Do We Really Need to Extend Syntax for Advanced Modularity?

Shigeru Chiba[1]
Tokyo Institute of Technology
chiba@acm.org

Michihiro Horie[2]
Tokyo Institute of Technology
horie@csg.is.titech.ac.jp

Kei Kanazawa
Tokyo Institute of Technology
kanazawa@csg.is.titech.ac.jp

Fuminobu Takeyama
Tokyo Institute of Technology
f_takeyama@csg.is.titech.ac.jp

Yuuki Teramoto
Tokyo Institute of Technology
teramoto@csg.is.titech.ac.jp

Abstract

For every new language construct (or abstraction), we have been always developing new syntax. Is this a right approach? In this paper, we propose that, if we develop a new language construct for advanced modularity, we should consider the use of *dynamic text* for designing the construct. We mention that language constructs designed with only syntactic extensions (*i.e.* static text) are not satisfactory in aspect oriented programming. Then we present our two prototype systems to demonstrate language constructs designed with dynamic text. One is synchronous copy and paste and the other is a virtual-file editor named *Kide*. We show how they enable aspect-oriented programming in plain Java.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords Modularity, aspect-oriented programming, dynamic text.

1. Introduction

Providing linguistic mechanisms for modularization is one of the primary concerns of programming language design. A recent trend in such mechanisms is aspect orientation. A number of language constructs for the aspect-orientation mechanisms have been proposed, for example, at a series of AOSD conferences. The most popular one is an *aspect* in AspectJ [12], which consists of pointcut definitions and advice declarations.

An aspect in AspectJ is expressed with a syntactic extension to Java as other language constructs for aspect orientation in Java. This is natural; whenever we invent a new language construct (or abstraction), we have been developing a new syntactic extension. The question we discuss in this paper is "Is this a right approach?"

A significant role of language constructs for modularity is to visually present module structures of programs. Traditional syntactic extensions lexically present the structures and thus they have a limitation in the presentation. Since a program is a one-dimensional array of characters, they can only present hierarchical or nested structures without abstract reasoning such as tracing references through symbolic names. This limitation was not a serious problem until crosscutting concerns are widely recognized.

Some code snippets implementing a crosscutting concern often relate to other secondary concerns. Thus, in an ideal presentation, developers should be able to easily comprehend that those code snippets are included in more than one modules if we follow the principle that every concern is implemented by a different dedicated module. Unfortunately, such an ideal presentation is not available through source code in aspect-oriented languages like AspectJ. Using tool supports like AJDT [18] is mandatory to obtain such an ideal presentation. This problem is also known as obliviousness [5], that is, a developer cannot see which aspect is woven into a class when she is working on that class and not reading the source code of the aspect.

To overcome this problem, we propose using not only static text but also *dynamic* text to design a language construct for modularity. Dynamic text automatically changes its shape depending on the contexts. It enables to express more complex module structures of programs, such as crosscutting structures, than hierarchical ones. The resulting language might be categorized into visual languages, which require tool supports for editing a program, but its pro-

[1] Also, The University of Tokyo and JST CREST.
[2] Currently, IBM Research – Tokyo.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD'12, March 25-30, 2012, Potsdam, Germany.

Copyright © 2012 ACM 978-1-4503-1092-5/12/03...$10.00

grams still consist of only a (dynamically changing) character string. Since tool supports like AJDT are almost mandatory in today's software development, considering tool supports at language design is a natural approach.

In the paper, we present two prototype systems based on this idea. One is synchronous copy and paste and the other is a virtual-file editor named *Kide*. Although the essential ideas behind these two systems are not very unique or novel, we developed these systems to demonstrate the design of language constructs with dynamic text. We show new linguistic extensions to Java for aspect-oriented programming although we do not extend the syntax of Java. Some readers might think that our extensions are not language constructs but just new tool supports because of no syntactic extensions. However, our extensions are integrated into the language and hence they are not genuine tool supports, which are optional when programming.

In the rest of the paper, Section 2 mentions the aims of modularization and a brief introduction of aspect orientation. Then it presents limitations of syntactic extensions. Section 3 describes our idea and presents synchronous copy and paste. Then Section 4 presents Kide. Section 5 discusses related work and Section 6 concludes this paper.

2. Language design for aspect orientation

Modularization is a significant issue in modern programing language design. It has at least four aims: code reuse, information hiding [16], composability, and grouping related code.

Here, *code reuse* is to run the same code snippet at different sites in a program. In other words, it is to avoid code duplication as much as possible. Since a program often contains a number of code snippets that are not identical but alike, most modularization mechanisms enable the reuse in that case by some generic expressions such as parametrization.

Information hiding is to hide the knowledge of design decisions on code snippets included in a module. Only the limited knowledge is visible through its interface from the outside. Implementation details are normally invisible.

Composability allows composing a complete program of arbitrarily selected modules with little or no modification of the module code. For example, a procedure is a minimal form of module and it can be added into a program *as is* without modification. Only the caller-site code in the program must be modified to explicitly invoke it. When the procedure is replaced with another, the code at the caller site must be modified since the name of the called procedure changes. However, the definitions of the old and new procedures are not modified at all.

Finally, *grouping related code* is to increase the spacial locality of code snippets belonging to the same concern. This enables independent development of every group of code, which is a module. This also helps developers efficiently maintain a program. Reasoning about the behavior of the program on a specific concern is made easier since developers do not have to read an entire program when they want to understand that behavior.

Listing 1. DisplayUpdate aspect

```
1 aspect DisplayUpdate {
2   pointcut change(Shape s):
3     execution(void Shape+.set*(..)) && this(s);
4
5   after(Shape s): change(s) {
6     s.display().repaint(s);
7   }
8 }
```

Aspect orientation

Aspect orientation is a modularization technique of a recent trend. It was invented to modularize a *crosscutting* concern. Under the existence of crosscutting concerns, some code snippets in a program belong to multiple concerns. Hence, existing modularization mechanisms are not satisfactory with respect to the criterion of *grouping related code*.

An example of crosscutting concern is a security concern. Suppose that there is a database of academic papers and every access to the database must be checked to confirm that the access is permitted. The code snippets to check this obviously belong to the security concern. On the other hand, they also belong to other application-level concerns, for example, to generate a web page showing a list of papers presented at some past AOSD conference, or to process an HTTP request for registering a new paper in the database.

Since these concerns are functional, some readers might not think that the security-check code belongs to them as well as the security concern, which is non-functional. However, when developers discuss the user interface of obtaining a web page listing papers at a specified AOSD conference, they will want to consider when the security-check code is executed; at the very beginning when the user visits this web site? just before submitting a database query? Except for small applications, the security-check code is not transparent or independent of the code belonging to a functional concern, such as generating a web page.

AspectJ

AspectJ [12] is a programming language designed for addressing the modularity problem of crosscutting concerns. It provides three language constructs such as aspects, pointcut, and advice, and it provides syntactic extensions to Java for them.

An aspect in AspectJ is a main module for implementing a crosscutting concern. Listing 1 is a famous example of aspect, which is an aspect-oriented implementation of Observer pattern [6]. This aspect is part of the implementation

Listing 2. Line class

```
1  class Line extends Shape {
2    int angle, len;
3    void setPos(int nx, int ny) {x = nx; y = ny;}
4    void setLength(int nlen) { len = nlen; }
5    void setAngle(int a) { angle = a; }
6    int getLength() { return len; }
7  }
```

of a figure editor and its concern is to notify a Display object to repaint a window whenever a figure is manipulated by the user and thereby its field is set to a new value. We assume that a figure is represented by Line objects, Rectangle objects, and so forth and Shape is their super class. Listing 2 is an example of Line class.

The DisplayUpdate aspect contains one pointcut change and one advice (with no name). The change pointcut (line 2-3) specifies join points, which are the execution points when a method with a name starting with set (here, * represents a wild card) is invoked on an instance of Shape or its subclass. For example, the join points will include the time when a setLength method on a Line object is invoked. The change pointcut takes a parameter s that is bound to the object on that a set* method is invoked. The body of the advice (line 5-7) in the DisplayUpdate aspect is at line 6 and it is implicitly invoked just after the join points specified by the change pointcut since the advice declaration starts with a keyword after. While executing the advice body, the parameter s is bound by the change pointcut to the object on that the set* method is invoked.

The resulting behavior by the aspect is equivalent to object-oriented code shown in Listing 3 although the object-oriented code is inferior to the AspectJ code with respect to composability. Note that in Listing 3 a call to the advice method in the DisplayUpdate class has been inserted at the end of every method in the Line class. Where the method calls are inserted corresponds to the join points specified by the change pointcut in Listing 1. Since those method-call expressions are embedded in the Line class, decoupling the DisplayUpdate class from the Line class needs code modification to remove those call expressions in the Line class. In AspectJ, decoupling the DisplayUpdate aspect does not need modifying the Line class. The DisplayUpdate aspect can be removed from the rest of the program by just deleting or moving to elsewhere its source file in the source tree.

Listing 3. An object-orientated equivalence

```
1  class DisplayUpdate {
2    static void advice(Shape s) {
3      s.display().repaint(s);
4    }
5  }
6
7  class Line extends Shape {
8    int angle, len;
9    void setPos(int nx, int ny) {
10     x = nx; y = ny; DisplayUpdate.advice(this);
11   }
12   void setLength(int nlen) {
13     len = nlen; DisplayUpdate.advice(this);
14   }
15   void setAngle(int a) {
16     angle = a; DisplayUpdate.advice(this);
17   }
18   int getLength() { return len; }
19 }
20
21 // other subclasses of Shape are not shown.
```

Modularization in AspectJ

AspectJ improves modularity with respect to crosscutting concerns. As for code reuse, AspectJ eliminates redundant method calls in the Line class, which are necessary to invoke the advice method in the DisplayUpdate class in the object-oriented code in Listing 3. Information hiding is preserved as in the object-oriented code since the logic of how to repaint a window is hidden in the advice of the DisplayUpdate aspect. Composability is improved against the object-oriented code since the developers do not have to modify the program when the functionality of the DisplayUpdate aspect is added to the program or when it is removed. They do not have to modify the Line class or other subclasses of Shape.

However, the program in AspectJ is less satisfactory with respect to grouping related code. Listing 1 collects all the code snippets related to the concern of updating a display. On the other hand, the Line class in Listing 2 does not contain all the code related to the Line concern. The Line class in Listing 3 contains a method call to advice. The call to advice belongs primarily to the display-update concern but also to the concern of the Line figure. Since the AspectJ developers do not easily recognize that a call to set* method in Line causes update on a display, they are recommended to use tool supports by an integrated development environment (IDE), such as AJDT [18]. AJDT graphically shows on a source code editor that the advice in DisplayUpdate is invoked after the execution of set* methods in the Line class. The syntactic extension to Java alone does not fully express the modularity introduced by the new language construct, an aspect.

Another problem is that code duplication increases in AspectJ if an advice body needs local contexts different for each join point. Suppose that we want to call the repaint method at line 6 in Listing 1 only when an attribute of the Shape object is changed so that flicker noise will be reduced. Since it is different for every join point how to check whether an attribute is changed, the aspect will have multiple advice

Listing 4. Another DisplayUpdate aspect

```
1  aspect DisplayUpdate {
2    void around(Line line, int x, int y):
3        execution(void Line.setPos(int,int))
4        && this(line) && args(x, y) {
5      if (line.x != x || line.y != y) {
6        proceed(line, x, y);
7        line.display().repaint(line);
8      }
9    }
10
11   void around(Line line, int len):
12       execution(void Line.setLength(int))
13       && this(line) && args(len) {
14     if (line.len != len) {
15       proceed(line, len);
16       line.display().repaint(line);
17     }
18   }
19
20   void around(Line line, int angle):
21       execution(void Line.setAngle(int))
22       && this(line) && args(angle) {
23     if (line.angle != angle) {
24       proceed(line, angle);
25       line.display().repaint(line);
26     }
27   }
28
29   // advice bodies for other subclasses
30   //     :
31 }
```

bodies for each method and subclass as we show in Listing 4. The first advice (line 2-9) is for the setPos method in the Line class while the second advice (line 11-18) is for the setLength method and the third one (line 20-27) is for the the setAngle method. They compare the argument(s) with the value(s) of the corresponding attribute(s) and, if they are different, execute the original method body by proceed and call the repaint method. Although Listing 4 does not show, the aspect will also have an advice body for every method in the other subclasses of Shape. Since the difference is only the condition of the if statement, those number of advice bodies are similar to each other and hence redundant with respect to code duplication.

3. Language constructs designed with dynamic text

Our observation is that the problem mentioned in the previous section is due to using only a *static* syntactic extension for designing a new language construct for modularity. The use of a static syntactic extension is natural since programming languages have been "languages" in that programs are expressed in the form of one-dimensional simple character array. However, this design approach implies a limitation; software development in AspectJ premises tool supports by IDEs with respect to grouping related code.

Our idea is to use not only static text but also dynamic text for expressing a new language construct for modularity. Dynamic text automatically changes its shape while editing and browsing. We borrowed this concept from an interactive web page, in which its contents dynamically change in accordance with the user's input by, for example, embedded JavaScript code. A language using dynamic text is a variation of visual language but it is still a traditional text-based language since visual icons or fancy graphics are not used. Our proposal is that the text that may dynamically change could be considered as a building block when designing a new language construct. Thus we still design a syntactic extension although the syntax deals with not only static but dynamic text.

To further investigate this idea, we have developed two preliminary systems. We present the first one in the rest of this section and the other one in the next section. By using these systems, we illustrate language constructs with dynamic text to extend plain Java for aspect-oriented programming.

3.1 Synchronous copy and paste

"Copy and Paste" is an easiest way to reuse a code snippet. On most source-code editors, developers can *copy* a selected text into an internal memory ("the clipboard") and then *paste* it at a different position. Although this practice is easy to learn and thus it is popular with novice developers, it is known as bad one with respect to code reuse since it degrades the maintainability of programs. After finishing the paste action, a new copy of the code snippet inserted by that action is independent of the original copy of that code snippet. If a developer wants to modify text in the code snippet, she has to edit the two copies of the code snippet at the same time.

To decrease the negative impact of the original copy and paste on maintainability while keeping its intuitiveness, we propose a new mechanism named *synchronous copy and paste*. We also implemented its prototype on Eclipse IDE (Figure 1). Our idea is simple; a pasted copy of text is automatically updated when the original copy is modified by a developer, and vice versa. If there are multiple pasted copies, all the copies (including the original) are updated when one of them is modified. In Figure 1, the code snippets highlighted in green in editor panes are synchronously updated. The proposed mechanism also provides a small window (called "View" on Eclipse) that shows which group of copies of text synchronously update each other. This small window is called *the concerns view* of the synchronous copy and paste. Developers can name every group of copies syn-

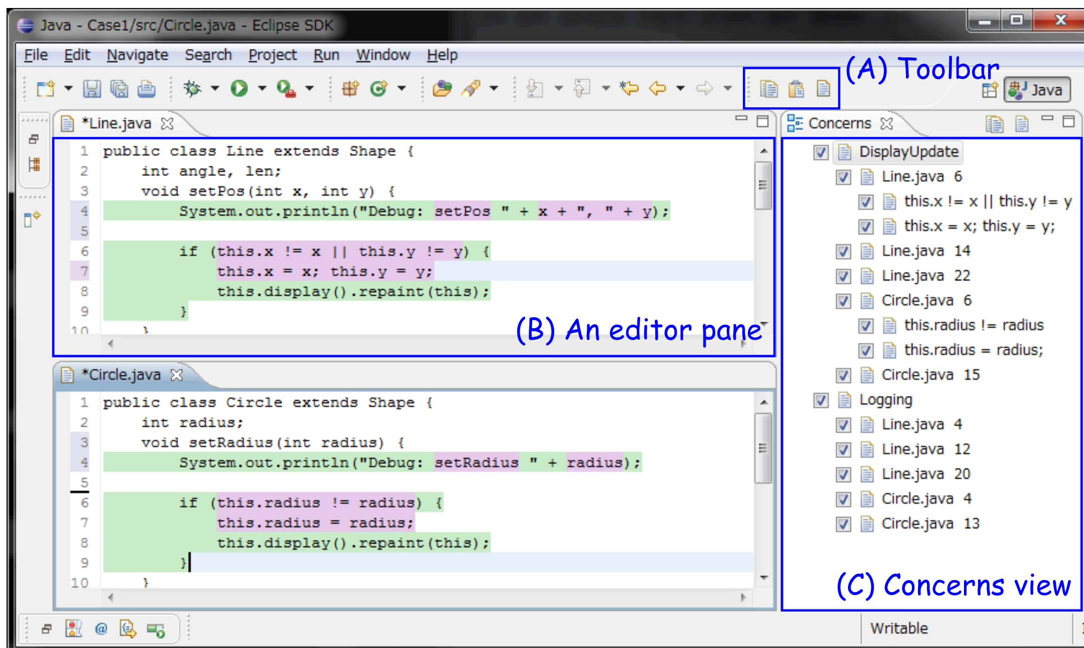

Figure 1. The synchronous copy and paste on Eclipse IDE

chronously updated so that they can easily identify it in the concerns view.

3.2 Procedure abstraction

The synchronous copy and paste is a simple approach to introduce dynamic text into a programming language. It can be used to express a language construct. For example, the expression of procedures can be redesigned with this mechanism. We first present this example to demonstrate our idea although the new design is much worse than typical design of procedure with static text. In Listing 3, the advice method is used as a procedure. Let us rewrite this procedure with our proposed mechanism. We would first modify the setPos method in the Line class into the following:

```
void setPos(int nx, int ny) {
  x = nx; y = ny;
  this.display().repaint(this);
}
```

The method call would be inlined. Then, for the setLength method, we would copy the inlined code and paste it in the setLength method by the proposed mechanism:

```
void setLength(int nlen) {
  len = nlen;
  this.display().repaint(this);
}
```

Since the pasted code is synchronized, if we change it to, for example,

```
void setLength(int nlen) {
  len = nlen;
  for (Display d: this.display()) {
    d.repaint(this);}
}
```

then the body of setPos is also updated (Figure 2). Once the code is "copied," it is listed in the concerns view and the developers can give it an appropriate name to easily identify. This name corresponds to a procedure name. Since the concerns view lists all the code snippets copied before (this information is saved as part of the project), when the developers want to reuse one of those code snippets, they can select its name in the concerns view and paste it at an appropriate place. This corresponds to write a procedure-call expression there.

The proposed mechanism can express a procedure with parameters. Developers can make "holes" in a copied code snippet so that the text in the holes will not be synchronized. The holes are highlighted in magenta in Figure 1. The holes correspond to procedure parameters. If a code snippet has multiple holes, it is possible to synchronize some of the holes and keep them holding the same text. For example, suppose that the two green text-regions in the following code are pasted copies of the same code snippet:

```
int mhp = p.getMaximumHeight();
p.setHeight(mhp < h ? mhp : h);
if (q == null) { return; }
int mhq = q.getMaximumHeight();
q.setHeight(mhq < h ? mhq : h);
```

```
void setPos(int nx, int ny) {                    (1)
    x = nx; y = ny;
    this.display().repaint(this);
}

void setLength(int nlen) {
    len = nlen;
    this.display().repaint(this);
}
```

```
void setPos(int nx, int ny) {                    (2)
    x = nx; y = ny;
    for (Display d: this.display()) {
        d.
}

void setLength(int nlen) {
    len = nlen;
    for (Display d: this.display()) {
        d.
}
```

```
void setPos(int nx, int ny) {                    (3)
    x = nx; y = ny;
    for (Display d: this.display()) {
        d.repaint(this);}
}

void setLength(int nlen) {
    len = nlen;
    for (Display d: this.display()) {
        d.repaint(this);}
}
```

Figure 2. Synchronously updated code snippets are like inlined procedure bodies.

The boxes represent the holes. In the upper green text region, three boxes of mhp and two boxes of p are holes. They are synchronously updated, respectively. This is similar to a macro function in the C language. Note that the text in the boxes is given at every pasted place; different text can be given to (every copy of) the same box at a different place. In the lower green text region, three red boxes hold mhq and two blue boxes hold q instead of mhp and p.

The new design of procedure using the synchronous copy and paste provides acceptable ability with respect to code reuse, composability, and grouping related code. Code reuse is easy; pasting code is a simple and intuitive action. Composition is as simple as traditional procedures with static text. Since the code snippets synchronized with each other are listed in the concerns view, developers can easily identify a group of related code snippets. However, the new design of procedure does not provide information hiding since the pasted code is directly inlined. The details of the code is not hidden at all. To address this problem, the proposed mechanism should be able to fold the pasted text into short text, such as its name, so that it will look like a procedure-call expression. We have not implemented this on our prototype system yet.

Some readers might wonder how to declare local variables when using the proposed mechanism. If a local variable is needed, we can surround the code with braces {} and put a variable declaration into it.[3] The proposed mechanism, however, cannot prevent the pasted code from accessing variables in an outer block. Furthermore, the proposed mechanism does not support recursive procedure calls. To do that, we must introduce something like "delayed paste" to represent an infinite recursive structure. It only indicates that some text will be pasted there on demand (conceptually at runtime). Note that our claim is not that the synchronous copy and paste replaces language constructs, which provide useful abstraction. Our claim is that it can be used for the expression of such language constructs.

Another issue is code size. Since our prototype naively implements the synchronous copy and paste, too many pasting may cause code explosion. This drawback, however, will be reduced if a compiler recognizes synchronous pasting and avoid generating redundant code. Again, the synchronous copy and paste is for just expressing a language construct. It is not a language construct itself.

3.3 Aspects

The synchronous copy and paste can be used in normal Java for expressing an aspect, which is a primary language construct in aspect-oriented programming. We next demonstrate it by rewriting the aspect in Listing 1. To implement this modularity in AspectJ, we start with the object-oriented code in Listing 3. Since this code contains several method calls to advice and those calls are redundant, we use the synchronous copy and paste so that all the call expressions to advice will be synchronized for updates. This improves the code reuse in Listing 3. Information hiding by the after advice in Listing 1 (line 6) is achieved by the advice method in the DisplayUpdate class in Listing 3 (line 3).

Although the obliviousness property [5] by AspectJ is not provided, composability is also improved than the original Listing 3, which has the problem mentioned in the previous section. In the original Listing 3, every method call to advice must be manually deleted when the DisplayUpdate class is decoupled from the Line class and other subclasses of Shape. Now it is easier to delete all the method calls all at once (or change them to empty expressions) since they are synchronously updated. Restoring the deleted method calls in the Line class is also easy since the synchronized code snippets remain as blanks after the method calls are deleted.

The synchronous copy and paste also improves grouping related code. Since the concerns view presents where the method calls to advice are pasted in the program, developers can see at a glance the overview of where all the code snippets of the display-update concern are. This is similar to

[3] Java does not allow variable declarations with the same name as a variable declared in an outer block. Hence this approach does not provide equivalent ability in Java to local variables within a procedure.

the information available from the DisplayUpdate aspect in Listing 1, that is, about the code related to the display-update concern.

Furthermore, unlike in AspectJ, developers can also see the entire code of the Line concern, which includes the method calls to advice. Since the method calls to advice belong to both concerns, showing this fact with traditional language constructs with static text is difficult. If only static text is used, every code snippet must belong to a single lexical module such as a source file. Otherwise, it would make extra code duplication. Dynamic text provides better flexibility to the language design as we showed above with the synchronous copy and paste.

The aspect in Listing 1 can be rewritten in another approach. This is preferable for addressing the problem of AspectJ shown in Listing 4 in the previous section. This problem was that code duplication increases if an advice body needs local contexts different for each join point. In fact, the DisplayUpdate aspect in Listing 4 has a number of similar advice bodies for every join point. Since the differences are only the condition part of the if statement, those advice bodies are somewhat redundant.

This redundancy is minimized if we use the synchronous copy and paste. See a normal Java program in Figure 3, where the advice bodies in Listing 4 are inlined in each corresponding method. Although this naive program may look problematic with respect to modularity, the synchronous copy and paste reduces this modularity problem. Suppose that the entire body of every method is synchronized for updates but the condition part of the if statement and the first statement of the then block are "holes," which are not synchronized. This text synchronization works as a macro function and hence improves code reuse.

Furthermore, the synchronous copy and paste provides the concerns view, which presents the same information that the pointcut in the AspectJ aspect in Listing 4 provides. It gives a view of the code snippets implementing the display-update concern as that AspectJ aspect does. The developers can easily see two views of group of related code: one is an editor pane of Line.java for the Line concern (and other shape concerns) and the other is the concerns view in Figure 4 for the display-update concern. Note that code snippets are shared by both views. In AspectJ, since the if statement and the call expression to repaint, such as line 4 and 6 in Figure 3, are contained only in the aspect in Listing 4, the entire overview of the code belonging to the Line concern is difficult to see without an appropriate tool support. Moreover, the concern view in Figure 4 presents the text in every "hole". This corresponds to arguments passed to an advice body in AspectJ although an aspect in AspectJ abstracts formal parameters from them. To obtain the information available in the concerns view, AspectJ developers must read the class declaration where the aspect is woven.

```
*Line.java
1  public class Line extends Shape {
2      int angle, len;
3      void setPos(int x, int y) {
4          if (this.x != x || this.y != y) {
5              this.x = x; this.y = y;
6              this.display().repaint(this);
7          }
8      }
9      void setLength(int len) {
10         if (this.len != len) {
11             this.len = len;
12             this.display().repaint(this);
13         }
14     }
15     void setAngle(int angle) {
16         if (this.angle != angle) {
17             this.angle = angle;
18             this.display().repaint(this);
19         }
20     }
21     int getLength() { return len; }
22 }
```

Figure 3. An aspect-oriented Line class in Java

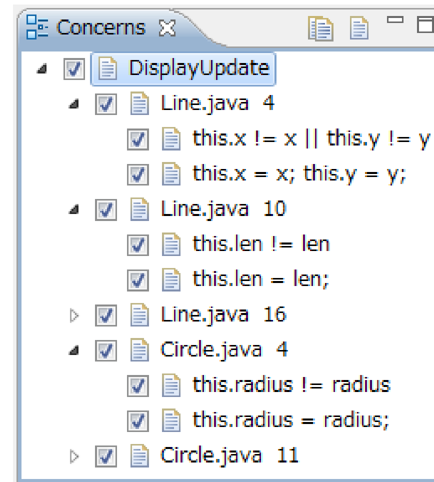

Figure 4. The concerns view for the synchronous copy and paste

4. Kide

We next show another prototype system for using dynamic text. We developed this system named *Kide* for investigating our idea of using dynamic text for modularizing a crosscutting concern (Figure 5). It is a plug-in for Eclipse IDE.

As we mentioned in Section 2, AspectJ assumes that every code snippet belongs to a single concern. An aspect in AspectJ is designed for moving a code snippet from an irrelevant class, where otherwise the code snippet would be placed, to an appropriate aspect, a special module provided

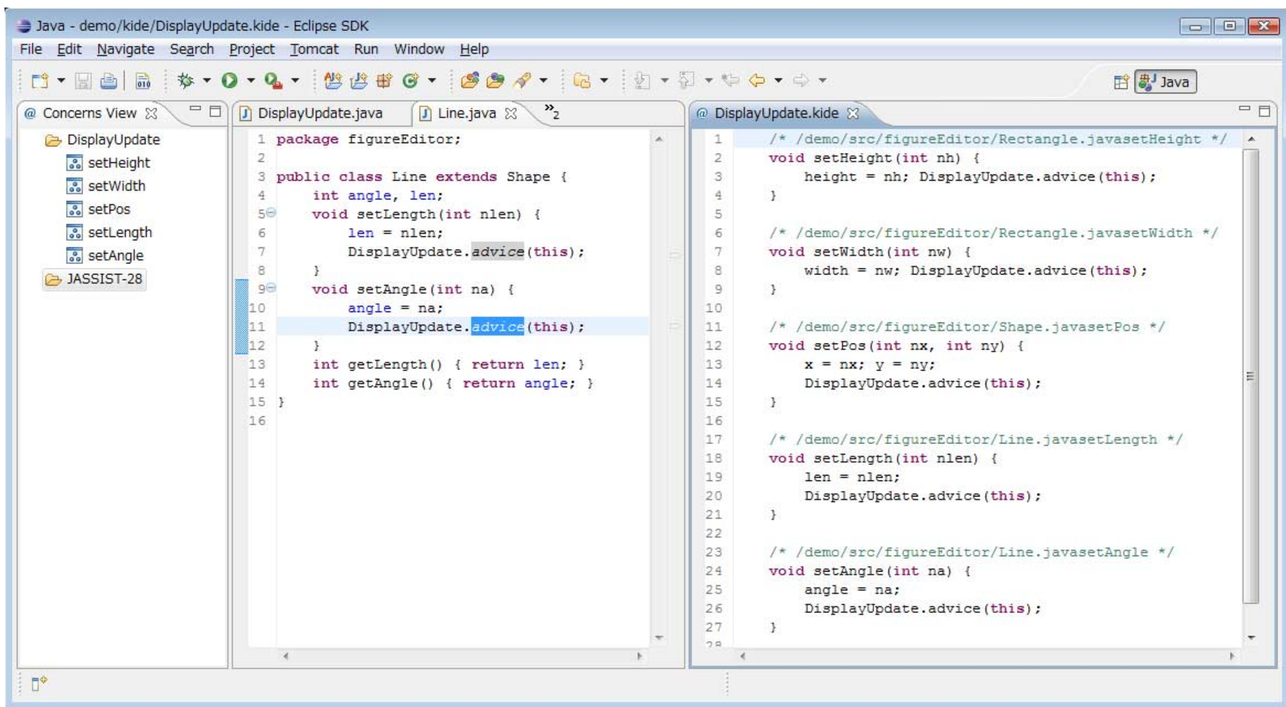

Figure 5. Showing the DisplayUpdate concern by Kide

by AspectJ as well as a class, if that code snippet is for a crosscutting concern. However, such code snippets often belong to not only the crosscutting concern but also other non-crosscutting concerns. Since an aspect in AspectJ is based on static text, it cannot directly show that the other concerns also need those code snippets; some extra tool supports are required to know the fact. For example, the method call to repaint at the line 6 in Listing 1 belongs to not only the display-update concern implemented by the aspect, but also the concerns of the Line class and other subclasses of Shape.

Kide allows developers to make a virtual source file that contains code snippets taken from other source files in Java. Hence one code snippet can be contained in multiple source files (one physical source file and multiple virtual source files). Since all the copies of a code snippet are shared, editing it in one source file is immediately reflected on other source files. In the current implementation of Kide, only a constructor, field, and method declaration can be collected into a virtual source file. Collecting part of a method body is not allowed. A virtual source file is saved as part of a project; once making it, it is always available until developers explicitly delete it.

Kide provides a support mechanism for selecting method declarations to be collected. While developers can manually select method declarations one by one, they can also select several method declarations at once. Kide currently allows selecting all the methods overriding a specific method or all the methods calling a specific method through wizard-like user interface (Figure 6). It is also possible to further screen the methods by hand after selecting by the wizard.

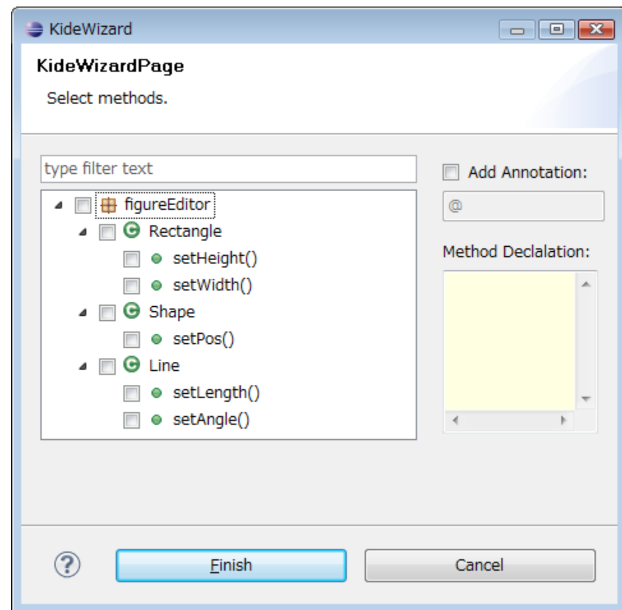

Figure 6. A wizard of Kide for selecting methods shown together

The virtual source files specified by developers are listed in a small window named *the concerns view*. Developers must give an appropriate name to a virtual source file. For every virtual source file, the concerns view presents its name and the names of the methods, fields, and/or constructors collected in that file. If developers open a virtual source file in the concerns view, a source code editor is launched and then they can edit that file. In a virtual source file, every method (and field, constructor) has a comment indicating the physical source file of its origin. The order of listing those methods can be changed by developers. The editor of a virtual source file allows developers to insert the same text at the beginning of all the method bodies collected in that source file or just before all the return statements contained in the methods (if no return statement, at the end of the method bodies).

Aspect orientation by Kide

Kide can be used to do aspect-oriented programming in Java without syntactic extensions. A virtual source file can be used as a module containing methods and fields related to a crosscutting concern. Since code snippets for a crosscutting concern often belong to other (non-crosscutting) concerns, Kide provides better modularity than AspectJ with respect to grouping related code. A code snippet can be contained in multiple source files and thus developers can browse and edit at a glance in one source file all the code snippets for a particular concern.

For example, when developers are working on the Line concern, they can open and edit a file Line.java, which contains the Line class shown in Listing 3 and thus all the code snippets related to the Line-figure concern. Note that, although calls to the advice method (line 10, 13, and 16) belong to the display-update concern, they also belong to the Line concern since they are part of the behavior of Line objects. When developers are interested in the display-update concern, they can make a virtual source file DisplayUpdate and collect into this file all setter methods in subclasses of Shape, such as setPos and setLength. A getter method, such as getLength, is not collected into the virtual file. Then developers can see all the methods related to the display-update concern by opening that virtual file.

Crosscutting over documentation

A virtual source file in Kide can contain not only Java code but also plain text taken from a separate text file. Like a method declaration, developers can select a text block surrounded by XML tags and put it in a virtual source file.

This helps literate programming [13] since it allows mixing code and text in one file. In practice, it is useful for bug tracking during software maintenance. When a development team receives a bug report, they investigate the bug, discuss how to fix it, implement the fix, and finally check the new code into the source code repository. This activity is often managed by a bug tracking system, which records a chat log among developers and the revision numbers of the related source files before/after the fix. When a similar bug is reported later, developers refer to this recorded chat log and understand how the source files are changed for fixing the original bug.

```
@ JASSIST-28.kide
1    Caused by: java.lang.LinkageError: duplicate class definition: de/mueller/
2 at java.lang.ClassLoader.defineClass1(Native Method)
3 at java.lang.ClassLoader.defineClass(ClassLoader.java:620)
4 at sun.reflect.GeneratedMethodAccessor2.invoke(Unknown Source)
5 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImp
6 at java.lang.reflect.Method.invoke(Method.java:585)
7 at javassist.util.proxy.FactoryHelper.toClass(FactoryHelper.java:159)
8 ... 46 more
9
10 The performance is also very bad in comparison with cglib proxies (in our perf
11
12 Solution:
13 The folloing are the methods I changed for fixing this bug.
14
15     /* /javassist/src/javassist/CtClass.java/addConstructor */
16     public void addConstructor(CtConstructor c)
17         throws CannotCompileException
18     {
19         checkModify();
20         if (c.getDeclaringClass() != this)
21             throw new CannotCompileException("cannot add");
22
23         getConstructorsCache();
24         constructorsCache = (CtConstructor)CtMember.append(constructorsCache,
25         getClassFile2().addMethod(c.getMethodInfo2());
26     }
27
28     /* /javassist/src/javassist/CtClassType.java/addMethod */
29     public void addMethod(CtMethod m) throws CannotCompileException {
30         checkModify();
31         if (m.getDeclaringClass() != this)
32             throw new CannotCompileException("cannot add");
33
34         getMethodsCache();
35         methodsCache = CtMember.append(methodsCache, m);
36         getClassFile2().addMethod(m.getMethodInfo2());
37         if ((m.getModifiers() & Modifier.ABSTRACT) != 0)
38             setModifiers(getModifiers() | Modifier.ABSTRACT);
```

Figure 7. A bug report written by Kide

Kide helps developers write a detailed report of how they fixed a bug for future reference. Kide makes it easy to write a documentation quoting several method declarations from source files. We demonstrate this idea by using a bug report for Javassist [2], which is a Java bytecode engineering library widely used by a number of software including commercial products such as Red Hat JBoss AS. One day, the development team received a bug report that pointed out that the software consumes too much memory.[4] The team found that the bug is not memory leak but decided to introduce a mechanism to reduce memory consumption. Then they implemented a prune method, which discards unnecessary information of a specified class file. This method is invoked when a class file is loaded by the Java virtual machine. The implementation of the prune method involves several support methods in different classes. Furthermore, since after calling the prune method, part of the Javassist functionality on the class is not available, the development team had to modify several methods so that they would check at runtime that the prune method is not called yet. A method call to checkModify was inserted at the beginning of every method that accesses the information discarded by the prune method. After the prune method is called, the checkModify method

[4] https://issues.jboss.org/browse/JASSIST-28

throws a runtime exception to notify that the functionality of Javassist is not available.

If Kide were available, the development team could write a virtual source file that includes the original bug report, the description of how to change the source code for it, and the implementations of the prune method and its support methods. The methods calling the checkModify method could be included in that documentation since they are part of the *pruning* concern (Figure 7). Both code and documentation crosscutting over a number of classes and files are collected into a single file. The resulting file will be useful for future bug fixes and maintenance.

5. Related work

There have been a number of mechanisms similar to the synchronous copy and paste or Kide. Our main contribution is to propose the use of those mechanisms for designing a language construct for advanced modularity. Those mechanisms should not be supplementary tool supports provided by IDEs but they should be tightly integrated into a language.

Code clone

Simultaneous editing [15] and Linked Editing [19] are the most direct related work of the synchronous copy and paste. Both were developed for managing duplicated code or code clone. They allow developers to edit distinct code regions simultaneously as they do with the synchronous copy and paste. Simultaneous editing provides a generalization mechanism so that the system automatically generalizes developers' editing actions to apply other code regions. Like our synchronous copy and paste, a motivation of the study of Linked Editing is to overcome limitations of the language constructs with static text with respect to duplicated code.

However, either simultaneous editing or Linked Editing does not provide the correspondent to the concerns view of the synchronous copy and paste. The concerns view is significant since the design focus of the synchronous copy and paste is on modularity instead of code duplication whereas Linked Editing is a tool mainly for managing code duplication than modularity as pointed out in [9]. Furthermore, the user interface when dealing with more than two code regions synchronously updated together would be inefficient without the concerns view. Giving a name to a group of code regions synchronously updated is also important for modularity. This ability is also provided by the concerns view.

The idea of synchronous updates of copy-and-paste induced code clone is also found in other systems. For example, CReN [8] is a tool for detecting code clones and maintaining clone evolution by synchronous updates. It enables automated consistent renaming of program elements such as variables in synchronized code clones. The main focus of those systems, however, is on automated maintenance of code clones. It is not on providing better views of programs with respect to modularity.

Aspect orientation

In the contexts of aspect-oriented programming, several tool supports have been proposed for providing multiple views of programs for developers. Mylar [11] (now Eclipse Mylyn) monitors developers' editing actions and automatically detect a group of related source files. This group corresponds to a concern that the developer was working on during a certain period. For example, when the developer is working on fixing a bug, Mylar records a group of files that the developer opens and edits for that work. Mylar uses the information of this group for presenting only interested elements in windows such as a package explorer and an outline view. It filters out unnecessary source files and program elements by using that information. A difference from our work is that Mylar is a support tool and it deals with temporal concerns whereas our work deals with concerns on program structures. Furthermore, unlike Kide, a source code editor of Mylar does not dynamically change the expression of a program according to the developer's current concern.

Fluid AOP [7] is an IDE for aspect-oriented programming (AOP). Like our work, the goal of Fluid AOP is to overcome limitations of language constructs designed with static text. Fluid AOP still uses AspectJ-like syntax but also utilizes dynamic text. It simultaneously provides different views of a program independently of actual source files. A difference is that Fluid AOP uses AspectJ-like pointcut definitions for specifying module structures whereas our work does not introduce extra syntactic extensions to Java; our developers do not have to learn the pointcut language. They can specify module structures by copy-and-paste actions or wizard-based user interface. Furthermore, unlike Kide, Fluid AOP cannot deal with documentation.

A fluid source code view [3] is an source code editor in which, when the developer clicks a method-call expression, the declaration of that called method is superimposed on that method-call expression. This reduces navigation efforts of the developer. Although a fluid source code view can present the declarations of multiple methods in a single editor pane, the methods presented together must have caller-callee relations. In Kide, it is possible to present the declarations of arbitrary selected methods in a single editor pane.

Other systems

CIDE [10] is an Eclipse-based tool for decomposing applications into features. Developers can mark a code snippet with a color of the feature that the code snippet belongs to. After coloring, developers can easily include/exclude the code snippets belonging to a specific feature as they can do with a preprocessor directive `#ifdef`. CIDE also provides a navigation panel similar to our concerns view. The difference is that CIDE does not present a view of virtual source file including all the program elements related to a specific feature. It can change the expression of a program only within an actual source file. On the other hand, a virtual source file of

Kide can collect all program elements from different source files. Kide is better designed for dealing with crosscutting concerns.

Code Bubbles [1] is an IDE in which developers can make a small editor pane for a method and freely place it on the screen. The small editor is called *a bubble*. With bubbles, developers can see and work with a complete working set of related code-snippets. Although the Code Bubbles IDE does not provide a wizard-based support tool, which Kide provides, to collect methods related to a specific concern, a set of bubbles could be used as a replacement of a virtual source file in Kide. However, Kide gives developers a uniform view of group of related code snippets, which is a source file, whichever the code snippets are collected from other various source files or a single physical source file. A source file is a traditional metaphor of module.

Cedalion [14] supports projectional editing. It allows programmers to directly edit an abstract syntax tree through a human-readable textual view of that tree. Since the textual view is a projection of the tree, it can contain special symbols and be displayed with different fonts and unorthodox layout. The edit-time MOP [4] also enables extended presentation of source programs. These systems share a similar idea with our proposal but their primary focus is on internal domain-specific languages.

The idea of virtual source files in Kide is also found in Desert [17]. Desert is an IDE in which developers can create a virtual file to make navigation among code snippets efficient. Desert can generate a virtual file that contains code snippets including compilation errors or matching a given search pattern. Our contribution is that we pointed out that virtual source files are useful abstraction for crosscutting modularity.

6. Conclusion

In this paper, we presented limitations of syntactic extensions for new language constructs for modularity, specifically, aspect orientation. Then we proposed using dynamic text when designing new language constructs. To demonstrate this idea, we presented the synchronous copy and paste and Kide. We illustrated concrete examples of language constructs with dynamic text as aspect-oriented programming extensions to Java without modifying the original syntax. Although there have been a number of similar linguistic mechanisms like ours, our main contribution is that we showed that designing language constructs with dynamic text is useful for aspect oriented programming. Another message is that language designers should consider tool supports as not optional features but integrated part of programming language design.

Acknowledgments

This work has been strongly influenced by our joint work on universal AOP with Awais Rashid, Ruzanna Chitchyan, and Phil Greenwood. We would also like to thank Eric Tanter, Romain Robbes, and the members of the PLEIAD group for their insightful comments.

References

[1] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering (ICSE '10)*, pages 455–464. ACM, 2010.

[2] S. Chiba. Load-time structural reflection in Java. In *ECOOP 2000*, LNCS 1850, pages 313–336. Springer-Verlag, 2000.

[3] M. Desmond, M.-A. Storey, and C. Exton. Fluid source code views. In *14th IEEE Int'l Conf. on Program Comprehension (ICPC'06)*, pages 260–263. IEEE, 2006.

[4] A. D. Eisenberg and G. Kiczales. Expressive programs through presentation extension. In *Proc. of 6th Int'l Conf. on Aspect-Oriented Software Development (AOSD 2007)*, pages 73–84. ACM, 2007.

[5] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, 2005.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[7] T. Hon and G. Kiczales. Fluid aop join point models. In *Proc. of ACM OOPSLA*, pages 712–713. ACM, 2006.

[8] D. Hou, F. Jacob, and P. Jablonski. Exploring the design space of proactive tool support for copy-and-paste programming. In *Proc. of the 2009 Conf. of the Center for Advanced Studies on Collaborative Research (CASCON '09)*, pages 188–202. ACM, 2009.

[9] C. J. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[10] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. of the 30th Int'l Conf. on Software Engineering (ICSE '08)*, pages 311–320. ACM, 2008.

[11] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proc. of the 4th Int'l Conf. on Aspect-Oriented Software Development (AOSD '05)*, pages 159–168. ACM, 2005.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pages 327–353. Springer, 2001.

[13] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[14] D. H. Lorenz and B. Rosenan. Cedalion: a language for language oriented programming. In *Proc. of ACM OOPSLA*, pages 733–752. ACM, 2011.

[15] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *Proc. of the General Track:*

2002 USENIX Annual Technical Conference, pages 161–174. USENIX Association, 2001.

[16] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[17] S. P. Reiss. Simplifying data integration: The design of the desert software development environment. In *Proc. on Int'l Conf. on Software Engineering (ICSE '96)*, pages 398–407. IEEE, 1996.

[18] The Eclipse Foundation. AspectJ development tools (ajdt). http://www.eclipse.org/ajdt.

[19] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *2004 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 173 –180, 2004.