An Introduction to **Aspect-Oriented Programming**

By Ken Wing Kuen Lee <cswkl@ust.hk>

Reading Assignment

COMP 610E 2002 Spring Software Development of E-Business Applications

The Hong Kong University of Science and Technology

Table of Contents

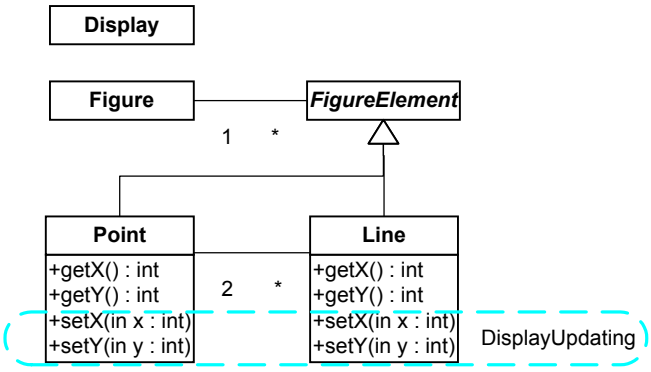
1 Background

1.1 Crosscutting Concerns

The programming language has evolved from machine codes and assembly languages to variety paradigms such as formula translation, procedural programming, functional programming, logic programming, and object-oriented programming. The programming technology advancement has improved the ability of software developer to achieve clear *separation of concerns*, or "the ability to identify, encapsulate, and manipulate only the parts of software that are relevant to a particular concept, goal, or purpose" [OT2001]. Nowadays, object-oriented programming has become the dominant programming paradigm where a problem is decomposed into objects that abstract behaviour and data in a single entity. Object technology, including OO methodologies, analysis and design tools, and OO programming language, reduces the complexity in writing and maintaining complex applications such as distributed applications and graphical user-interfaces.

Although OO technology offers greater ability for separation of concerns, it still has difficulty localizing concerns which do not fit naturally into a single program module, or even several closely related program modules. Concerns can range from high-level notions such as security and quality of services to low-level notions like buffering, caching, and logging. They can also be functional, such as business logics, or non-functional, such as synchronization. Some concerns, such as XML parsing and URL pattern matching, usually couple with a few objects, yet achieve good cohesion. Other concerns, such as logging, will intertwine with many highly unrelated modules.

We say that two concerns *crosscut* if the methods related to those concerns intersect [EAKLO2001]. In additional, crosscutting concerns cannot be neatly separately from each other. Consider the UML for a simple figure editor described in [EAKLO2001]. There are two concerns for the editor: keep track of the position of each figure element (data concern) and update the display whenever an element has moved (feature concern). The OO design nicely decomposes the graphical element so that the data concern is neatly localized. However, the feature concern must appear in every movement method, crosscutting the data concern. The software could be designed around the feature concern; but then the data concern would crosscut concern for display update.



• Figure 1: The UML of a simple figure editor.

The two concerns in the figure editor example crosscut each other no matter how the design is. Only one concern can be localized neatly by the OO paradigm. Other concerns cannot be encapsulated within the dominant modules, ending up being scattered across many modules and tangled with one another. This phenomenon is called the tyranny of the *dominant decomposition*, in which one dominant way of decomposition imposes software structure that makes separate of concerns difficult or impossible. This problem is difficult to avoid in many systems designed in OO paradigm:

- Example 1: Synchronization

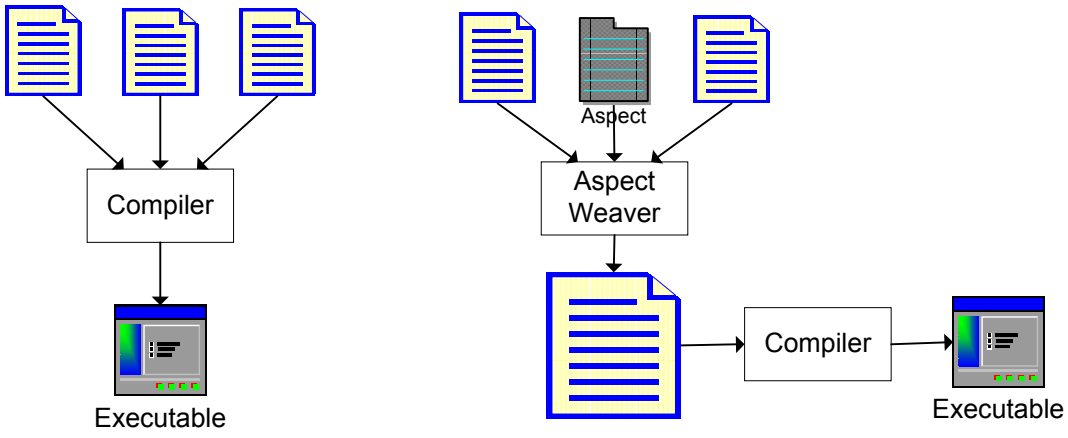
This problem is described in [GILLLMM1996] and [DC1997]. A distributed operating system contains many features, such as synchronization, scheduling, buffering, pre-fetching, security, etc. Consider the problem of implementing synchronization between concurrent objects in OO paradigm. The implementation of the synchronization policy, such as multiple-readers/single-writer, must be embedded into the definition of the objects whose activity is being synchronized, causing *tangling of aspects*. Code reuse is minimal, while maintenance cost increases.

- Example 2: Logging

Logging is frequently used in distributed applications to aid debugging by tracing method calls. Suppose as part of e-commerce project convention, all developers must do logging at both the beginning and the end of each function body. Clearly, logging crosscut all classes that have at least one function.

1.2 Introduction to AOP

Aspect-oriented programming (AOP) is a new technology for separation of crosscutting concerns into single units called aspects. An *aspect* is a modular unit of crosscutting implementation. It encapsulates behaviours that affect multiple classes into reusable modules. *Aspectual requirements* are concerns that introduce crosscutting in the implementation. Both synchronization and logging in previous examples are aspects. With AOP, each aspect can be expressed in a separate and natural form, and can then be automatically combined together into a final executable form by an *aspect weaver*. As a result, a single aspect can contribute to the implementation of a number of procedures, modules, or objects, increasing reusability of the codes.



- Figure 2: In contrast to the traditional approach, AOP allows codes and aspects to be woven together by an aspect weaver before the program is compiled into an executable. In case of the simple figure editor, the weaver can automatically insert displayUpdate call to all set operations, thus allows separation of crosscutting concerns at source code level.

An AOP language has three critical elements for separating crosscutting concerns: a join point model, a means of identifying join points, and a means of affecting implementation at join points [EAKLO2001, KHHKPG2001]. The join point model provides the common reference frame to define the structure of crosscutting concerns, and to describe the "hooks" where enhancements may be added. The elements will be explained more details in Section 2 by using AspectJ.

AspectJ is one of the highly available AOP languages. Another AOP language is Hyper/J, which supports multidimensional separation of concerns. Indeed, Hyper/J is more powerful than AspectJ because Hyper/J supports augmentation of multiple models while AspectJ supports integration of a single model. Nevertheless, I will focus on AspectJ because it is sufficient for understanding the concept of AOP. Next section will introduce AspectJ by showing how it can be used to separate the crosscutting concerns.

2 AOP Languages

2.1 AspectJ

AspectJ is a general-purpose AO extension to Java, developed by Xerox Palo Alto Research Center, which enables plug-and-play implementations of crosscutting in Java.

AspectJ development tool consists of a set of command line tools, a set of GUI tools and online documentations. The command line tools include a compiler for AspectJ language (ajc), a debugger for Java class files produced by ajc (ajdb), and a documentation generator (ajdoc) that produces HTML API document with crosscutting structure. The GUI tools are: AspectJ browser for navigating crosscutting structure and compiling with ajc; AspectJ Development Environment (AJDE) support for both JBuilder and Forte; and environment support for GNU Emacs/XEmacs. Each tool is distributed as a self-extracting Java-based GUI installer. The installer can be invoked by the command `java -jar aspectj-tools.jar`.

Section 2.1.1 introduces some basic language constructs of AspectJ. AspectJ documentation [AJPg2002] contains a complete description of all language constructs of AspectJ. The small example in Section 2.1.2 (based on the example in [IBM2002]) will be used to illustrate how to use AspectJ to solve the crosscutting problem.

2.1.1 Basic Language Constructs

AspectJ is a linguistic-based AOP language. It is defined by a set of language constructs: a join point, pointcuts, advice, introduction and aspects. Pointcuts and advice dynamically affect program flow, while introduction statically affects the class hierarchy of a program.

2.1.1.1 Join Point

A *join point* is a well-defined point in the flow of a program. AspectJ has several kinds of join points:

- Method/constructor call join points
- Method/constructor execution join points
- Field get and set join points
- Exception handler execution join points
- Static and dynamic initialization join points

For example, a method call join point is a point in the flow when a method is called, and when that method call returns. The lifetime of the method call join point includes all the actions that comprise a

method call, starting after all arguments are evaluated up to and including normal or abrupt return. Each method call itself is one join point.

2.1.1.2 Pointcut

A *pointcut designator*, or simply *pointcut*, select particular join points by filtering out a subset of all join points, based on defined criteria. The criteria can be explicit function names, or function names specified by wildcards. Pointcuts can be composed using logical operators. Customized pointcuts can be defined, and pointcuts can identify join points from different classes. Examples of pointcuts are as follow:

Pointcuts	Descriptions
`call(void Point.setX(int))`	Identifies any call to Point.setX(int).
`Call(void Point.setX(int)) ||` `call(void Point.setY(int))`	Identifies any call to either Point.setX or Point.setY.
`call(public * Figure.*(..))`	Identifies any public method defined on Figure.

`pointcut move():`
`  call(void Point.setX(int))  ||`
`  call(void Point.setY(int))  ||`
`  call(void Line.setP1(Point) ||`
`  call(void Line.setP2(Point));`
	Define a pointcut called move. Notice that this pointcut crosscuts different classes.

2.1.1.3 Advice

Pointcuts are used in the definition of advice. An *advice* in AspectJ is used to define additional code that should be executed at join points. AspectJ has the following advices:

- *Before advice*: the code executes when a join point is reached but before the computation proceeds.
- *After advice*: the code executes after the computation of under a join point has completed, but before the exit of that join point.
- *Around advice*: the code executes when the join point is reached and conditions specified in the advice are satisfied.

For example, an advice `after(): move() { System.out.println("moved"); }` prints a message immediately after a point of a line is moved.

2.1.1.4 Introduction

An *introduction* in AspectJ introduces new members to classes and therefore changes inheritance relationship between classes. Unlike advices, introduction takes effect at the compilation time. Introduction is a useful construct when we want to add new concern to some existing classes, such as adding timestamp to a data structure, yet we want to have clear separation of these crosscutting concerns.

2.1.1.5 Aspect

In AspectJ, an aspect is declared by a keyword "aspect" and is defined in terms of pointcuts, advice, and introductions. Only aspects may include advice.

2.1.2 Usage Example

2.1.2.1 Logging

Suppose an e-commerce project convention demands all developers to do logging by calling Logger.entry(String) and Logger.exit(String) for all functions they write. This concern crosscuts all classes; such programming convention is also expensive to enforce in large projects and changing logging policy may involve a lot of tedious editing. Without AOP, logging is scattered around the program and cannot be modularized.

For simplicity, the project contains only two classes: the class Logger is responsible for writing a debugging statement to a log file, while the class Main is used to illustrate the logging convention. The program listing is as the following:

```
Package aop;
import java.io.*;

/** Logger usually be accessed using Factory + Singleton pattern */
public class Logger {
    private static final boolean DEBUG = true;
    private static BufferedWriter out = null;
    private static final String logfile = "log.txt";

    public Logger() {  }

    /** Write to the log a record about entering a function */
    public static void entry(String func) {
        try {
            if (null == out) {
                out = new BufferedWriter (new FileWriter(logfile));
            }
            out.write("N:"+ func);
            out.newLine();
            out.flush();
        } catch (Exception exc) {
            exc.printStackTrace();
        } finally {
            if (DEBUG) System.out.println("N:"+ func);
        }
    }

    /** Write to the log a record about exiting a function */
    public static void exit(String func) {
        try {
            if (null == out) {
                out = new BufferedWriter (new FileWriter(logfile));
            }
            out.write("X:"+ func);
            out.newLine();
            out.flush();
        } catch (Exception exc) {
            exc.printStackTrace();
        } finally {
```

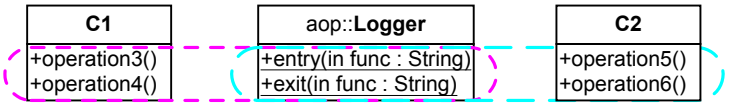
```
            if (DEBUG) System.out.println("X:"+ func);
        }
    }
}
```

```
package aop;

/** Main, without AOP support */
public class Main {
    public Main() {   }
    public void foo() {
        Logger.entry("foo()");
        System.out.println("foo1");
        Logger.exit("foo()");
    }
    public void foo(int i) {
        Logger.entry("foo(int)");
        System.out.println(i++);
        Logger.exit("foo(int)");
    }
    public double bar(double x, double y) {
        Logger.entry("bar(double, double)");
        return x * y;
        Logger.exit("bar(double, double)");
    }
    public static void main(String[] args) {
        Logger.entry("main(String[])");
        Main main1 = new Main();
        main1.foo();
        System.out.println(main1.bar(1.2, 1.3));
        main1.foo(10);
        Logger.exit("main(String[])");
    }
}
```

Notice that the developer of Main has forgotten to also specify the return value and the package of each function being logged, and the statement "Logger.exit(…)" in Main.bar is never be reached. Without code review, this mistake may not be caught until the time the developers consult the log file for debugging, but find the logged records not useful for tracing the logic flow.

In reality, a large e-commerce project should involve hundreds of classes tangling with Logger in a way similar to Main, causing unrelated concerns crosscutting. Suppose an e-commerce project contains at least 30 classes (similar to the scale of an e-commerce project I have participated), and each class has at least 5 methods. Then the number of logging statements are at least 30×5×2 = 300. It will be more tedious if logging statements are to be inserted during testing phase (which I did in the project).



• Figure 3: In a large project, logging will crosscut many concerns, resulting in high coupling but low cohesion.

Using AspectJ, the logging calls can be replaced with a single aspect that automatically logs both parameters and return values types along with method entries and exits. The following is the aspect written in AspectJ for automatic logging:

```
public aspect AutoLog {
    pointcut publicMethods(): execution(public * *..*(..));
    pointcut logObjectCalls(): execution(* aop.Logger.*(..));
    pointcut loggableCalls(): publicMethods() && (!logObjectCalls());
    before(): loggableCalls() {
        aop.Logger.entry(thisJoinPoint.getSignature().toString());
    }
    after(): loggableCalls() {
        aop.Logger.exit(thisJoinPoint.getSignature().toString());
    }
}
```

The first pointcut in the aspect selects all public methods execution join points. The second pointcut selects only methods executions at Logger. The third pointcut defines a loggable call as all public method executions except those of Logger, because including Logger will cause infinite recursion. The advice calls the appropriate Logger functions to do logging for all loggable calls. The statement, `thisJoinPoint,getSignature()`, is an AspectJ special reflective object that returns the run-time context of the join point.

With the aspect AutoLog, Main and other classes no longer need to explicitly couple with Logger:

```
package aop;

/** Main, with AOP support, no longer needs to couple with Logger */
public class Main {
    public Main() {  }
    public void foo() {
        System.out.println("foo1");
    }
    public void foo(int i) {
        System.out.println(i++);
    }
    public double bar(double x, double y) { return x * y; }
    public static void main(String[] args) {
        Main main1 = new Main();
        main1.foo();
        System.out.println(main1.bar(1.2, 1.3));
        main1.foo(10);
    }
}
```

The aspects must be woven with the codes they modify before the aspects can affect the Java class. To weave aspects with Java codes, all the Java source codes and AspectJ codes must be compiled at the same time with the AspectJ compiler ajc. The command for weaving and compiling the logging example is like: `ajc -classpath c:\aspectj1.0\lib\aspectjrt.jar @aop.lst`, where aop.lst is a text file listing the source codes to be compiled. The AspectJ compiler can generate either Java classes or Java codes, which must then be compiled or executed with the AspectJ runtime JAR. Thus, to execute the logging example, just enter: `java -cp`

`c:\aspectj1.0\lib\aspectjrt.jar; aop.Main`. The log file after running the example is as follows:

```
N:void aop.Main.main(String[])
N:void aop.Main.foo()
X:void aop.Main.foo()
N:double aop.Main.bar(double, double)
X:double aop.Main.bar(double, double)
N:void aop.Main.foo(int)
X:void aop.Main.foo(int)
X:void aop.Main.main(String[])
```

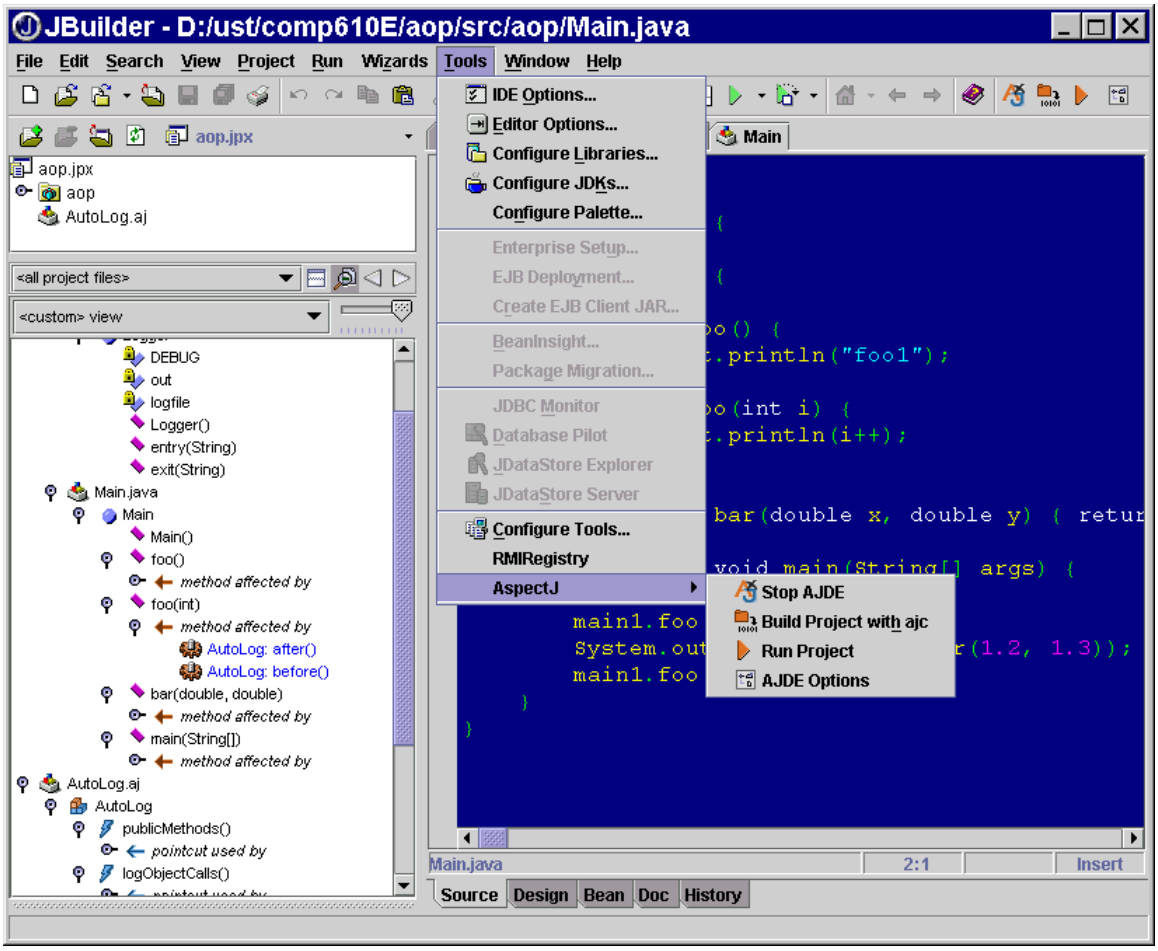
We can also ask the AspectJ compiler to weave the codes together into Java source codes only. The woven Main will look like the following:

```
/*   Generated by AspectJ version 1.0.4 */
package aop;
import AutoLog;
public class Main {
  static org.aspectj.runtime.reflect.Factory ajc$JPF;
  private static org.aspectj.lang.JoinPoint.StaticPart foo$ajcjp1;
  private static org.aspectj.lang.JoinPoint.StaticPart foo$ajcjp2;
  private static org.aspectj.lang.JoinPoint.StaticPart bar$ajcjp3;
  private static org.aspectj.lang.JoinPoint.StaticPart main$ajcjp4;
  public Main() {
    super();
  }
  public void foo() {
    try {
      AutoLog.aspectInstance.before0$ajc(Main.foo$ajcjp1);
      System.out.println("foo1");
    } finally {
      AutoLog.aspectInstance.after0$ajc(Main.foo$ajcjp1);
    }
  }
  public void foo(int i) {
    try {
      AutoLog.aspectInstance.before0$ajc(Main.foo$ajcjp2);
      i = i + 1;
    } finally {
      AutoLog.aspectInstance.after0$ajc(Main.foo$ajcjp2);
    }
  }
  public double bar(double x, double y) {
    try {
      AutoLog.aspectInstance.before0$ajc(Main.bar$ajcjp3);
      return x * y;
    } finally {
      AutoLog.aspectInstance.after0$ajc(Main.bar$ajcjp3);
    }
  }
  public static void main(String[] args) {
    try {
```

```
      AutoLog.aspectInstance.before0$ajc(Main.main$ajcjp4);
      Main main1 = new Main();
      main1.foo();
      double z = main1.bar(1.2d, 1.3d);
      main1.foo(10);
    } finally {
      AutoLog.aspectInstance.after0$ajc(Main.main$ajcjp4);
    }
  }
  static {
    Main.ajc$JPF = new org.aspectj.runtime.reflect.Factory("Main.java", Main.class);
    Main.foo$ajcjp1 = Main.ajc$JPF.makeSJP("method-execution",
        Main.ajc$JPF.makeMethodSig("1-foo-aop.Main----void-"), 7, 5);
    Main.foo$ajcjp2 = Main.ajc$JPF.makeSJP("method-execution",
        Main.ajc$JPF.makeMethodSig("1-foo-aop.Main-int:-i:--void-"), 10, 5);
    Main.bar$ajcjp3 = Main.ajc$JPF.makeSJP("method-execution",
        Main.ajc$JPF.makeMethodSig("1-bar-aop.Main-double:double:-x:y:--double-"), 14, 5);
    Main.main$ajcjp4 = Main.ajc$JPF.makeSJP("method-execution",
        Main.ajc$JPF.makeMethodSig("9-main-aop.Main-[Ljava.lang.String;:-args:--void-"), 16,
5);
  }

}
```

Alternatively, the aspects can be woven and compiled using AspectJ Development Environment for JBuilder or Forte. The AJDE provides a graphical browser for navigating the relationships between aspects and the base codes. Figure 4 is a screenshot of JBuilder with AJDE. The lower-left panel is the aspect browser; the AspectJ compiler can be invoked from the GUI.



• Figure 4: JBuilder with AspectJ Development Environment.

3 AOP Issues

3.1 Does AOP Work?

Although AOP looks promising for separating crosscutting concerns, how useful is it software development practitioners? It is resource-infeasible for software development organizations to evaluate the usefulness of AOP by building software both with and without AOP and compare the results. Therefore, some researchers have conducted experiments and case studies for evaluating the usefulness of AOP.

[MWBRLK2001] has assessed the usefulness of AOP by conducting two semi-controlled experiments and two case studies. One experiment considered whether AspectJ improves developers' ability to locate and fix faults in a multi-threaded program. Another experiment investigated the ease of changing a distributed system. Each experiment had three trials, run by two groups: one with AspectJ, and the other with a control language. One case study explored designing and programming with aspects by implementing a web-based learning environment. Another case study attempted to separate concerns in two existing programs using AspectJ and Hyper/J. The conclusions of the assessments are:

- The scope of effect of an aspect affects the ability of reasoning the codes. It is easier for participants to understand (and debug) an aspect or a class if the effect of the aspect is on the discernible parts of a system.

- AOP changed how participants tackled the tasks. The AOP group first searched for a solution that could be modularized in an aspect. It may thus take longer time to find a solution if the solution could not be encapsulated within an aspect.

- It is easier to build and debug AOP-based programs if the interface is narrow (i.e. an aspect has well-defined effect on particular points in a code), and the reference from the aspect to the base code is unidirectional.

- It is easier to understand and manage an aspect if it forms the glue between two OO structures.

- Refactor concerns in existing systems into aspects requires restructuring of the base code to expose join points. Tools that assist codes restructuring would make it easier to introduce aspects into existing systems.

- AOP shows promising. However, more studies are required to know how, when, what project AOP are beneficial.

Another group [PC2001] assessed the usefulness of AOP by conducting a case study of developing a temperature control system (TCS). Four major aspects and their relationships in the TCS were identified: mathematical modeling, real-world mapping, scheduling, and synchronization. Mathematical modeling should be kept separated from the real-world mapping aspect to avoid tangling problems. Moreover, scheduling works together with synchronization which also interacts with real-world mapping. In the study, one group developed the TCS using OO approach, while another group developed the system with AspectJ. The conclusions of the assessments are:

- The separation provided by AOP is most helpful when the interface is narrow. This is same as the result found by [MWBRLK2001].

- Most of the design effort was on locating the interface between relevant concerns and the base code. In contrast, there was no increase in implementation effort. This suggests researchers need to learn more about how to understand a system in term of aspects.

- AOP helps maintenance by controlling change.

- AOP has no significant disadvantages in performance.

- Good separation of concerns must be enforced by architectural means. The mechanisms provided by AOP cannot replace good design.

- It is easier to write and change certain types of concerns; but others have to be matched with the aspect constructs.

4 Conclusion

Aspect-oriented programming introduces a new style of decomposition and is a promising way of separating crosscutting concerns that are usually hard to do in object-oriented programming. As discussed in previous sections, AOP is particularly useful in separating concerns that are unidirectional and have well-defined effect, such as debugging and tracing concerns (to which I will apply AOP in future projects). However, the current AOP technology is still in research stage. Currently, three AOP research areas are important: 1) Ease of use. Techniques on how to understand a system in term of aspects should be well documented. Guidelines and design principles, such as UML for AOP, AOP design patterns, must also be established. They help reduce the cost for software engineers to adapt to this new paradigm. 2) Studies and experiment that prove the usefulness of AOP by measurable benefits. This encourages the computer industry giants and influential committees to endorse the technology. 3) Development tools for discovering aspects, restructuring existing codes, recutting, etc. AOP technology will only be adapted by the software industry. Those three factors are significant to determine whether if the software industry will eventually adapt AOP technology.

5 References

[AJFaq2002] PARC Inc, "Frequently Asked Questions about AspectJ", Feb 8, 2002. URL: http://aspectj.org/doc/dist/faq.html

[AJPg2002] Xerox Corporation, "The AspectJ Programming Guide", 2002. URL: http://aspectj.org/doc/dist/progguide/index.html.

[AJTut2001] Bill Griswold, Erik Hilsdale, Jim Huhunin, Wes Isberg, Gregor Kiczales, Mik Kersten, "Aspect-Oriented Programming with AspectJ", 2001. URL: http://aspectj.org/doc/dist/tutorial.pdf

[CKFHO2001] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong, "Structuring Operating System Aspects", *Communications of the ACM*, vol. 44, no. 10, pp 79-82, October 2001.

[Czar1998] K. Czarnecki. "Chapter 7: Aspect-Oriented Decomposition and Composition", Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Ph.D. thesis, in preparation, Technische Universität Ilmenau, Germany, 1998.

[DC1997] John Dempsey and Vinny Cahill, "Aspects of System Support for Distributed Computing", *ECOOP '97 Workshop on Aspect-Oriented Programming*.

[EAKLO2001] Tzilla Elrad, Mehmet Aksits, Gregor Kiczales, Karl Lieberherr, and Harold Ossher, "Discussing Aspects of AOP", *Communications of the ACM*, vol. 44, no. 10, pp 33-38, October 2001.

[EFB2001] Tzilla Elrad, Robert E. Filman, and Atef Bader, "Aspect-Oriented Programming", *Communications of the ACM*, vol. 44, no. 10, pp 28-32, October 2001.

[GBNT2001] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling", *Communications of the ACM*, vol. 44, no. 10, pp 87-93, October 2001.

[GILLLMM1996] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videria Lopes, Chris Maeda, and Anurag Mendhekar, "Aspect-Oriented Programming", *ACM Computing Surveys*, December 1996.

[IBM2002] Nicholas Lesiecki, "Improve Modularity with Aspect-oriented Programming", *IBM developerWorks: Java technology*, January 2002. URL: http://www-106.ibm.com/developerworks/library /j-aspectj/index.html

[KHHKPG2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, "Getting Started with AspectJ", *Communications of the ACM*, vol. 44, no. 10, pp 59-65, October 2001.

[MWBRLK2001] Gail C. Murphy, Robert J. Walker, Elisa L.A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten, "Does Aspect-Oriented Programming Work?", *Communications of the ACM*, vol. 44, no. 10, pp 75-77, October 2001.

[OT2001] Harold Ossher and Peri Tarr, "Using Multidimensional Separation of Concerns to (Re)shape Evolving Software", *Communications of the ACM*, vol. 44, no. 10, pp 43-50, October 2001.

[PC2001] J. Andrés Díaz Pace and Marcelo R. Campo, "Analyzing the Role of Aspects in Software Design", *Communications of the ACM*, vol. 44, no. 10, pp 67-73, October 2001.

[PKS1999] E. Pulvermüller, H. Klaeren, and A. Speck, "Aspects in Distributed Environments". *Generative and Component-Based Software Engineering*, First International Symposium, Erfurt, Germany, September 1999.