

Coding Issues in AspectJ

Brian de Alwis, Stephan Gudmundson, Greg Smolyn, and Gregor Kiczales

{*bsd, stephang, smolyn, gregor*}@cs.ubc.ca

Department of Computer Science

University of British Columbia

Vancouver, BC, Canada

1 Introduction

AOP [1] has been proposed as a way to write programs that elegantly implement crosscutting concerns. This paper presents initial results of an exploration into whether this is true for AspectJ [2]. During this this exploration we have written several small and medium scale systems using AspectJ.

Writing elegant code requires good implementation structure. This paper presents some of the more interesting coding style decisions we encountered. We present different approaches to implementing several crosscutting concerns and comment on the relative strengths and weaknesses of each approach.

The purpose of the paper is to share our practical experience of using an AOP language to give workshop participants a sense of what it is like to use AOP and in particular the design issues one faces when trying to write elegant code with AspectJ. We offer this information to support a well-grounded discussion of the strengths and weaknesses of AOP, and the challenges facing AOP researchers.

The paper assumes that the reader can read Java programs, and has basic familiarity with AOP.

1.1 A Brief Overview of AspectJ

AspectJ extends the Java programming language with constructs for expressing crosscutting concerns. *Join points* identify points in the execution of the system, which can be composed into a *pointcut*. A crosscutting concern is modularized as an *aspect*, which may be considered an extension of the standard Java class. An aspect may attach *advice* at a pointcut: this is a code fragment executed whenever execution encounters one of the identified join points. Pointcuts and advice can be parameterized to present data accessible at the join points. Other language features are described as necessary.

2 System Description

This paper takes its examples from OAF (Object/Aspect FTPd), an FTP server (File Transfer Protocol [3]) we wrote in AspectJ. OAF is a rewrite of the freely-available jFTPd server [4]. The code is approximately 17300 lines, with 7700 lines of comments.

2.1 The File Transfer Protocol

FTP is an Internet standard for remotely accessing and managing files. The computer hosting the files is the FTP server, and the user's computer is the FTP client. It is an entirely client-driven protocol: clients issue commands in a specialized command-line language, and receive specially-formatted messages in response. Client and server communicate using standard TCP/IP connections. Commands and their responses are sent on a special *control* connection, which remains open for the life-time of the session. Short-lived *data* connections are used to transfer file contents and directory listings.

Before file access is permitted, the user must identify herself to the server, done with the `USER` (username) and `PASS` (password) commands. This information is typically checked against the operating system's account information to verify the user's identity. FTP servers often allow public access via the special user names "anonymous" or "ftp", with the password being, by convention, the user's e-mail address.

Because data transfers may be large, the protocol provides for the client to abort a data-transfer in progress. Aborts are initiated by sending a special Telnet alert sequence on the control channel followed by an `ABOR` command. Servers must therefore monitor the control socket whilst performing data transfers.

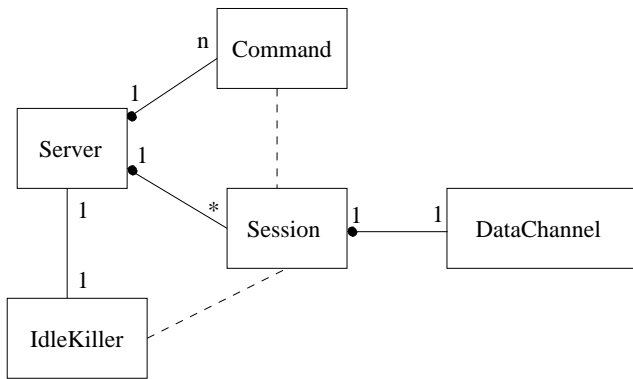


Figure 1: Simple architecture of OAF

2.2 OAF

OAF has a simple object structure (figure 1). A *Server* instance manages multiple *Sessions*, each of which represents one client control connection. *Session* retrieves command lines from clients, which is given to an the appropriate *Command* instance for processing; every FTP command is represented by an implementation of *Command*. Data is transferred using appropriate types of *DataChannels*. An *IdleKiller* works in conjunction with the *Server* to terminate idle sessions. *Server*, *Session* and *IdleKiller* are all *active* objects, meaning that each has a running thread.

3 Style and Modularity Decisions

This section presents some of the most interesting style and modularity we had to make during development. Each is illustrated with one or more aspects, and alternatives discussed.

3.1 Naming Matters

A primary stylistic issue has to do with naming of pointcuts and the implementation of advice body.

3.1.1 Naming of Pointcuts

During the early development of OAF, we typically named our pointcuts to reflect how they were used within the aspect. But we found that this naming convention did not help make the code more readable.

This is illustrated by OAF’s *AbortChecking* aspect. Because data transfers may be large, the FTP protocol allows the client to abort a data transfer in progress by special signalling on the control connection. We added a method to

Session, *checkAbort()*, which does such checking; it throws a special exception when an abort is detected.

AbortChecking defines a pointcut that denotes points at which abort checking should happen, and before advice that calls *checkAbort()* at those points. Consider these two variants of the advice, written using different names for the pointcut:

```

before(): requiringChecks() {
    session.checkAbort();
}
  
```

This reads as: “before doing something that requires abort checking, check for an abort”. This is indirect: determining what needs abort checking requires consulting the pointcut’s definition.

Contrast this with:

```

before(): lowLevelDataOperations() {
    session.checkAbort();
}
  
```

This reads as: “before performing any low-level data operation, check the session for an abort request”. With this naming of the pointcut, the advice communicates more: it describes, at a high level, what sorts of operations require abort checking. The details of what *are* data operations and *how* to check for abort requests are hidden at a lower level.

3.1.2 Advice Bodies

A similar issue has to do with how to code advice bodies. Working within the same example, compare these two variants of the before advice:

```

before(): lowLevelDataOperations() {
    if (!session.commandSocketReader.ready()) { return; }
    if (session.checkAvailableByte(IAC) &&
        session.checkAvailableByte(IP) &&
        session.checkAvailableByte(Sync) &&
        session.checkAvailableByte('A') &&
        session.checkAvailableByte('B') &&
        session.checkAvailableByte('O') &&
        session.checkAvailableByte('R') &&
        session.checkAvailableByte('\r') &&
        session.checkAvailableByte('\n')) {
        throw new AbortError("operation aborted");
    }
}

before(): lowLevelDataOperations() {
    session.checkAbort();
}
  
```

In the second case, *session.checkAbort()* is a helper method that contains the implementation from the body of the first advice. We find this coding style more clear, because it allows us to read the advice declaration as bridging between a high-level description of the join points—in terms of what is happening at those join points—and a high-level description of the action the aspect wants to take at those join points.

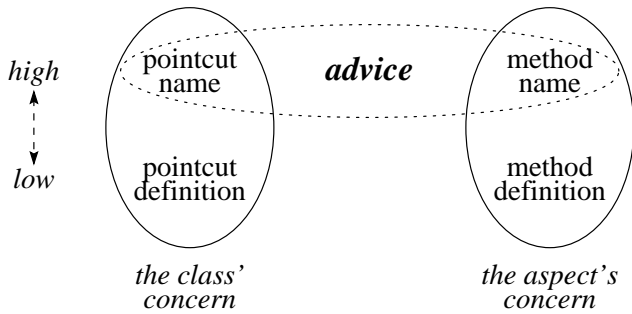


Figure 2: The advice body bridges the aspect and the object

3.1.3 High-Level Bridging

In this coding style, implementation of an aspect can be partitioned into four distinct elements:

1. The pointcut name identifies, at a high-level, join points of interest to the aspect. It does so in terms of what the classes are doing at those points.
2. The pointcut definition (not shown) provides a concrete description of the specific join points that comprise the pointcut.
3. The advice body describes, at a high-level, the behaviour that applies at the join points. It does so in terms of what the methods should do at those points.
4. The method(s) called from the advice provide a concrete implementation of the that behaviour.

Aspects thus have both high-level and low-level features: the pointcut names and advice are high-level while the pointcut definitions and implementations of the methods called from the advice are low-level. The advice body acts as a bridge between the high-levels: describing *what* should happen using suitably-named methods, but deferring *how* they should happen to the method implementations (figure 2).

We adopted this philosophy of naming part-way through implementation, and feel that it has improved the readability of the code.

3.2 Defining Pointcuts

In addition to the naming issues, we discovered that there are different styles of defining pointcuts, with advantages and disadvantages to each. Many of the examples of pointcuts in AspectJ select join points by their Java language-level properties. This may not match the application semantic properties.

The *AbortChecking* aspect (mentioned in the previous section) declared a pointcut named *lowLevelDataOperations*. Our first definition explicitly enumerated all the applicable join points completely within the aspect. We call this a ‘centralized’ definition.

```
pointcut lowLevelDataOperations():
    (instanceof(AsciiDataChannel) &&
     (receptions(String readLine(..)) ||
      receptions(void writeLine(..)))) ||
    (instanceof(BinaryDataChannel) &&
     (receptions(long read(..)) ||
      receptions(void write(..)))) ||
    (instanceof(ListCommand) &&
     receptions(void writeFileInfo(..)));
```

The relevant operations are all identified in this one place, and thus are immediately apparent. But the reasons for choosing these methods are not so clear: are there other operations on these classes that might be relevant? Or are they deliberately excluded from this list? From a maintenance perspective the question is whether new operations on these classes should be added to the list? Will the implementor remember to do so?

We tried improving on this by decomposing the pointcut on a per-class basis. In this style, the pointcut is composed of separate pointcuts defined for each relevant class. We call this a ‘decomposed’ definition.

```
pointcut lowLevelDataOperations():
    asciiDataChannelDataOps() ||
    binaryDataChannelDataOps() ||
    listCommandDataOps();

pointcut asciiDataChannelDataOps():
    instanceof(AsciiDataChannel) &&
    (receptions(String readLine(..)) ||
     receptions(void writeLine(..)));

pointcut binaryDataChannelDataOps():
    instanceof(BinaryDataChannel) &&
    (receptions(long read(..)) ||
     receptions(void write(..)));

pointcut listCommandDataOps():
    instanceof(ListCommand) &&
    receptions(void writeFileInfo(..));
```

This style clarifies that the pointcuts identify the data-operations of each class. But the maintenance questions remain – there is still a concern that a programmer adding a new method to one of the classes might forget to update one of these pointcuts.

There is some tension in the decomposed style about *where* the pointcut should “live”. Some maintenance problems might be solved by “pushing” the pointcut definitions onto the classes themselves. This effectively decouples the aspect from the pointcut definition: the aspect has little knowledge of the class implementation. This is akin to Parnas’ information hiding principle [5]. We call this a ‘decoupled’ definition.

```
public class AsciiDataChannel extends
    AbstractDataChannel {
    pointcut dataOperations():
        instanceof(AsciiDataChannel) &&
```

```

        (receptions(String readLine(...)) ||
         receptions(void writeLine(...)));
    // ...
}

public aspect AbortChecking ... {
    pointcut lowLevelDataOperations():
        AsciiDataChannel.dataOperations() ||
        BinaryDataChannel.dataOperations() ||
        ListCommand.dataOperations()
    // ...
}

```

But now the definition of *lowLevelDataOperations* is obscured: it is no longer possible to see immediately where the advice applies. But it may answer some of the maintenance questions.

Once we discovered this decoupled style, we began to see it elsewhere. Both the *PropertyPassing* and *SessionContext* aspects independently — and incorrectly — defined pointcuts to identify the top-level public operations on *Session*. We decided to merge and move the common pointcut onto *Session*, naming it *availableOperations*, and then used it from the aspects. This seemed appropriate because it meant that the class was the source of knowledge of its provided operations. It also seemed to be effective reuse of pointcuts.

Despite the advantages of decoupled pointcuts, we did not then rush and move all pointcuts onto the various classes: each style has its use and place. We discuss this further in the next section.

This discussion reinforces the importance of having well-named pointcuts (section 3.1). *dataOperations* is clear when defined on the *DataChannel* classes due to the context implied from the class: these are the operations performing the data operations on this class. The same pointcut must be further qualified to *lowLevelDataOperations* within *AbortChecking* as it does not have the same implied context.

3.3 Locating Pointcuts

Defining the per-class *dataOperations* pointcuts poses an interesting question: where should pointcuts be defined? ¹ In the aspect using the pointcut? The classes identified in the pointcut? Or an entirely separate aspect? Each seems to have its place: OAF has examples of each.

SessionTermination is a tightly-coupled aspect of the *Session* class, dealing with the specifics of terminating a running session. It has a centralized pointcut *socketSetter* identifying the join points setting the control socket within *Session* — entirely appropriate, given their degree of coupling.

```

aspect SessionTermination of
    eachobject(instanceof(Session)) {
    pointcut socketSetter(Socket s):

```

¹We differentiate between where a pointcut is *declared*, meaning named, and *defined*, where the terminal join points are identified.

```

instanceof(Session) &&
receptions(void setSocket(s));
// ...
}

```

We have already seen *AbortChecking*'s *lowLevelDataOperations* pointcut from section 3.2.

```

public aspect AbortChecking ... {
    pointcut lowLevelDataOperations():
        AsciiDataChannel.dataOperations() ||
        BinaryDataChannel.dataOperations() ||
        ListCommand.dataOperations()
    // ...
}

```

The pointcut is declared within the aspect but is defined by composing the particular *DataChannel* implementations.

Section 3.2 also very briefly described the pointcut *Session.availableOperations*, whose purpose is to identify high-level session operations. It is unusual as it is declared and defined within the source code for *Session* itself, rather than in any particular aspect. It is used by aspects, such as:

```

public aspect SessionContext of
    eachcflow(requiringContext(Session)) {
    pointcut requiringContext(Session session):
        Session.availableOperations(session);
    // ...
}

public aspect PropertyPassing of
    eachcflow(rootExecContexts()) {
    pointcut rootExecContexts():
        rootExecContextInSession(Session) || ...;
    pointcut
        rootExecContextInSession(Session session):
            Session.availableOperations(session);
    // ...
}

```

Note that both “alias”, or redeclare, the pointcut within the aspect, should a subspect need to override it.

Our last example of pointcut-location deals with the pointcuts defined in the aspect *Events*, part of our (failed) attempt to transform events into aspects. *Events* is simply a container for these pointcuts, and has no other function. The pointcuts identify the high-level FTP session operations (client-connect, file transfer completed, etc.), and are available for any to latch onto. There is one significant downside to this approach: as code must refer to the *Events* pointcuts explicitly (i.e. *Events.login()*) by name, they cannot be overridden, and thus cannot be extended.

3.4 Locating and Invoking Methods

Next, we look at a related style issue: method code placement. Our aspects' advice use methods which may “live” in many possible places. Some may be better being on the aspect, while others may belong on the class.

In considering method placement, a primary distinction is the calling interface to be presented. If the method will be called only within the aspect — by advice and/or other methods within the aspect — we call it an *aspect private*

method. If the method will be called from outside the aspect we call it an *aspect interface* method.

3.4.1 Locating Aspect Private Methods

For aspect private methods, cohesiveness seems to be a primary issue. We intuit cohesiveness as to how the method *fits* with the other methods. This also depends on the location and “classification” of the state and behaviours required by the method, since the method has privileged access only to the class or aspect in which it is defined.

This was true of *Session.checkAbort()*, mentioned in passing in section 3.1. *checkAbort()*, as an internal method, could be placed in the aspect *AbortChecking*.

```
aspect AbortChecking {
    void checkAbort () {
        if (!session .commandSocketReader .ready ()) {
            return ;
        }
        if (session .checkAvailableByte(IAC) &&
            session .checkAvailableByte(IP) &&
            session .checkAvailableByte(Sync) &&
            session .checkAvailableByte((int)'A') &&
            session .checkAvailableByte((int)'B') &&
            session .checkAvailableByte((int)'O') &&
            session .checkAvailableByte((int)'R') &&
            session .checkAvailableByte((int)'\r') &&
            session .checkAvailableByte((int)'\n')) {
            throw new AbortError ("operation aborted");
        }
    }
}
```

But it requires access to two protected parts of the *Session* normally carefully hidden from view. We felt it was best put on the class *Session*:

```
class Session {
    protected Socket commandSocket ;
    protected BufferedReader commandSocketReader ;

    boolean checkAvailableByte(int ch) {
        // ...
    }
    void checkAbort () {
        if (!commandSocketReader .ready ()) { return ; }
        if (checkAvailableByte(IAC) &&
            checkAvailableByte(IP) &&
            checkAvailableByte(Sync) &&
            checkAvailableByte((int)'A') &&
            checkAvailableByte((int)'B') &&
            checkAvailableByte((int)'O') &&
            checkAvailableByte((int)'R') &&
            checkAvailableByte((int)'\r') &&
            checkAvailableByte((int)'\n')) {
            throw new AbortError ("operation aborted");
        }
    }
}
```

3.4.2 Locating Aspect Interface Methods

The location of aspect interface methods is more heavily influenced by the proposed calling interface for them. Or alternatively, evaluating cohesiveness for interface methods is

different from private methods. We found three styles in our code.

The first decides that the interface should be placed on the actual class, and not the aspect. We found this when separating *Session*'s gentle-termination code into a separate aspect *SessionTermination*. We decided that since termination was intrinsic to a session, it was more natural to have the initiating operation be invoked from the class:

```
session .terminate ();
```

The implementation simply defers to the aspect.

We captured describing the idleness of a *Session* as the *SessionMonitoring* aspect. Viewing it as independent from *Session*, it receives method calls directly, such as:

```
SessionMonitoring .aspectOf(session) .isIdle ();
SessionMonitoring .aspectOf(session) .isBusy ();
```

We found that certain aspects do not lend themselves to either of these styles, *eachflow* aspects in particular. For our *SessionContext* aspect, responsible for tracking the active FTP session, we chose to use:

```
PropertyPassing .getProperty ("oaf.timeout");
```

This was implemented as a static method on the aspect:

```
public aspect PropertyPassing of
    eachflow (rootExecContexts ()) {

    public static String
        getProperty (String propertyName) {
        PropertyPassing context =
            PropertyPassing .aspectOf ();
        return context .getProperty (propertyName);
    }

    public String getProperty (String propertyName) {
        // ...
    }
}
```

This is more than just a short-cut: removing the *aspectOf()* distances the use from the implementation.

3.4.3 Escape Clause: A Compromise Solution

Should any of these factors be at odds, a compromise is available: the method can be introduced from the aspect into the class. This allows the method to access the private state of the class, while keeping the code within the aspect.² This was the actual solution used for *checkAbort()*:

```
public aspect AbortChecking {
    introduction Session {
        public void checkAbort () {
            // do the appropriate checking
            // of the control connection
        }
    }
}
```

²AspectJ's upcoming "privileged" access for aspects would achieve a similar result, with similar consequences.

3.5 Capturing the Intended Use of a Pointcut

In our development we found that we wanted to capture certain kinds of events as pointcuts so that other parts of our system could add advice to them easily. In essence, we wanted to use pointcuts and advice as a simple event mechanism. But we were unable to develop satisfactory solutions to some of these situations.

Consider this method:

```
protected void login(String user, String pass)
    throws BadLoginException {
    // ...
}
```

This performs a login, which succeeds if the provided username and password are acceptable. Now, suppose that we have advice that applies at successful or unsuccessful login attempts. We would then write code something like:

```
pointcut loginAttempt(String user):
    instanceof(Session) &&
    receptions(void login(user, String));

static after(String user) returning():
    loginAttempt(user) {
    log("Login successful for " + user);
}

static after(String user)
    throwing(BadLoginException e):
    loginAttempt(user) {
    log("Login failed for " + user);
}
```

But we'd really like to be able to expose successful and failed login attempts as separate pointcuts. In the above code, the pointcut `loginAttempt` identifies only the overall action (a login attempt), not specifically login success or failure.

This distinction can be captured in the current AspectJ framework as follows:

```
pointcut failedLogin(String user):
    instanceof(ThisAspect) &&
    receptions(void loginFailed(user));

static void loginFailed(String user) {
    // dummy method
}

after(String user)
    throwing(BadLoginException e):
    loginAttempt(user) {
    loginFailed(user);
}
```

The exported pointcut `failedLogin` targets the dummy method within the aspect (`void loginFailed`), which is called by advice on the original pointcut (`loginAttempt`).

AspectJ could support our requirements more cleanly if the declarative power of the existing advice was available to the pointcut mechanism itself. This might allow our code example to be written as follows:

```
pointcut loginAttempt(String user):
    instanceof(Session) &&
    receptions(void login(user, String));
```

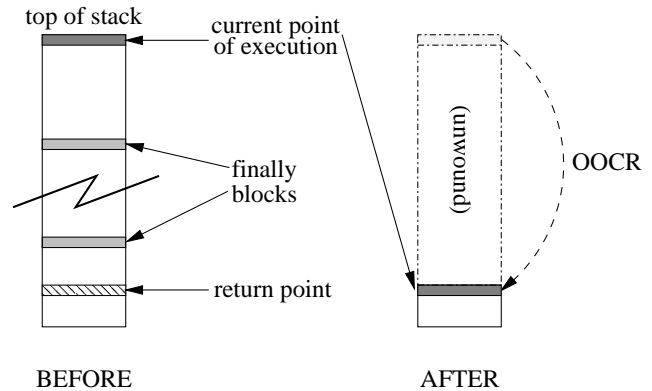


Figure 3: Actions of an Out-Of-Context Return

```
pointcut successfulLogin(String user):
    after returning() loginAttempt(user);

pointcut failedLogin(String user):
    after throwing(BadLoginException);

static after(String user): successfulLogin(user) {
    log("Login successful for " + user);
}

static after(String user): failedLogin(user) {
    log("Login failed for " + user);
}
```

In this code, `successfulLogin` and `failedLogin` are special kinds of pointcuts to which only `after` advice can be added. This represents a significant change to the AspectJ language model and as such would have to be considered carefully.

3.6 Out-Of-Context Returns

We have several aspects that feature a similar design pattern: an *out-of-context return* [6]. All have a condition occurring at some lower level necessitating being able to unwind the stack to an earlier point (figure 3).

We will use `AbortChecking` to demonstrate our approach. First, we must identify where and when abort-sequence-checking is required: this is before performing low-level data operations in the data channels. We must then arrange for the abort check to occur at these times.

```
public aspect AbortChecking {
    pointcut lowLevelDataOperations(): ...;

    static before(): lowLevelDataOperations() {
        SessionContext.getActiveSession().checkAbort();
    }
}
```

Next, we define the abort-sequence-checking method. Should an abort-sequence be detected, the method throws an `AbortError`. This initiates the out-of-context return.

```
introduction Session {
    public void checkAbort() {
        if (abortCondition()) {
```

```

        throw new AbortError ();
    }
}

```

Finally, we provide the higher-level catch blocks; these are the return sites.

```

static around() returns Result:
    instanceof (RetrCommand) &&
    receptions (* doCommand()) {
    try {
        return thisJoinPoint.runNext();
    } catch (AbortError abortError) {
        // ...
        return new FTPResult(
            FTPResult.CLOSING_DATA_CONNECTION,
            "Transfer aborted.");
    }
}

```

This code implements out-of-context returns, but it has two deficiencies. First, it is possible that somewhere between the *throw* and *catch* there may be a handler for *java.lang.Error*. In this case, our code will not function as we expected, and the code that catches the error will receive it unexpectedly. Second, there is no guarantee that the error is thrown in a context that will catch it.

Ideally, the code could look something like this:

```

pointcut abortContext():
    cflow (instanceof (*Command) &&
        receptions (* doCommand (...)));

pointcut lowLevelDataOperations(): ...;

returnpoint abortDetected(int x);

static around() returns Result: abortContext() {
    try {
        thisJoinPoint.runNext();
    } catchreturnpoint (abortDetected()) {
        // as in the previous around advice
    }
}

static before(): lowLevelDataOperations() {
    Session session = SessionContext.getActiveSession();
    if (session.isAbortRequested()) {
        returnpoint abortDetected();
    }
}

introduction Session {
    public boolean isAbortRequested() {
        // Check for an abort sequence
    }
}

```

returnpoint searches up the execution stack for a *try ... catchreturnpoint* construct with a matching *returnpoint* parameter, entirely parallel to Java's *throw*. The *returnpoint* parameter allows multiple out-of-context returns to overlap, and could pass parameters describing the reason for the return in detail.

This proposal solves the problems identified above: nobody else can intercept the out-of-context return and the AspectJ compiler can statically ensure that all *returnpoint* statements occur within a valid *try ... catchreturnpoint*

block. However, more investigation would be required before proposing this as an extension to AspectJ.

4 Summary

This paper covered several style issues and presented the seeds for some further language research. We demonstrated how even simple style decisions such as pointcut names had a sizable impact on the readability and modularity of our code. We also identified two language features that would have been useful in our system. A larger-scale study of these preliminary findings—particularly, one which includes maintenance considerations—may reveal general design guidelines or more solid motivation for these or other new language features.

Acknowledgments

Thanks to UBC's Separation of Concerns discussion group for their constructive comments and encouragement.

References

- [1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.
- [2] X. PARC. Aspectj homepage. <http://aspectj.org>.
- [3] Jon Postel and Joyce Reynolds. File Transfer Protocol (FTP). RFC 959, 1985.
- [4] B. Nenninger. jftpd. <http://www.kreative.net/bwn/jftpd/>.
- [5] D. Parnas. The criteria to be used in decomposing systems into modules. *Communications of the ACM*, pages 1053–1058, December 1972.
- [6] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.