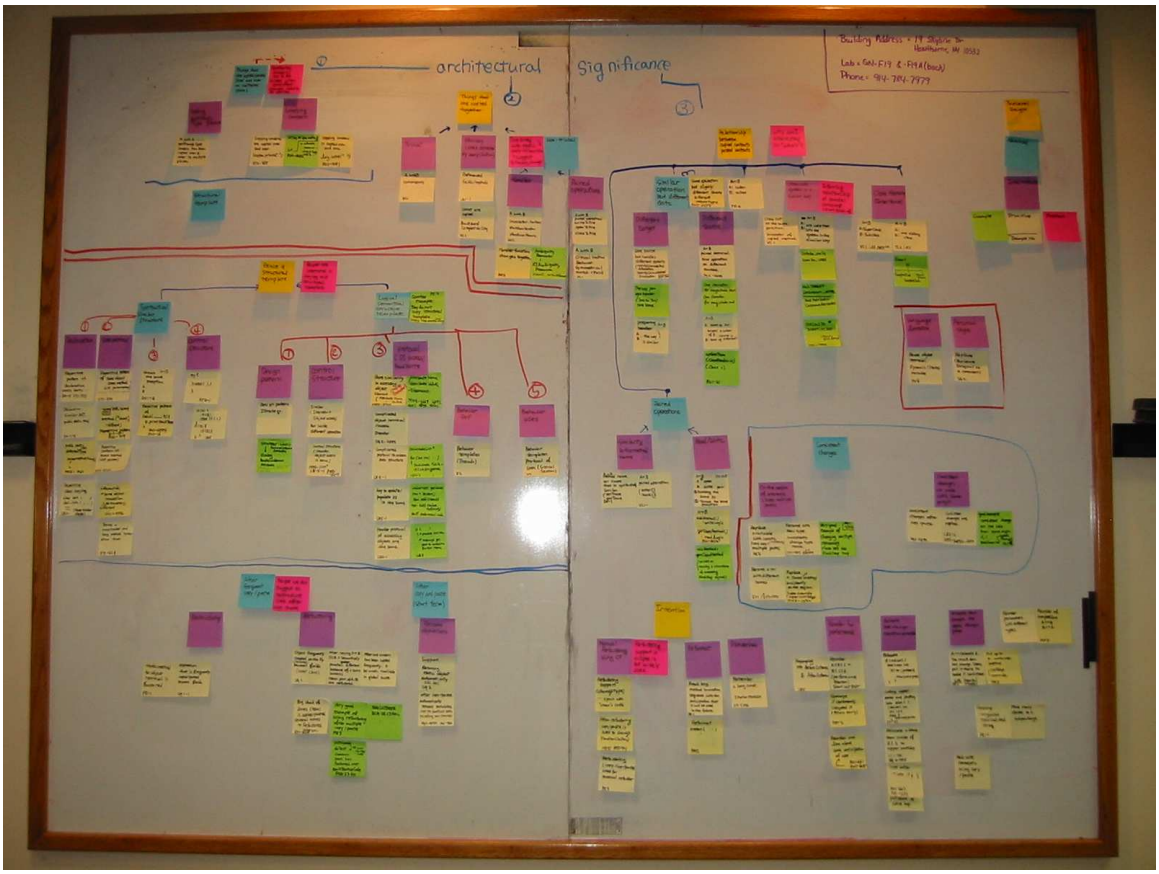

Figure 2 The Affinity Diagram

3 Operation View

In the operation view, an instance is described with a focus on the content of text that is copied (or cut) and the sequence of editing operations. We chose the operation view because each instance cannot be described without a concrete procedural description and the structural entity of relevant text. The procedural steps are the only information observable by the instrumented editor, and this information can help us infer the intention of C&P operations.

We generalize the editing procedure of multiple C&P instances and describe it in the format of an operation model. The operation model is built by extending an and-or graph [RN03]. We annotated the graph with a structural entity of the text that is copied or cut. For example, one frequent C&P pattern that we observed is to repeatedly change the name of a variable. We generalized these copy and paste instances as a hierarchical procedure. The procedure consists of selecting a variable, copying the variable, and pasting the variable for n times, and optionally searching for the variable for n times (where n is the number of appearances of the variable within its scope). A required step is represented as an “and” node and an optional step is represented as an “or” node in the graph. Structural entities are categorized as variables, types, constant declarations, method invocations, blocks, method declarations, and classes. Note that because our instrumented observation is limited to the programming language Java, the categorization is Java specific.

Figure 4 presents the operation model of the copy and paste instance in Figure 3. 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).*

```
public void addMethod(String classType, String
methodName, String signature,
int modifiers) {
    HashSet set =
    (HashSet)classesToMethods.get(classType);
     if (set == null) {
            set = new HashSet();//modified
            classesToMethods.put(classType, set);
    }
    set.add(new MethodInfo(methodName,
    signature, modifiers));
}
```

Figure 3 Copying a variable name "set" and pasting it for multiple times to rename the variable within the "addMethod" method

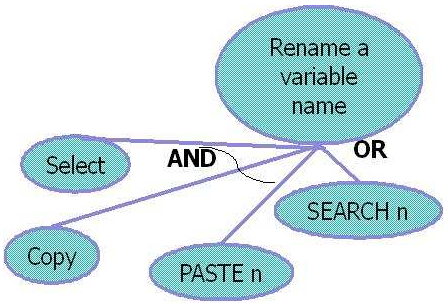

Figure 4 Operation Model of "Renaming a Variable"

By generalizing the editing procedure of C&P instances, we identified the C&P usage patterns and inferred the intentions associated with these patterns. We describe the categorization of the intentions in the next section.

4 Intention View

In the intention view, we examine the high level intents of programmers when they copy and paste. The categorization of programmers' intensions was constructed by asking questions such as "Could you describe why you were copying and pasting?" In our study, the high-level intentions of C&P are as follows:

Relocate /Regroup/ Reorganize
Programmers move a code fragment from one place to another place according to their mental model of a program structure.
Reorder
Programmers use copy and paste to reorder code fragments. For example, a boolean expression (A||B||C) is rewritten as an equivalent expression (B||C||A) for a performance reason. Or several if-blocks are reordered so that negated if-statements return early.
Reformat
Programmers use copy and paste to format a code fragment. For example, a long nested method invocation sequence is broken into several statements with the anticipation that the retrieved object might be used in the future.
Mnemonics
Programmers copy a long complicated variable (type) name to remember its name.
Refactoring Support
Programmers use copy and paste to restructure (or refactor) their code manually.
Reuse as a Structural Template
Programmers intend to use a code snippet as a structural template.

- **Syntactic template**

Programmers copy one code snippet as a syntactic template for the code snippet that they intend to write, because both snippets are textually similar.

- **Semantic template**
  Programmers copy one code snippet as a semantic template for the code snippet that they intend to write, because both snippets are similar in programming logic.

We discuss each sub category of structural templates in detail.

The usage of syntactic templates is explained with the example in Figure 5. The statement "protectedClasses.add("java.lang.Object")" was copied for multiple times. The duplicates were modified after they were pasted. We analyzed that the programmer intended to reuse "protected$$$$.add("java.lang.$$$")" as a template for other statements in the static method initialization. We conjecture that needs of copying syntactic templates are increased by the lack of functionality in today's IDE or limitations in language constructs. For example, the absence of a smart key word completion in an IDE or the lack of "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");
        protectedClasses.add("java.lang.ref.Reference$1");
        protectedClasses.add("java.lang.ref.Reference$Lock");
        protectedMethods.add("java.lang.Thread<init>");
        protectedMethods.add("java.lang.Object<init>");
        protectedMethods.add("java.lang.Thread.getThreadGroup");
}
```

Figure 5 Example of Syntactic Template

Now we describe the categorization of semantic templates (the second sub category of structural templates).

- **Design Pattern**

In our study, we observed one case when a programmer copied a usage of the *Strategy* design pattern. We suspect that the programmer used a concrete instantiation of the Strategy pattern as a template, because it is easier than writing code from an abstract description of that design pattern.

- **Usage of an Interface**

Programmers copy a code snippet to reuse the usage of an interface. We observed many cases where a code snippet was copied because it contains logic for accessing a frequently used data structure. In Java, there are a number of built-in data structures, and programmers are required to know the usage protocol of the data structure that they intend to use. For example, in order to traverse keys in the "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 such cases in our study. One example is shown in Figure 6. The code snippet was copied because it contains frequently used code for traversing over *Element* nodes in a DOM *Document*.

```
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 6 Code Snippet: Traversing over Element nodes in a DOM Document

- **Implementation of an Interface**

Behavior-Def means that a code snippet contains a definition of particular behavior. For example, any thread should implement the run () method. By inheriting built-in classes and interfaces, we believe that it is possible to avoid copying the implementation(or declaration) of an interface.

- **Control Structure**

Programmers frequently reuse a complicated control structure [e.g. 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.

5 Design View

The AOP community observed that primary design decisions that are already embedded in a system sometimes do not allow the secondary design decisions to be modularized in a small module when they are added to the system [TOHS99]. We believe that the lack of modularity leads programmers to insert the similar code snippets across a code base, -- which programmers often carry out by copying and pasting.
In this section, we examine the underlying design decisions that induce programmers to copy and paste in particular patterns. While in the intention view, code snippets involved in each C&P instance were analyzed in isolation, in the design view, we analyze the code snippets in relation to other code snippets in the system. We asked several questions to understand the architectural (or design) context of copy and paste operations. Each sub-section discusses why we chose each question and describes the categorization of answers to the question.

5.1 Why is text copied and pasted repeatedly in multiple places?

We observed that a particular code snippet is 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* [TOHS99] are difficult to separate from the execution context because these concerns require accessing the execution context. For example, the code of a logging concern in Figure 7 was copied and pasted 4 times within one file, and much more across the code base.

For the same reason, extending a feature in software sometimes requires changes in scattered places across a code base. In one project that we observed, a programmer added a feature of displaying a user-friendly type instead of the internally used XML type for the objects in his software. 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 in 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 needs to compromise its modularity with the primary design decision of a system (such as a system architecture), programmers are required to duplicate some code even if it is possible to refactor the body of the secondary design decision.

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

Figure 7 Code Snippet: Logging Concern

5.2 Why are blocks of text copied together?

This question is raised by our observation 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 by answering this question, we can design a tool that recommends which code fragments to copy together. 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:

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 is copied. In one case that we observed, a programmer copied sender.cpp to heartbeat.cpp in order to create a heartbeat thread that has similar behavior as the sender thread. After he finished modifying the heartbeat.cpp, he copied the invocation statement of startsender() and pasted as the invocation statement of startheartbeat() in the test driver file. He also copied the invocation of shutdownsender() and pasted as the invocation of shutdownheartbeat().
Paired operations
Programmers copy and paste paired operations together. For example, when a programmer copies writeToFile(), he also copies openToFile() and closeToFile(). Likewise when enterCriticalSection() is copied, leaveCriticalSection() is copied as well.

5.3 What is the relationship between copied and pasted text?

We raised this question to understand why programmers choose a particular code fragment as a template. We believe that design decisions that already exist in a system not only lead a programmer to duplicate code of a particular concern, but also guide him or her to select a specific code snippet as a template. To answer this question, we examined the relationship of the copied (cut) text and the pasted text. We also believe that understanding the relationship between copied text and pasted text would allow us to assess the significance of dependencies that are created by C&P.

Similar Operations but (from/to) Different Data
This category is a special case of semantic templates where the duplicated code snippets read (write) from (to) different sources (targets). For example, we observed one software tool with a pipeline architecture. The tool sends error messages from one stage to the next stage by calling method A. The tool also notifies a user and displays the error messages by calling method B. A is copied and used as a template for B, because A and B contain logic for reading the same header but only differ in the targets they direct error messages to. In Figure 8 and Figure 9, 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. We believe that we can avoid this sort of copy and paste in functional programming languages by passing a function which reads (writes) from (to) different data.

```
public void updateFrom  (Class c ) {
    String cType =
    Util.makeType(c.getName());
    if (seenClasses.contains(cType)) {
        return;
    }
    seenClasses.add(cType);
    if (hierarchy != null) {
        addToHierarchyViaReflection(c);
```

```
public void updateFrom  (ClassReader cr ) {
    String cType =
    CTDecoder.convertClassToType
    (c.getName());
    if (seenClasses.contains(cType)) {
        return;
    }
    seenClasses.add(cType);
    if (hierarchy != null) {
```

```
    }
    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 8 Code Snippet: updateFrom(Class c)

```
    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 9 Code Snippet: updateFrom (ClassReader cr)

Semantically Parallel Concerns

We define semantically parallel concerns as design decisions that crosscut a system in a similar way. For example, we call the concern of supporting an integer operation and the concern of supporting a floating point operation in a compiler as 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 object. 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. Although these two concerns are independent, code of the "appendChildren" concern should be inserted into the same places where code of the "serialize" concern appears. The programmer identified all the code related to the "serialize" concern by exploiting the fact that information transparent modules are often encoded with the same signature, such as the use of particular variables, data structure, or language features [Griswold99]. Then he copied the identified code snippets and modified them as the code snippets related to 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. The similar case was observed in [Griswold99], when C-Star was retargeted to Ada. The pipeline architecture of CStar guided the programmer to identify all the code related to C specific support and convert them to the code related to Ada specific support.

Paired Operations

In Section 5.2, 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 (mentioned in Section 4). For example, in Figure 10 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.

```
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
```

```
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,
```

```
} else {
    // create an array list and add it to a map
}
// add a method to the array list
```

```
    and if matches, return "m"
} else {
    // if signature matches, return that method
}
```

Figure 10 Write / Read Logic

Inheritance
We observed several cases where a super class is copied and used as a template for subclasses and a sibling class is used as a template for other sibling classes.

In the design view, we examined various kinds of C&P dependencies[2] 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 C&P dependencies reflect important design decisions such as crosscutting concerns, feature extensions, paired operations, semantically parallel concerns, and type dependencies (inheritance).

6 Maintenance View

We investigate maintenance tasks for duplicated code, because failing to perform such tasks may create defects in software. Although this ethnographic study was not a longitudinal study, we approached the maintenance problems associated with copy and paste by raising questions such as "what does a programmer do immediately after a C&P operation?" and "how does a programmer modify code duplicates created by copy and paste?"

Short term
By observing what programmers do right after they copy and paste, we noticed that cautious programmers modify a pasted block to prevent naming conflicts or modify the portion of the pasted code that is specific to the current intended use and is not part of the structural template.

Long term
Programmers restructure (or refactor) their code after frequent copy and paste of a large text. For example, after one code snippet is copied and pasted for multiple times, the code snippet is 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 defines an inner class and creates a member variable that holds the object.

By observing how programmers handle the code duplicates created by copy and paste, we noticed that programmers tend to apply consistent changes to the 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 that code elements that change together must look similar.

7 Statistics

In this section, we present simple statistics about C&P usage patterns. We are interested in understanding how frequently significant C&P dependencies are created in the structure of a program.

[2] In this paper "C&P dependencies" mean the relationships of code snippets involved in C&P operations. For example, the relationship of code snippets that are copied together is one of C&P dependencies. However a term "dependencies between copied code and pasted code" specifically denotes the relationship between copied text and pasted text. Thus "C&P dependencies" include "dependencies between copied code and pasted code".

With the instrumented editor, we observed 460 copy and paste instances for about 50 coding hours. To understand how frequently people copy and paste, we used two measures. The first measure is the rate of an editing operation of a particular type. We define the *rate* of an <type> editing operation as a ratio of (# of <type> operation) to (# of all editing operations). In our observations, the average copy rate is 0.012, the average cut rate is 0.004, and the average paste rate is 0.024. The second measure is a frequency of copy and paste instances (i.e. the number of copy and paste instances / hour). The average number of copy and paste instances per hour is 16 instances / hour and the median is 12 instances / hour.

In order to understand how often copy and paste operations of a certain type occurred, we grouped C&P instances into four different buckets and counted them. We used the structural entity of copied text as a standard for selecting different buckets, because we discovered that there exist correlations between the structural entity of copied text and the intention of C&P.
When text of less than a single line is copied and pasted, the text is likely to be a part of a variable name, a type name, or a method invocation statement. We discovered that it is used as a syntactic template or it is copied in the course of renaming a variable (or a type). We inferred that when a code block is copied, programmers use the block as a semantic template, or insert the code of scattered concerns in a code base. And when programmers copy a method, they use it as a semantic template, or manually refactor it.

Figure 11 shows a distribution of copy and paste instances across 4 buckets of different structural entities. About 74 % of copy and paste instances are in the category of copying text less than a single line. In these cases, we believe that copying was performed to save typing. However, about 25 % of instances were produced by copying and pasting a block or a method. We believe that copying in this category created C&P dependencies that reflect design decisions in a program. When we multiply this percentage (25%) with the average 16 instances / hour, it means that a programmer produces 4 interesting copy and paste dependencies per hour on average.

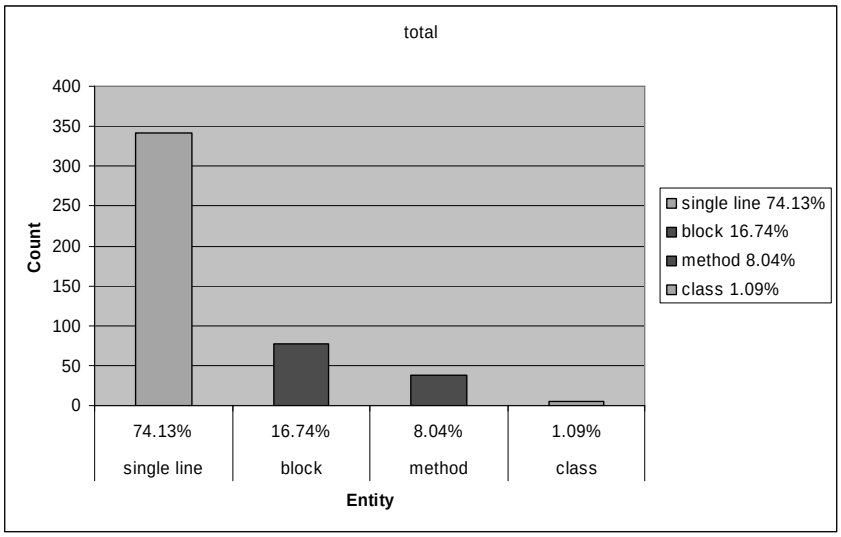

Figure 11 Distribution by Structural Entity

8 Summary of Insights

By constructing a taxonomy of C&P operations and understanding C&P behaviors quantitatively, we obtained in-depth understanding about C&P. In this section, we summarize our insights about C&P.

Unavoidable Duplicates

We suspect that limitations of a particular programming language produce unavoidable duplicates in a code base. For example, the lack of multiple inheritance in Java makes it difficult to impose a certain behavior or an aspect without creating duplicates. Also, during the interviews with the subjects, one subject told us that he would not copy the phrase of "public static final String", if there were the "enum" construct in Java. Because our observations were in the scope of Java (and a few more object oriented programming languages) at this point, we do not know whether this claim would hold for other programming languages. We conjecture that particular C&P usage patterns may not occur in functional programming languages.

Sometimes programmers do not remove code duplicates although it is possible to refactor them, because the units in programming languages such as a function do not match with programmers' conceptual unit of a function.

Delayed Restructuring
We observed that programmers start to restructure code after they copy and paste the same code snippet several times. A few programmers told us that they delay code restructuring on purpose until they copy and paste several times to discover the right level of abstraction. We suspect that larger or frequently copied code fragments are good candidates for refactoring.

Significance of C&P Dependencies.
By observing examples in Section 5.1, we conjecture that the code snippets that are duplicated repeatedly across a code base are what the aspect oriented programming [Kiczales97] community referred as *aspects*. Thus, we believe that C&P dependencies could be used for identifying aspects of a specific kind.

Needs of Maintaining Dependencies between Copied Text and Pasted Text.
Programmers tend to lose track of dependencies between copied text and pasted text. We believe that it is desirable to maintain such dependencies. We found one motivating example from Mozilla open source project. One bug in Mozilla required a programmer to fix bugs that were propagated to 12 different places by copy and paste. A bug was introduced to the code snippet in Figure 12 by invoking "AppendFrames" method instead of "insertFrames" method. The code snippet was copied twice within the same method and the method was copied for three times. Finally 12 structural clones containing that faulty code snippet were produced. The programmer who fixed the bug had to lexically search the code base with comments starting with "XXX" in order to apply the appropriate modifications consistently. We believe that maintaining dependencies between copied text and pasted text would assist tracing code duplicates when programmers need to apply the same change to the structural template of the code duplicates.

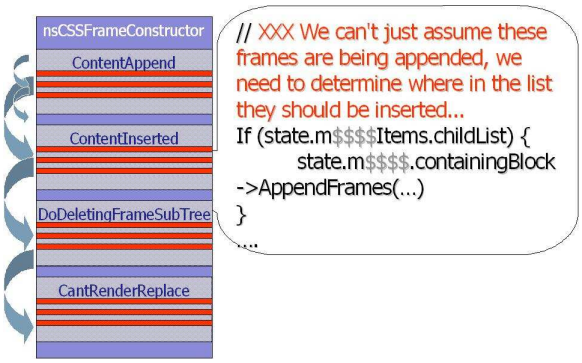

Figure 12 Mozilla Bug 217604

Supporting Structural Templates
We believe that programmers copy the 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 support reuse of structural code templates. We also believe that identifying frequently used structural templates will provide inputs for better programming language design.

9 Proposed Tools

Based on our insights from our study results, we now propose tools that both can minimize software maintenance problems that can be incurred by copy and paste and can support programmers' intentions.

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. It can also provide another level of abstraction by annotating the cluster of duplicated code snippets with the intention of duplication.

Extraction of Structural Templates
We propose a tool that learns a relevant structural template of a code snippet by observing repetitive duplication of a code snippet followed by modifications on it. By extracting a structural template of a code snippet, we can provide an advanced sentence (or block) completion, or we can minimize maintenance problems that can be incurred by failing to modify the portion that is not part of the structural template.

Warning / Notification
From the examples of semantic templates in Section 4, we conjecture that the structural templates may indicate protocols or agreements on the usage of an interface. Using the two previously proposed tools, we can warn a programmer when he or she attempts to change a structural template of code fragments. We can also notify other programmers when there is a change in the structural template. This tool can prevent inconsistent changes in a code base.

Recommendation
Although Eclipse IDE provides a number of automatic restructuring methods, Eclipse IDE does not suggest where to restructure or which refactoring method to use. We believe that by identifying a structural template of code snippets and by monitoring frequency and size of copied text, we can provide smarter code restructuring.

10 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 may not frequently copy a code snippet that contains a complicated control structure because functions can be passed as parameters in functional programming languages. However we believe that OOPLs are widely used in many projects and our study results provide valuable insights for designing software engineering tools for them.

The correlations that we presented in Section 7 are not quantitatively verified yet. At this point, we do not understand the interconnection between four different views in Section 3~6 completely. We may need to conduct more user study that verifies these correlations.

Participants in our study were researchers at IBM TJ Watson Research Center. They were expert programmers in Java and were involved in a small team research project. Our results may not be applicable to larger projects or projects that involve programmers with different levels of expertise in programming.

11 Conclusion

In order to systematically investigate the implication of copy and paste programming practices in terms of software evolution, we conducted an ethnographic study by observing programmers. We built the

instrumented editor and the replayer in order to observe programmers effectively. These tools can be also used as a platform to conduct various kinds of studies for understanding programming practices. Based on our observation, we built a taxonomy of copy and paste usage patterns. We conclude that C&P not only saves typing, but also reflects design decisions made by a programmer. However it creates clones of a structural template, and they require maintenance during software evolution. From our insights about copy and paste programming behaviors, we proposed software engineering tools that can reduce problems incurred by C&P.

References

[BH98] H. Beyer and K. Holtzblatt, Contextual design: defining customer-centered systems. San Francisco, Calif.: Morgan Kaufmann Publishers, 1998

[Fowler00] Martin Fowler. Refactoring: Improving the Design of Existing Code. Object Technology Series. Addison Wesley, 2000

[Griswold99] W. Griswold. Coping with Software Change Using Information Transparency. ICSE 99.

[K03] Personal Communication with a user study expert, John Karat.

[Kiczales97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In ECOOP'97---Object-Oriented Programming, 11th European Conference, LNCS 1241

[LM89] B. Lange and T. Moher: "Some Strategies of Reuse in an Object-Oriented Programming Environment ", Proc. of CHI89, pp.69-73 (1989).

[RC93] M.B. Rosson and J.M. Carroll. Active Programming Strategies in Reuse. In Proc. of the European Conference on Object Oriented Programming (ECOOP '93), Volume 707, pages 4-20 of Lecture Notes in Computer Science.

[TOHS99] P Tarr, H Ossher, W Harrison and SM Sutton, Jr., "N degrees of Separation: Multidimensional separation of concerns," Proc. ICSE 99, IEEE, Las Angeles, May 1999, ACM press, pp. 107-1

[RN03] S. Russell, P. Norvig, “Artificial Intelligence; A modern Approach”, Prentice Hall Series in Architectural Intelligence, pp. 218-219 2003

Acknowledgement

I am very grateful to Lawrence Bergman, Tessa Lau and David Notkin for their advices on this project, and Vibha Sazawal, Annie Ying, and Pradeep Shenoy for their comments on an early draft.