Innovations Syst Softw Eng (2007) 3:281–289
DOI 10.1007/s11334-007-0030-3

ORIGINAL PAPER

Pointcuts, advice, refinements, and collaborations: similarities, differences, and synergies

Sven Apel · Christian Kästner · Martin Kuhlemann · Thomas Leich

Received: 24 May 2007 / Accepted: 24 July 2007 / Published online: 29 August 2007

Abstract Aspect-oriented programming (AOP) is a novel programming paradigm that aims at modularizing complex software. It embraces several mechanisms including (1) pointcuts and advice as well as (2) refinements and collaborations. Though all these mechanisms deal with crosscutting concerns, i.e., a special class of design and implementation problems that challenge traditional programming paradigms, they do so in different ways. In this article we explore their relationship and their impact on modularity, which is an important prerequisite for reliable and maintainable software. Our exploration helps researchers and practitioners to understand their differences and exposes which mechanism is best used for which problem.

1 Introduction

Separation and modularization of concerns is a long-standing goal in software development [12,29]. *Crosscutting concerns* are design and implementation problems that cut across several places in a program. They challenge traditional programming paradigms, e.g., *object-oriented programming* (OOP), because their implementation typically leads to code tangling, scattering, and replication [17]. Modularizing crosscutting concerns requires advanced mechanisms beyond traditional concepts like classes and procedures. Work on *aspect-oriented programming* (AOP) addresses this issue.

In this paper we evaluate and compare two aspect-oriented approaches that provide diverse mechanisms and tools to improve crosscutting modularity. On the one hand, there are *pointcuts and advice* that are grouped typically in *aspects*. We call languages that use mainly these mechanisms to modularize crosscutting concerns *advice and pointcut languages* (ALs). Popular examples of ALs are *AspectJ* [18], *Aspect C++* [31], and *Eos* [30]. On the other hand, there are *class refinements* that are grouped typically in *collaboration modules*. These are used primarily in *collaboration languages* (*CLs*) such as *Classbox/J* [9], *Jiazzi* [25], *Scala* [28], *Jak* [8].

In this article we explore the similarities and differences of ALs and CLs and their impact on crosscutting modularity. Modularity is an important prerequisite for reliable and maintainable software, relevant for organizations that require high standards on software quality like NASA. It is crucial for researchers and, especially, for practitioners in this field to understand the relationship between both paradigms, especially since there is an ongoing confusion on what aspect-oriented languages should offer. Our exploration sheds light on this issue and assists in choosing the right design or implementation mechanism for the right problem, with the goal of a suitable modular structure of software.

2 Crosscutting in a chat application

We start our discussion with a simple chat application. It consists of multiple clients that communicate through a server.

S. Apel (✉)
Department of Informatics and Mathematics,
University of Passau, Passau, Germany
e-mail: apel@uni-passau.de

C. Kästner · M. Kuhlemann
School of Computer Science, University of Magdeburg,
Magdeburg, Germany
e-mail: kaestner@iti.cs.uni-magdeburg.de

M. Kuhlemann
e-mail: kuhlemann@iti.cs.uni-magdeburg.de

T. Leich
Department of Applied Informatics,
Metop Research Institute, Magdeburg, Germany
e-mail: thomas.leich@metop.de

```
 1 public class Client implements Runnable {
 2   public Client(String host, int port) {
 3     history = new ArrayList();
 4     login();
 5   }
 6   public void run() {
 7     while(true)
 8       handleMsg(inputStr.readObject());
 9   }
10   private void handleMsg(Object msg) {
11     if (msg instanceof EncryptedMessage)
12       msg = ((EncryptedMessage)msg).decrypt();
13     history.add(msg);
14     if (msg instanceof TextMessage)
15       receivedTextLine(((TextMessage)msg).content);
16   }
17   public void send(Message msg) {
18     msg = Encrypter.encrypt(msg);
19     try {
20       outputStr.writeObject(msg); outputStr.flush();
21     } catch (IOException ex) {
22       listener.stop();
23     }
24   }
25   private void login() { send(new AuthMessage("user", "pwd")); }
26   /* ... */
27 }
```

```
28 public class Server {
29   private Set connections = new HashSet();
30   private Set authCons = new HashSet();
31   public Server(int port) throws IOException {
32     ServerSocket server = new ServerSocket(port);
33     while (true) {
34       Socket client = server.accept();
35       Connection c = new Connection(s, this);
36       connections.add(c); c.start();
37     }
38   }
39   public void broadcast(Message msg) {
40     for (Connection c : connections)
41       if (authCons.contains(c))
42         c.send(msg);
43   }
```

```
44 // continuing public class Server
45   public void disconnect(Connection c) {
46     authCons.remove(c); connections.remove(c);
47   }
48   public boolean login(Connection c, String u, String p) {
49     boolean res = checkLogin(u, p);
50     if (res) authCons.add(c); return res;
51   }
52   /* ... */
53 }
```

```
54 public class Connection extends Thread {
55   public void run() {
56     handler.broadcast(name + " has joined.");
57     while(true)
58       handleMsg(name, inputStr.readObject());
59   }
60   private void handleMsg(String name, Object msg) {
61     if (msg instanceof EncryptedMessage)
62       msg = ((EncryptedMessage)msg).decrypt();
63     if (msg instanceof TextMessage)
64       server.broadcast(((TextMessage)msg).content);
65     if (msg instanceof AuthMessage) {
66       AuthMessage amsg = (AuthMessage)msg;
67       if (server.login(this, amsg.user, amsg.pwd))
68         server.broadcast(name + " authenticated");
69       else this.send(new TextMessage("denied"));
70     }
71   }
72   public void send(Message msg) {
73     msg = Encrypter.encrypt(msg);
74     synchronized(os) { outputStr.writeObject(msg); }
75   }
76   /* ... */
77 }
```

```
78 abstract class Message implements Serializable {
79   /* ... */
80 }
81 class TextMessage extends Message { /* ... */ }
82 class EncryptedMessage extends Message { /* ... */ }
83 class AuthMessage extends Message { /* ... */ }
```

Fig. 1 Java implementation of a chat application (excerpt)

The server forwards incoming messages to all registered clients. Clients authenticate themselves (concern AUTHENTICATION), all message transfer is encrypted (concern ENCRYPTION), and each client keeps a history of received messages (concern HISTORY). We use this application to illustrate and classify crosscutting concerns.

Our first implementation of the chat application is purely object-oriented and written in Java. Though it could be implemented differently, it serves our need to explain the effects of crosscutting. The implementation consists of the three main classes `Client`, `Server`, and `Connection`. Their listings—shortened for brevity[1]—are shown in Fig. 1. Client and server communicate with a standard Java stream. A client waits for new messages from the stream (Lines 6–9) and interprets them with the method `handleMsg` (Lines 10–16). New messages are sent to the stream with the method `send` (Lines 17–24).[2]

The chat server is implemented similarly to the client, but creates a new thread for each incoming connection (Lines 33–37). This thread receives messages from the stream (Line 58) and passes them to the method `handleMsg` (Lines 60–71), and it sends messages back to the connected client (Lines 72–75). In order to broadcast a received message, `handleMsg` informs the server (Line 64), which forwards the message to all connected clients (Lines 39–43).

This simple example demonstrates the crosscutting nature of some concerns incorporated in our implementation of the chat application. For instance, the implementation of the concerns ENCRYPTION (underlined and red) and HISTORY (**sans serif font** and blue) are tangled within the method `handleMsg` in `Client` (Lines 10–16); the implementation of the concern AUTHENTICATION (*slanted font* and green) is scattered over the classes `Client`, `Server`, and `Connection`; the implementation of the concern ENCRYPTION contains replicated code scattered across the classes `Client` and `Connection`.

Code tangling, scattering, and replication occur typically in the implementation of crosscutting concerns [35,17]. Code scattering refers to the code belonging to one concern scattered across multiple modules; code tangling refers to the code belonging to multiple concerns mixed in one module; code replication refers to multiple code fragments in one program that are equal or similar.

Note that crosscutting concerns and their negative effects on modularity are not a matter of bad or good programming style, but an inherent problem of traditional programming

[1] The complete listings are available at http://wwwiti.cs.uni-magdeburg.de/iti_db/research/chat.

[2] Messages are shown in a GUI, which is not relevant for this article.

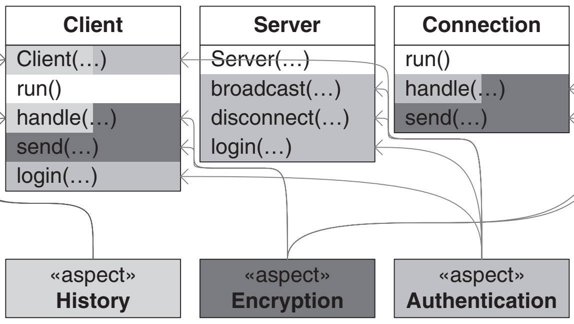

Fig. 2 Implementing the chat application with aspects

```
 1 public aspect Encryption {
 2   pointcut send(Message msg) :
 3     execution(void *.send(Message)) && args(msg);
 4   pointcut receive(Message msg) :
 5     execution(void *.handleMsg(Message))&& args(msg);
 6   void around(Message msg): send(msg){
 7     msg = Encrypter.encrypt(msg); proceed(msg);
 8   }
 9   void around(Message msg): receive(msg){
10     if (msg instanceof EncryptedMessage) {
11       msg = Encrypter.decrypt((EncryptedMessage)msg);
12     }
13     proceed(msg);
14   }
15 }
```

Fig. 3 Implementing ENCRYPTION as an aspect

```
pointcut send(Message msg) :
  (execution(void Server.broadcast(Message) ||
  execution(void *.send(Message)) && args(msg) &&
  !cflow(execution(void Server.broadcast(Message)));
```

Fig. 4 Capturing join points based on the control flow

paradigms such as OOP [35,17]. We could have implemented the chat application differently to avoid some tangling and scattering, but we would observe these problems again in different locations in the source code or in other concerns. ALs and CLs solve this problem with novel language mechanisms. In this article we examine these mechanisms and their properties and discuss them with regard to our initial object-oriented implementation.

3 Advice and pointcut languages

ALs like AspectJ or AspectC++ are popular languages in the AOP community. They aim at modularizing crosscutting concerns to resolve code replication, tangling, and scattering, which occur also in our chat application.

The idea behind ALs is to encapsulate all code associated with one crosscutting concern into a single module, usually called an *aspect*. An aspect specifies several points (a.k.a. *join points*) in the computation of a *base program*, which can be understood as events. Furthermore, an aspect defines what happens when these events occur. Typically, a programmer uses a *pointcut* to specify a set of join points and *advice* to define what code is executed additionally at these points; *inter-type declarations* are not related directly to pointcuts and advice but frequently used together with them. They inject new members into existing classes from within an aspect.

In our chat application we highlighted three crosscutting concerns. AUTHENTICATION crosscuts three classes, ENCRYPTION crosscuts three classes, and HISTORY crosscuts one class but in two different methods. In Fig. 2 we depict these three concerns implemented as aspects. Each aspect encapsulates one concern of the chat application, while the base program is implemented by traditional classes. The arrows denote the places where the aspects extend the base program.[3]

In Fig. 3 we show the feature ENCRYPTION written in AspectJ. The aspect `Encryption` contains the two pointcuts `send` (Lines 2–3) and `receive` (Lines 4–5). They capture the join points associated with the execution of the methods `send` and `handleMsg`. Two pieces of advice execute the actual encryption code when a message is sent (Lines 6–8) and received (Lines 9–14). Both advise two join points each, which is achieved with a wildcard (‘`*`’) for classes in the pointcut expressions.

Additionally, ALs provide several sophisticated language mechanisms for expressing where, when, and how an aspect affects a base program. Notice that in our original source code a message broadcasted by the server is encrypted separately for every individual client. Instead we could already encrypt the message in the server’s `broadcast` method. However, then we would have to encrypt the message that was denied by the server in the class `Connection` somehow differently (Fig. 1, Line 69). Using the pointcut `cflow` available in several ALs, we can implement an alternative solution by modifying the pointcut `send`, as shown in Fig. 4. The pointcut matches either the `broadcast` method, or all `send` methods. To avoid encrypting a method twice, `cflow` excludes all executions of the method `send` that occur in the control flow of the method `broadcast` and that have been encrypted already.

4 Collaboration languages

Much like ALs, CLs are considered aspect-oriented languages. However, they offer alternative mechanisms to modularize crosscutting concerns: (1) a *class refinement* (a.k.a. *refinement*) extends an existing class by new members and declarations and (2) a *collaboration module* (a.k.a. *collaboration*) groups multiple refinements (and further classes), thus modularizing a crosscutting concern.

A refinement is a unit of change that can be applied to a given class. A refinement introduces new members to this class and/or extends existing methods by method overriding

[3] To be precise, the arrows do not refer to the join points but to their projection on static program code (a.k.a. *join point shadows*) [15].

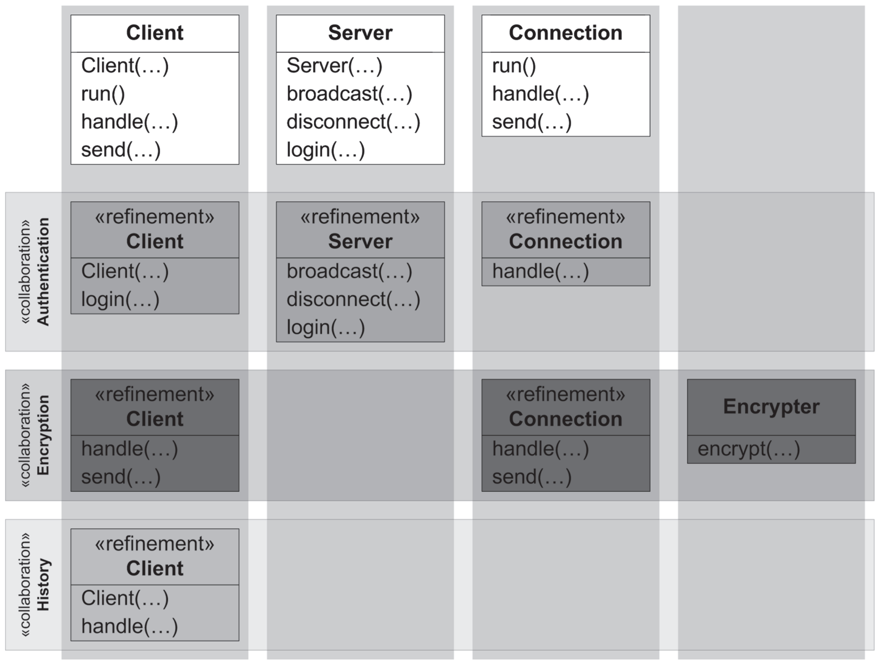

Fig. 5 Implementing the chat application with collaborations

(a.k.a. *method extension*). There are several techniques for implementing class refinements, e.g., *mixins* [10,14], *classboxes* [9], *virtual classes* [23], or *nested inheritance* [27].

For example, in our chat application the concern ENCRYPTION extends two classes at four places and adds a further class `Encrypter`. These changes could be implemented by refinements. In Fig. 5 we depict the collaboration-based design of our chat application. We implemented AUTHENTICATION, ENCRYPTION, and HISTORY each with a collaboration that contains multiple classes and refinements. The base program consists of the three classes `Server`, `Connection`, and `Client`. AUTHENTICATION applies a refinement to each class of the base program, thereby injecting code for client authentication. ENCRYPTION refines `Client` and `Connection` and introduces a new class `Encrypter`. HISTORY applies only a single refinement to `Client`.

In Fig. 6 we show AUTHENTICATION implemented in *Jak* [8], a CL for Java which we picked because of its simplicity. In Jak refinements are declared using the keyword '`refines`'. The refinement of `Client` adds a new method `login` (Lines 5–7) and refines the constructor of `Client` (Lines 2–4). The refinement of `Server` adds a field `authCons` (Lines 10) and a method `login` (Lines 14–18) as well as extends the method `remove` (Lines 11–13). The refinement of `Connection` extends the methods `send` (Lines 21–24) and `handleMsg` (Lines 25–33).[4]

```
 1 refines class Client(){
 2   public Client(String host, int port) {
 3     super(host, port); login();
 4   }
 5   private void login() {
 6     send(new AuthMessage("user", "pwd"));
 7   }
 8 }
```

```
 9 refines class Server {
10   private Set authCons = new HashSet();
11   public void remove(Connection con) {
12     authCons.remove(con); super.remove(con);
13   }
14   public boolean login(Connection con, String user,
15                        String pwd) {
16     boolean res = checkLogin(user, pwd);
17     if (res) authCons.add(con); return res;
18   }
19 }
```

```
20 refines class Connection {
21   public void send(Message msg) {
22     if (server.authCons.contains(this))
23       super.send(msg);
24   }
25   protected void handleMsg(Message msg) {
26     super.handleMsg(msg);
27     if (msg instanceof AuthMessage) {
28       AuthMessage amsg = (AuthMessage) msg;
29       if (server.login(this, amsg.user, amsg.pwd)) {
30         server.broadcast(name + " authenticated");
31       } else { send(new TextMessage("denied")); }
32     }
33   }
34 }
```

Fig. 6 Implementing AUTHENTICATION as a collaboration

In this example we can observe that our crosscutting concerns extend several places of a program and collaborations encapsulate the necessary changes in form of refinements and classes.

[4] Note that in our example the collaborations are not expressed explicitly in the program text but implicitly by storing classes and refinements in directories [8]. The classes and refinements found inside a directory belong to the same collaboration. Alternatively, in Jak, much like in other collaboration languages [7,9], there are keywords that declare which classes and refinements belong to which collaboration.

5 Classification of crosscutting concerns

In order to explore the differences between ALs and CLs and their performance with respect to the modularization of different crosscutting concerns, we classify the nature of crosscutting. First, we distinguish between *homogeneous* and *heterogeneous crosscutting concerns* [11]. Homogeneous crosscutting concerns affect multiple join points and apply one piece of code, i.e., the same extension to all join points. In AOP terminology this is called *quantification* [13]. By contrast, heterogeneous crosscutting concerns apply different pieces of code to different join points. In our initial implementation, ENCRYPTION is a homogeneous crosscutting concern since it extends the methods `send` and `handleMsg` in `Client` and `Connection` by the same piece of code. In contrast, AUTHENTICATION is a heterogeneous crosscutting concern since it extends `Client`, `Server`, and `Connection`, all with different pieces of code (cf. Fig. 1).

A further way to distinguish crosscutting concerns is regarding the point in time at which they affect a program [24]. Static crosscutting concerns affect the static structure of a program by adding new classes and interfaces, by injecting new methods or fields, and by declaring new super-classes and interfaces [26]. Dynamic crosscutting concerns crosscut the dynamic computation of a program and thus can be defined in terms of events and actions. The events are also called *dynamic join points* [36], e.g., invocation of a method, assignment of a field, or throwing of an exception. A defined action is executed when a corresponding event occurs that implements the desired extension to the base program.

In our example, AUTHENTICATION adds a new class and injects three new members, which is a static crosscutting concern. AUTHENTICATION is also in parts a dynamic crosscutting concern since it extends the executions of the methods `handleMsg`, `broadcast`, and `remove`, which can be understood in terms of events and actions.

6 Comparison and programming guidelines

The classification of crosscutting concerns as heterogeneous and homogeneous or as static and dynamic allows us to compare ALs and CLs.

With regard to homogeneous crosscutting concerns ALs perform better than CLs since they provide mechanisms for capturing multiple join points and applying a single extension. For example, the aspect `Encryption` extends two methods in two classes by the same code (Fig. 3). In a collaboration-based solution both classes would be extended by a distinct refinement, which leads to code replication (Fig. 7).

Both ALs and CLs can express heterogeneous crosscutting concerns. For example, AUTHENTICATION is heterogeneous and is implemented with a collaboration of three refinements

```
refines class Client {
  void handleMsg(Message msg) {
    if (msg instanceof EncryptedMessage) {
      msg = Encrypter.decrypt((EncryptedMessage)msg);
    }
    super.handleMsg(msg);
  }
  void send(Message msg) {
    msg = Encrypter.encrypt(msg);
    super.send(msg);
  }
}
```

```
refines class Connection {
  void handleMsg(Message msg) {
    if (msg instanceof EncryptedMessage) {
      msg = Encrypter.decrypt((EncryptedMessage)msg);
    }
    super.handleMsg(msg);
  }
  void send(Message msg) {
    msg = Encrypter.encrypt(msg);
    super.send(msg);
  }
}
```

Fig. 7 Implementing ENCRYPTION as a collaboration

and one class in Fig. 6. In an AL solution one (or more) aspect(s) would bundle a set of inter-type declarations, pointcuts, and pieces of advice, as shown in Fig. 8. Notice that, in the case of heterogeneous crosscutting concerns, there are no advantages regarding code replication. So it seems that both the AL and the CL solution are equivalent. However, especially in larger programs, there is a difference: since an AL solution merges all individual extensions to the target classes arbitrarily and implicitly, the resulting program is difficult to comprehend [32]. The straightforward mapping between classes and refinements in the CL solution facilitates a better program comprehension [32]. Even though arguments regarding comprehension are difficult to prove [6], recent work supports this hypothesis [2,3,16,26,32].

With regard to static and some dynamic crosscutting concerns ALs and CLs perform similarly. They can both be used to inject any kind of member to existing classes, to declare super-classes and interfaces, and to extend existing methods. The difference lies in the constructs used. A refinement encapsulates all new elements to be applied to a class. An aspect injects several new elements into a class by a set of inter-type declarations. A collaboration introduces simply a new class or interface, while an AL solution would require to use static inner classes or external workarounds [22], because otherwise the association between a crosscutting concern and a single aspect would be lost. Furthermore, method extensions are implemented with method overriding in CLs, while the AL solutions use advice, which advises a method's execution. CLs enforce a mapping between classes an their refinements which helps to understand the source code, while ALs allow one to specify all extensions to multiple classes inside a single aspect. Finally, we argue that CL constructs are easier to use because their syntax is usually more concise and close to what is known from OOP. ALs achieve exactly the same

```
privileged aspect Authentication {
  after(Client c) : this(c) && execution(Client.new(String, int) {
    c.login();
  }
  private void Client.login() {
    send(new AuthMessage("user", "pwd"));
  }
  private Set Server.authCons = new HashSet();
  before(Server s, Connection con) : this(s) && args(con) && execution(void Server.remove(Connection)) {
    s.authCons.remove(con);
  }
  public boolean Server.login(Connection con, String user, String pwd) {
    boolean res = checkLogin(user, pwd);
    if (res) authCons.add(con); return res;
  }
  void around(Connection con) : this(con) && execution(void Connection.send(Message)) {
    if (con.server.authCons.contains(con)) proceed(con);
  }
  after(Connection con, Message msg) :
    this(con) && args(msg) && execution(void Connection.handleMsg(Message)) {
    if (msg instanceof AuthMessage) {
     AuthMessage amsg = (AuthMessage) msg;
     if (con.server.login(con, amsg.user, amsg.pwd)) {
       con.server.broadcast(name + " authenticated");
      } else { con.send(new TextMessage("denied")); }
    }
  }
}
```

Fig. 8 Implementing AUTHENTICATION as an aspect

effect, but with a verbose new syntax. This becomes apparent when comparing the two variants to implement AUTHENTICATION in Figs. 6 and 8.

However, ALs provide sophisticated mechanisms to implement dynamic crosscutting that are not available in CLs. For example, several ALs provide mechanisms to implement a crosscutting concern dependently on the runtime control flow. This allows the programmer a view on the dynamic structure of a program. We call those dynamic crosscuts that go beyond method extensions *advanced dynamic crosscuts*. Advanced dynamic crosscuts are supported only by (some) ALs. An example for an advanced dynamic crosscut is the enhanced variant of ENCRYPTION described earlier (Fig. 4), which uses `cflow` to advise code based on the program control flow. CLs do not support such advanced dynamic crosscutting concerns. To implement an equivalent extension with CLs would result in a workaround with additional parameters or methods.

From the above discussion we conclude that AL mechanisms are more suitable for homogeneous and CLs for heterogeneous crosscutting concerns. Furthermore, ALs perform well for concerns that demand a dynamic join point model, while CLs are more suitable for the introduction of new classes and interfaces and for simple dynamic crosscutting concerns that aims only at method extensions.

Due to the complexity and problems introduced by pointcuts, advice, and quantification [1,16,33,34], it is reasonable to use ALs only for what they are really beneficial. Of course, the choice of a programming language depends on many factors, but based on these observations we can assist with a programming guideline regarding the crosscutting nature of the problem to solve: (1) use pointcuts and advice for homogeneous and advanced dynamic crosscutting concerns (2) use refinements and collaborations for heterogeneous crosscutting concerns and method extensions.

This programming guideline reflects the different capabilities of ALs and CLs with respect to the modularization of different kinds of crosscutting concerns. Choosing the right mechanisms is crucial for software modularity, but also for software complexity. The guideline takes the complexity of AL mechanisms into account and suggests to use them only when they are necessary. Applications with high reliability and quality standards require a proper modularity, but they should remain manageable.

7 Aspectual feature modules

Our programming guideline implies that a programmer requires different aspect-oriented mechanisms in the implementation of a program or software system. What follows is that the programmer needs to combine pointcuts, advice, refinements, and collaborations within one programming language or environment. Several languages support such a combination following our guideline, e.g., *CaesarJ* [7] and *FeatureC++* [4]. In this article we use a combination of Jak and AspectJ to illustrate how to implement *aspectual feature modules* (AFMs) [5], an approach which integrates collaborations and aspects.

AFMs follow the concepts of collaborations but additionally integrate aspects beside classes and refinements. In Fig. 9 we show the collaboration-based design of our chat application, but now ENCRYPTION is implemented with an aspect and a class. This example illustrates that an AFM encapsulates a collaboration of classes, refinements, and aspects, which is not possible with either only an AL or only a CL. The aspect is used to implement a homogeneous crosscutting concern, which otherwise leads to code replication (cf. Fig. 7), and a dynamic crosscutting concern, which otherwise requires a workaround (cf. Fig. 4).

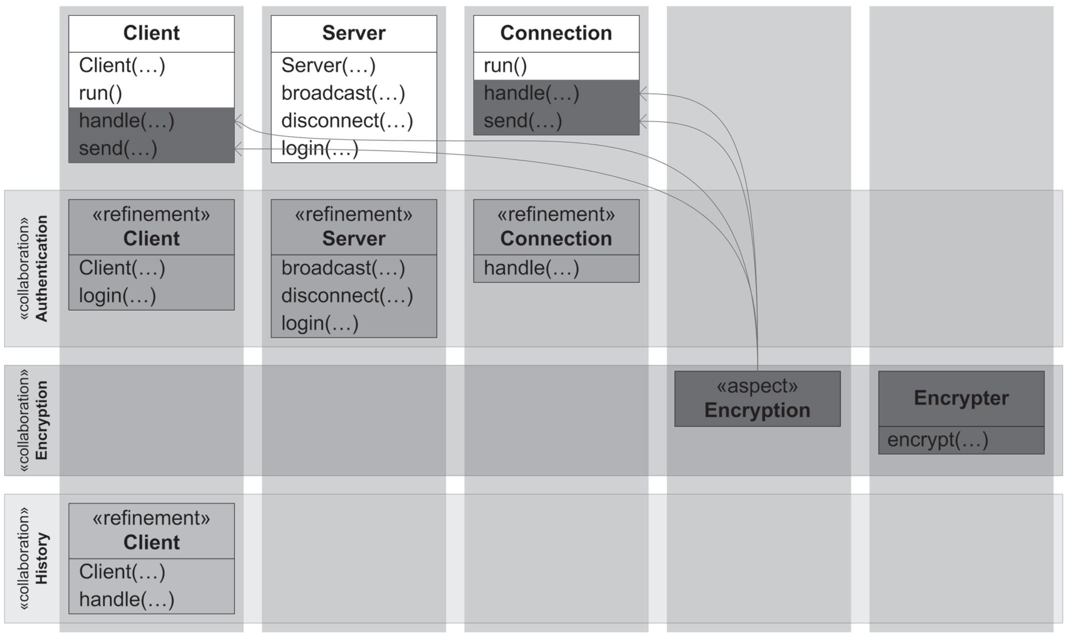

Fig. 9 Implementing the chat application with aspectual feature modules

8 Case studies

In a first case study, we applied the combination of AL und CL mechanisms implemented in AspectJ and Jak to a non-trivial medium-sized software project: a *framework for peer-to-peer overlay networks* (FrameP2P) [3]. It was developed by the first author in the course of a project that aims at variability and customizability in distributed systems and at advanced overlay network features.

The framework implementation consists of 6,426 lines of code (LOC) implementing 113 concerns. Ninety-nine of the 113 concerns were implemented completely with collaborations—without aspects. Fourteen concerns were implemented with a combination of refinements and aspects, including pointcuts and advice. This imbalance comes from our programming guideline, which defines what mechanism is used for which kind of crosscutting concern. We found 14 concerns that are advanced dynamic and homogeneous. The remaining 99 concerns were heterogeneous, so that refinements and collaborations were sufficient. Though we could have implemented the 14 aspects with collaborations as well, this would have resulted in code replication, scattering, and tangling. Also we could have implemented the 99 collaborations with aspects, but this would have destroyed the object-oriented structure of the framework, which would have decreased program comprehension.

We have observed that the code associated with advanced AL mechanisms for homogeneous and advanced dynamic crosscuts sums up to only a minor fraction of the code base of the framework (6% aspects, 46% refinements, 48% classes). At the same time, 17% of the framework's code base are simple method extensions and 77% are pure static crosscuts, i.e., the introduction of new structural elements. This result is remarkable since it shows that, following our programming guideline, pointcuts and advice have only a small impact on the development of the framework for peer-to-peer overlay network. Nevertheless, by using aspects for homogeneous crosscutting concerns, we reduced the code size by 5%, which by itself is a notable result.

Once we had finished our case study and stumbled onto the disproportion between AL and CL mechanisms, we became interested in how other programmers use CLs and ALs. In several studies we and others analyzed to what extend CL and AL mechanism have been used in programs and what their benefits are.

A study of our own revealed that in ten analyzed AspectJ programs (1–130 thousand LOC) only 2% of the code is associated with AL mechanisms and 12% with CL mechanisms [2]. The remaining code (86%) is purely object-oriented. Studies conducted by others who analyzed and compared programs written in ALs and CLs revealed similar proportions [19–21,37].

9 Concluding remarks

In this article we have illustrated that aspect-oriented languages based on advice and pointcuts (ALs) and languages based on refinements and collaborations (CLs) are complementary. Our classification of crosscutting concerns revealed that the strengths of one approach map roughly to the weaknesses of the other approach. We used this complementarity to establish a guideline for programmers: (1) use pointcuts and advice for homogeneous and advanced dynamic crosscutting concerns and (2) use refinements and collaborations for heterogeneous crosscutting concerns. In a case study we showed that a combination of AL and CL mechanisms in form of AFMs along with our guideline are applicable to a non-trivial medium-sized software project. We noticed that AL and CL mechanisms have been used to different extents (6% aspects, 46% refinements, 48% classes), which is also supported by other studies.

We conclude that crosscutting modularity is crucial for the development of maintainable and reliable software, e.g., as for applications in the NASA environment. Our guideline helps developers to choose the right mechanism for a given problem while balancing software modularity and software complexity.

Acknowledgments We would like to thank Don Batory and Christian Lengauer for fruitful comments on earlier drafts of this paper.

References

1. Alexander R (2003) The real costs of aspect-oriented programming. IEEE Softw 20(6):92–93
2. Apel S (2007) The role of features and aspects in software development. PhD thesis, School of Computer Science, University of Magdeburg
3. Apel S, Batory D (2006) When to use features and aspects? A case study. In: Proceedings of the international conference on generative programming and component engineering (GPCE), ACM Press, New York, pp 59–68
4. Apel S, Rosenmüller M, Leich T, Saake G (2005) FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming. In: Proceedings of the international conference on generative programming and component engineering (GPCE), Lecture Notes in Computer Science, vol 3676. Springer, Heidelberg, pp 125–140
5. Apel S, Leich T, Saake G (2006) Aspectual mixin layers: aspects and features in concert. In: Proceedings of the International conference on software engineering (ICSE), ACM Press, New York, pp 122–131
6. Apel S, Kästner C, Trujillo S (2007) On the necessity of empirical studies in the assessment of modularization mechanisms for crosscutting concerns. In: ICSE workshop on assessment of contemporary modularization techniques (ACoM'07)
7. Aracic I, Gasiunas V, Mezini M, Ostermann K (2006) An overview of CaesarJ. Transactions on aspect-oriented software development I. Lect Notes Comput Sci 3880:135–173
8. Batory D, Sarvela JN, Rauschmayer A (2004) Scaling step-wise refinement. IEEE Trans Softw Eng (TSE) 30(6):355–371
9. Bergel A, Ducasse S, Nierstrasz O (2005) Classbox/J: controlling the scope of change in Java. In: Proceedings of the international conference on object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, New York, pp 177–189
10. Bracha G, Cook WR (1990) Mixin-Based Inheritance. In: Proceedings of the international conference on object-oriented programming, systems, languages, and applications (OOPSLA) and the European conference on object-oriented programming (ECOOP), ACM Press, New York, pp 303–311
11. Colyer A, Rashid A, Blair G (2004) On the separation of concerns in program families. Tech Rep COMP-001-2004, Computing Department, Lancaster University
12. Dijkstra EW (1976) A discipline of programming. Prentice Hall, Englewood Cliffs
13. Filman RE, Friedman DP (2005) Aspect-oriented programming is quantification and obliviousness. In: Aspect-oriented software development, Addison-Wesley, Reading, pp 21–35
14. Flatt M, Krishnamurthi S, Felleisen M (1998) Classes and Mixins. In: Proceedings of the international symposium on principles of programming languages (POPL), ACM Press, New York, pp 171–183
15. Hilsdale E, Hugunin J (2004) Advice weaving in AspectJ. In: Proceedings of the international conference on aspect-oriented software development (AOSD), ACM Press, New York, pp 26–35
16. Kästner C, Apel S, Batory D (2007) A case study implementing features using AspectJ. In: Proceedings of the international software product line conference (SPLC)
17. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier JM, Irwin J (1997) Aspect-oriented programming. In: Proceedings of the European conference on object-oriented programming (ECOOP), lecture notes in computer science, vol 1241. Springer, Heidelberg, pp 220–242
18. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG (2001) An overview of AspectJ. In: Proceedings of the European conference on object-oriented programming (ECOOP), lecture notes in computer science, vol 2072. Springer, Heidelberg, pp 327–353
19. Liu J, Batory D, Lengauer C (2006) Feature-oriented refactoring of legacy applications. In: Proceedings of the international conference on software engineering (ICSE), ACM Press, New York, pp 112–121
20. Lopez-Herrejon R (2006) Understanding feature modularity. PhD thesis, Department of Computer Sciences, The University of Texas at Austin
21. Lopez-Herrejon R, Batory D (2006) From crosscutting concerns to product lines: a function composition approach. Tech Rep TR-06-24, Department of Computer Sciences, The University of Texas at Austin
22. Lopez-Herrejon R, Batory D, Cook WR (2005) Evaluating support for features in advanced modularization technologies. In: Proceedings of the European conference on object-oriented programming (ECOOP), lecture notes in computer science, vol 3586. Springer, Heidelberg, pp 169–194
23. Madsen OL, Moller-Pedersen B (1989) Virtual classes: a powerful mechanism in object-oriented programming. In: Proceedings of the international conference on object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, New York, pp 397–406
24. Masuhara H, Kiczales G (2003) Modeling crosscutting in aspect-oriented mechanisms. In: Proceedings of the European conference on object-oriented programming (ECOOP), lecture notes in computer science, vol 2743, Springer, Heidelberg, pp 2–28
25. McDirmid S, Flatt M, Hsieh WC (2001) Jiazzi: New-age components for old-fashioned Java. In: Proceedings of the international conference on object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, New York, pp 211–222
26. Mezini M, Ostermann K (2004) Variability management with feature-oriented programming and aspects. In: Proceedings of the international symposium on foundations of software engineering (FSE), ACM Press, New York, pp 127–136
27. Nystrom N, Chong S, Myers AC (2004) Scalable extensibility via nested inheritance. In: Proceedings of the international conference on object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, New York, pp 99–115

28. Odersky M, Zenger M (2005) Scalable component abstractions. In: Proceedings of the international conference on object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, New York, pp 41–57
29. Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM (CACM) 15(12):1053–1058
30. Rajan H, Sullivan KJ (2005) Classpects: unifying aspect- and object-oriented language design. In: Proceedings of the international conference on software engineering (ICSE), ACM Press, New York, pp 59–68
31. Spinczyk O, Lohmann D, Urban M (2005) AspectC++: an AOP extension for C++. Softw Developer's J, 68–74
32. Steimann F (2005) Domain models are aspect free. In: Proceedings of the international conference on model driven engineering languages and systems (MoDELS/UML), lecture notes in computer science, vol 3713. Springer, Heidelberg, pp 171–185
33. Steimann F (2006) The paradoxical success of aspect-oriented programming. In: Proceedings of the international conference on object-oriented programming, systems, languages, and applications (OOPSLA), ACM Press, New York, pp 481–497
34. Störzer M, Graf J (2005) Using pointcut delta analysis to support evolution of aspect-oriented software. In: Proceedings of the international conference on software maintenance (ICSM), IEEE Computer Society, pp 653–656
35. Tarr P, Ossher H, Harrison W, Stanley M Sutton Jr (1999) N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the international conference on software engineering (ICSE), IEEE Computer Society, pp 107–119
36. Wand M, Kiczales G, Dutchyn C (2004) A semantics for advice and dynamic join points in aspect-oriented programming. ACM Trans Program Lang Syst (TOPLAS) 26(5): 890–910
37. Xin B, McDirmid S, Eide E, Hsieh WC (2004) A comparison of Jiazzi and AspectJ for feature-wise decomposition. Tech Rep UUCS-04-001, School of Computing, The University of Utah