

# Aspect-Oriented Solutions to Feature Interaction Concerns using AspectJ

Lynne BLAIR  
Computing Department,  
Lancaster University, Bailrigg  
Lancaster, LA1 4YR, U.K.  
lb@comp.lancs.ac.uk

Jianxiong PANG  
Computing Department,  
Lancaster University, Bailrigg  
Lancaster, LA1 4YR, U.K.  
j.pang@lancaster.ac.uk

## Abstract

In this paper, we propose a two-level architecture for feature driven software development, consisting of a base layer for a feature's core behaviour and a meta-layer for resolution modules that provide solutions to feature interaction problems. Whilst a standard programming language is used at the base level, e.g. an object-oriented language such as Java, we propose the use of an aspect-oriented programming language for the inherent cross-cutting concerns that exist at the meta-level. We evaluate the use of AspectJ for the implementation of interaction resolution modules at the meta-level. This evaluation is carried out through an in-depth study of an email system. We conclude that aspect-oriented approaches are highly suited for this split-level architecture and that the architecture has many benefits for feature driven software development. Finally, we also highlight a number of problems with AspectJ for our intended use, but discuss how the selection of an alternative aspect-oriented technique would avoid these problems.

*Keywords: feature driven development, aspect-oriented programming, feature interaction and interaction resolution, feature composition.*

## 1. Introduction

Agile methodologies [13][14] are gaining increasing popularity for tackling today's software development challenges. Examples of such methodologies are Extreme Programming, Feature-Driven Development, Adaptive Software Development, and Dynamic Systems Development, ranked in this order of popularity according to the results of a survey reported in [4].

In this paper, we focus on the Feature Driven Development (FDD) approach [7][27]. In this approach the unit of development is a feature: a small piece of client-valued functionality. The granularity of these features is guided by projected development time: if, during decomposition of the software task, a feature looks bigger than two weeks' work then it should be decomposed further.

In addition to being valuable units of development, features can also be seen as valuable units of evolution, for example with respect to system adaptation, re-configuration, versioning and even billing. The telecommunications industry has a tradition of organising development projects, people and marketing by features [32]. Microsoft has also apparently followed this process in their software product line for a number of years [5].

However, as has been witnessed in the telecommunications industry for many years now, whilst the value of feature-oriented approaches is clear, interactions between different parts of a program are an inevitable, and often intended, result of modularisation. Referring back to FDD, unit testing for the features is carefully prescribed as part of the methodology (see table 1), yet cross-feature testing is left rather ad-hoc.

This paper proposes a framework in which Feature Driven Development is enhanced with the power of Aspect-Oriented Software Development (AOSD) techniques [20], the enhancement being provided specifically to deal with the handling of the resolution of interactions. In this work, we use the Java programming language to implement the core functionality of a feature (we refer to this as a feature’s hard logic), but we do not entangle any code that would be specifically added to this feature to allow it to work with other features. Instead, this is kept separate by utilising aspect-oriented programming (AOP) constructs to implement any composition or interaction concerns (we refer to this as a feature’s soft logic). In this paper, we illustrate the implementation of the feature’s soft logic by using the notion of the aspects provided by AspectJ [19].

**Table 1:** FDD Process #5, table taken from [10].

Unit Test	
	Each Class Owner tests their code to ensure that all requirements on their classes for the feature(s) in the work package are satisfied. The Chief Programmer determines what, if any, feature team-level unit testing is required - in other words, what testing across the classes developed for the feature(s) is required.

The structure of the rest of the paper is as follows. We initially give an overview of aspect-oriented software development (section 2), before moving on to describe our two-level architecture (section 3). In section 4, we then present an email system with just three features initially; this is used to illustrate our approach. In section 5, we develop the case study further, to include ten features and a graphical user interface. Rather than describing our approach at this stage, we use this further study to drive our evaluation. This raises a number of significant points about our architecture and the use of AspectJ, and leads us to discussions on other related approaches and plans for our future work. Finally, in section 6, we draw our conclusions.

## 2. Aspect-oriented software development (AOSD)

Over recent years, AOSD has received a lot of attention as a new paradigm to more effectively achieve the separation of concerns alluded to by Parnas [29] and Dijkstra [8]. Whilst a large degree of success has been achieved through advances such as abstract data types and object-oriented programming, such techniques have failed to elegantly capture concerns that cut across modules. AOSD has become the umbrella term relating to all approaches that achieve this explicit separation of cross-cutting concerns, including aspect-oriented programming [18], subject-oriented programming [12] and reflection (in which AOP has some roots) [17].

AspectJ [19] has been developed by the Xerox PARC team over the last few years and, at the moment, is arguably the most dominant technique within the AOSD community. AspectJ provides new language constructs to deal with cross-cutting concerns, and excellent tool support for the subsequent weaving process: the AspectJ tool acts as a pre-processor that weaves programs together to generate (tangled) Java code. Note that other techniques such as JAC (Java Aspect Components) [30] have been proposed as alternatives to AspectJ should weaving be required dynamically at run-time.

In AspectJ, *aspects* are defined in a similar way to Java classes (with attributes and methods as required). These aspects may also define sets of join points (called *pointcuts*) that represent well-defined points in the execution of the program (thus forming the basis for supporting crosscutting concerns). In addition, two further constructs are provided: an *advice* construct that allows the addition of code before, after or around an existing method (or methods), and an *introduce* construct that allows the introduction of a new attribute or method into an existing class (or classes). For further information we refer the reader to [19] or the associated AspectJ web site.

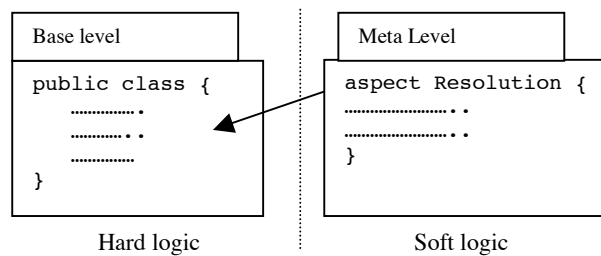
### 3. A two-level architecture for features

In our proposed framework, we assume that every feature has a clear specification of its functionality. Although the implementation of that specification varies, it is generally easy to distinguish the *pure feature code*. We call this the *hard logic* of the feature, i.e. the inevitable part of the implementation of the feature’s functional specification.

However, in feature driven (or feature-oriented) development, features must clearly be capable of working with other features. Since the feature’s hard logic is unable to adapt itself to different execution contexts (different connected features), we also require a corresponding *soft logic* to soften the behaviour, making it flexible enough to adapt to other interacting features. Therefore, a feature’s soft logic is responsible for boundary condition checking and taking action to smooth any incompatibilities.

Since a feature designer cannot foresee the future features that will interact with his/her developed feature, soft logic should be able to be added to the hard feature logic at any stage. To support this kind of addition, the soft logic is ideally raised up to the meta-level so as to provide a separation from the hard logic and facilitate reuse and easier maintenance/ evolution.

It is this soft logic that we believe is ideally suited to aspect oriented software development techniques. The soft logic forms a resolution module that clearly cross-cuts the base-level system representing a feature’s hard logic.



**Figure 1.** A two-level architecture for features

To achieve this separation of concerns in this piece of work, we will use Java to implement a feature’s hard logic and AspectJ to implement the associated soft logic.

### 4. A case study: an email system

As our case study, we wanted to choose a system beyond the area of traditional telephony, but with known interactions that could be extended to more general distributed systems such as Internet-based systems. This led us to the email system studied in [11], where several different email features were studied and a large number of interactions found.

Our intention in this work is, by using the already identified interactions, to show how AOP techniques can be used to provide resolution modules (soft logic) for the problematic features, allowing them to work together correctly. Crucially, as mentioned above, this solution aims to maintain a clean separation between the core feature code (a feature’s hard logic) and the code required to allow it to work correctly with other features (a feature’s soft logic). We have chosen to focus initially on three problematic features, all of which have had interactions identified in [11]:

- **RemailMessage:** This feature forwards an incoming message and replaces a sender’s true ID with a pseudonym. *RemailMessage* can also handle messages sent in the other direction as well, i.e. delivering a message addressed to a pseudonym to the real receiver (by reversing the address mapping).
- **MailHost:** This feature receives incoming messages for the *MailHost*’s users, and delivers the messages to users’ accounts.
- **AutoResponder:** This feature automatically replies to each incoming sender with a prescribed message, but only one reply should be returned per incoming sender.

More features will be mentioned later in the paper with respect to evolving this system to include additional functionality (see section 5). Importantly, all of the aspect algorithms in this paper (representing feature resolution modules), along with other features from [11], have been implemented in a simulation of an email system (see also section 5).

In this case study, we will follow the same approach as [11] regarding the architecture and the features, i.e. a user originates a message, conforming to email message format standards [2][23], from an email client program. This message then passes through one or more feature processing components, each termed an *email feature component*, until the message is delivered to the email client of the intended recipient(s). We also borrow terminology from the Distributed Feature Composition (DFC) approach of [16], calling each feature component a *feature box*.

#### 4.1 An interaction resolution for *RemailMessage* vs *MailHost*

As a first illustrative example of an undesirable interaction, consider the composition of *RemailMessage* with *MailHost* (both server-side features). A problem arises if a message is sent via *RemailMessage* to a user supposedly associated with the *MailHost*, but who is actually unknown to this feature. In this case, the *MailHost* will create and send a message to the message’s originator saying “There is no such a user as ...”. As a result, the message body of this reply has leaked the real ID corresponding to the pseudonym, therefore defeating *RemailMessage*’s intention of maintaining anonymity.

This feature interaction problem can be resolved by two possible approaches:

- **res1:** allowing the *RemailMessage* feature to query the existence of a recipient before it carries on remailing the message; or alternatively,
- **res2:** forcing any reply message to also be remailed, to ensure the real user name is not leaked (see also section 4.3 on preventing the leaking of user names).

The existence of two possible approaches here highlights an interesting point: when we say something is a resolution of an interaction, this is a subjective judgement since resolutions on the same feature interaction problem may vary from developer to developer. Sometimes the resolution just meets a requirement of the features’ users, rather than being a sound rationalisation. In this paper, we assume that any solution that is able to mitigate a feature

interaction constitutes a *resolution* of that interaction. Therefore, the simplest resolution is to disable one of the interacting features. However, real world applications are likely to need a more deliberate resolution so as to improve the quality of service.

As an example, an overview of a possible Java implementation of *RemailMessage*'s hard logic is shown in figure 2:

```

class Remailer implements Pipe {
    String myID;
    ArrayList pairTable;
    public Remailer(...) {
        .....
    }
    public void send(Message msg) {
        .....
    }
    public void receive(Message msg) {
        if ( !(msg.getReceiver().equals(myID)) )
            msg = changeIncomingID(msg); //incoming message
        else
            msg = changeOutGoingID(msg); //outgoing message
        send(msg);
    }
    private Message changeOutGoingID(Message msg) {
        String content= msg.getContent();
        String target = content.substring(0,content.indexOf("\n"));
        content = content.substring(content.indexOf("\n")+
            String alias = findAlias(msg.getSender());
        msg.setReceiver(target);
        msg.setSender(alias);
        msg.setContent(content);
        return msg;
    }
    private Message changeIncomingID(Message msg) {
        String realAddress = findRealID(msg.getReceiver());
        if (realAddress == null) throw new IllegalArgumentException();
        msg.setReceiver(realAddress);
        return msg;
    }
    public String findAlias(String outgoing) {
        .....
    }
    public String findRealID(String incoming) {
        .....
    }
}

```

**Figure 2.** RemailMessage's hard logic

The hard logic takes care the translation of user address from/ to pseudonym. In order to do this, for an incoming message, it will replace the receiver's address with a real user address; for an outgoing message, it will get the first line of the message body, and put it to the receiver slot, then replace the sender address with a pseudonym. The *Pipe* interface, which contains two methods, *receive(..)* and *send(..)*, must be implemented for the connection of feature boxes.

We can see that the hard logic of a feature is simple, cohesive, and highly consistent to its original specification, and thus easily understood. Typically, these features have two basic parts:

- Some data (structures) such as a forward address, a list of filter addresses or a list of <pseudonym, real name> pairs, or even the prescribed message content.
- Some methods to operate on the data and provide necessary feature logic to implement a service feature.

Under normal circumstances, this hard logic works fine with other features without problems. However, as is well known in the telecommunications domain, some combinations of features lead to undesirable interactions, as with the *RemailMessage* and *MailHost* interaction mentioned above.

With an interaction such as this, one feature (either *RemailMessage* or *MailHost*) is required to incorporate new additional behaviour to allow it to adapt to the countering feature; i.e. interaction resolution code needs to be introduced.

Now suppose we consider a possible resolution to the above and elect for the first resolution option, *res1*, from above (querying the existence of a user).

For the Java code presented above (figure 2), this resolution could take the form of directly modifying the receive method to allow *RemailMessage* to ask *MailHost* if the intended user exists or not. If this returns true, carry on as usual; if false, *RemailMessage* itself should create a “no such user” reply to the sender. An example of this direct modification is shown in figure 3:

```

public void receive(Message msg) {
    if ( !(msg.getReceiver().equals(myID)) ) //this is an incoming message
        msg = changeIncomingID(msg);
    if (queryExist(MailHost, userID) == false){           // 'tangled' resolution code
        msg = createReply(msg, "no such user");           // ...
        send(msg);                                       // ...
        return;                                          // ...
    }
    else //this is an outgoing message
        msg = changeOutGoingID(msg);
    send(msg);
}

```

**Figure 3.** An interaction resolution tangled with the feature code

Alternatively, resolution code could be added to the *MailHost*. For example, the *MailHost* could check if a message is from *RemailMessage*, in which case it should answer to *RemailMessage* instead of directly to the original sender.

As can be seen from figure 3, adding in resolution code for interworking with one other feature is not too drastic concerning the structure or complexity of the feature code. However, considering the number of interactions found in [11], adding resolution code to handle all such interactions is clearly going to very quickly destroy the structure of the features and the original clarity of the hard logic. It is also obviously going to be harmful with respect to the maintenance of the code or future evolution. For this reason, we have proposed the lifting of the resolution code to the meta-level. With the support of AspectJ this can be elegantly implemented as shown in figure 4:

```

aspect ResolveWithMailHost {
    after() returning(Message m):execution(Message Remailer.changeIncomingID(Message)){
        if (queryExist(MailHost, userID) == false) //throw exception if userID not known
            throw new IllegalArgumentException();
    }

    void around(Message msg):execution(void Remailer.receive(Message)) && args(msg) {
        try{
            proceed(msg); //execute the original receive method
        }
        catch(IllegalArgumentException e) { //handle new exception that may be thrown
            msg = createReply(msg, "no such user");
            send(msg);
        }
    }
}

```

**Figure 4.** Separating the interaction resolution (soft logic) from the core feature code

This aspect inserts code after the execution of the *Remailer's* *changeIncomingID* method, throwing an exception if the *userID* is false. Having thrown an exception, the receive method now needs exception handling code to be added. This is achieved by wrapping the try and catch constructs *around* the original receive method.

The above example reflects two points about the hard logic and soft logic of a feature:

- the *hard logic* is the relatively stable description of a feature’s behaviour, and
- the *soft logic*, representing an interaction resolution, is variable and may change depending on a need to adapt to a context change (such as new deployment of features and technology updates) or depending on subjective judgement regarding the best mechanism for handling interacting features (as discussed above).

#### 4.2 An interaction resolution for *RemailMessage* vs *AutoResponder*

Now consider the deployment of the *AutoResponder* feature. There are two scenarios with the *RemailMessage* being involved, both of which have been documented in [11]:

- **Scenario 1:** Bob has a remailer account and also switches on the *AutoResponder* feature before he goes on vacation. Alice sends a message to Bob’s *Remail* account (i.e. to Bob’s pseudonym rather than his real ID). When the *AutoResponder* receives the message via *RemailMessage*, it replies automatically and directly to Alice, but using Bob’s real ID rather than his pseudonym. Because of the *AutoResponder*’s direct reply, Alice can infer the *Remail* account is for Bob, thus defeating the *RemailMessage*’s purpose.
- **Scenario 2:** As above, Bob has a *Remail* account and switches on the *AutoResponder* feature before he goes on vacation. Alice sends a message that requires a reply to both Bob’s *Remail* account and to his *MailHost* account, since Alice doesn’t know that these two accounts are actually for the same person. According to the *AutoResponder*’s rule, this feature should reply only once for messages from the same address. For this reason, Alice will only receive a reply from one of Bob’s accounts, and never from the other account.

The general problem here is that the *AutoResponder* has no knowledge of whose messages it is answering (e.g. whether the message has been received via a remailer). One possible resolution is for the *AutoResponder* to check whether it is answering a message from the remailer. If so, reply to *RemailMessage* by following the remailing rule, namely set the receiver field as the *RemailMessage* and put the intended recipient’s address in the first line of the content. This algorithm can be represented as AspectJ as shown in figure 5 (but note that as discussed above, it is likely that there will be more than one viable resolution – see next section below).

Typical of resolution modules such as this, is the need for them to know information about prior feature boxes. One solution to this is to require each active feature box to add a tag to declare its process status when a message goes through it. Importantly, this simple protocol can also be implemented through AOP techniques as a crosscutting concern. For example, *pointcuts* can be defined that correspond to exit points from the execution of the feature boxes. On exiting a particular feature box, an *after advice* can be declared to manipulate the message and add the required tag.

```

aspect SoftenAutoForRemail {
  before(Message msg):execution(void AutoResponder.send(Message) && args(msg) {
    //if msg is from Remailer, then respond to Remailer using the remailing rule
    if (isFromRemail(msg)) {
      writeContentFirstLine(msg.getReceiver());
      msg.setReceiver(getRemailAddress(msg));
    }
  }
  boolean isFromRemail(Message msg) {
    //check if msg is from Remailer ...
  }
  void writeContentFirstLine(String str) {
    //write intended recipient's address in the 1st line of message content ...
  }
  String getRemailAddress(Message msg) {
    //get Remail Server's address from msg ...
  }
}

```

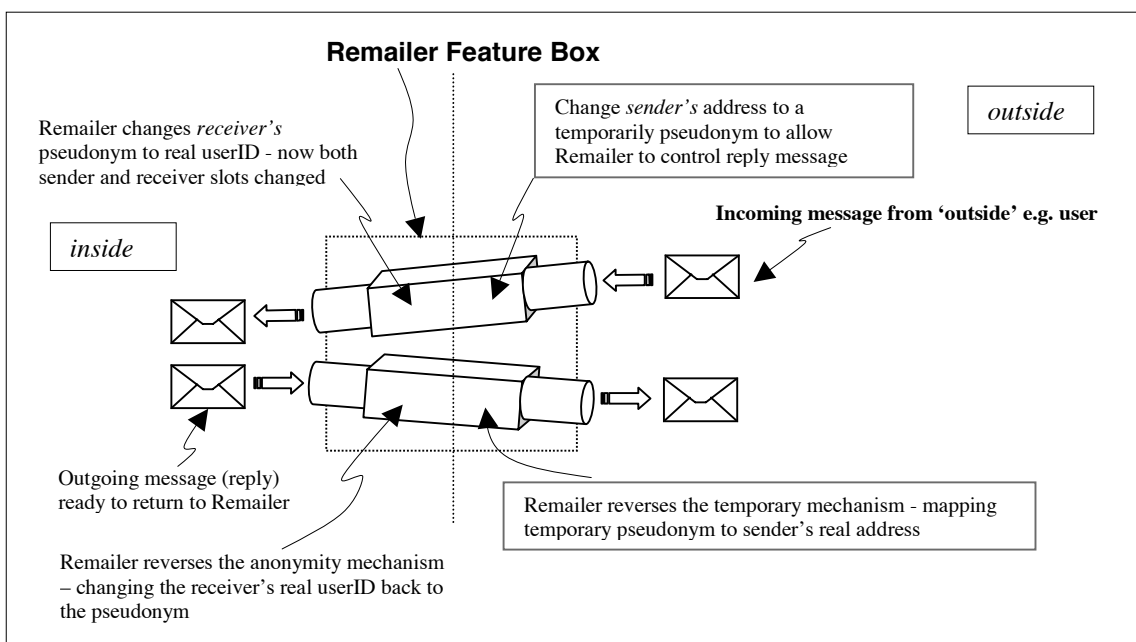
**Figure 5.** The AutoResponder's soft logic – to handle messages from a Remailer

### 4.3 A more general interaction resolution for RemailMessage and AutoResponder

To illustrate the possible diversity of resolutions for a given interaction problem, and to further evaluate the appropriateness of AspectJ, we present an alternative resolution for the *RemailMessage* and *AutoResponder* interaction problem. In this second solution, we introduce a mechanism into the *Remailer* (using AspectJ) to enforce the rule:

- For each message sent to *RemailMessage* from a user, any internal reply message must also pass back through *RemailMessage*.

In order to achieve this, *RemailMessage* can replace the sender's userID with a temporary pseudonym, thus ensuring that any return message must also pass through the remailer in order to be mapped back to the sender's real userID. Then, to allow the receiver to know the original sender of the message, the original sender's information should be put in another slot, e.g. the first line of message body. The mechanism for this resolution algorithm is showed in figure 6.



**Figure 6.** The Remailer feature box and interaction resolution algorithm

To reverse the process, when receiving the inner user's reply (probably from another feature box, such as *AutoResponder*), *RemailMessage* must replace the originating sender's temporary pseudonym with the sender's real address. If required during this translation, *RemailMessage* can also check user names against the email content to make sure there is no further leaking of the user ID (illustrated in figure 7 by the inclusion of a `preventLeaking` method).

To achieve this, *RemailMessage* should keep two lists of <pseudonym, realname> pairs: one for the anonymity of its customers, and one for the enforcement of controlling replies. In this way, the two interaction problems mentioned above are solved readily:

- **Scenario 1:** The *AutoResponder's* reply will be sent back to *RemailMessage* first, rather than directly to the originator. *RemailMessage* can thus anonymise the recipient's real identity.
- **Scenario 2:** In this scenario, two messages arrive at *AutoResponder* with different sender addresses (since the remailer has allocated each message a different temporary pseudonym). Consequently, both messages will be properly replied.

This interaction resolution algorithm can be implemented in Aspect J as shown in figure 7.

```

aspect ResolutionAll {

    public ArrayList Remailer.guestList; //introduce a new dual list

    public String Remailer.createAnAlias(String guest) {
        //introduce new Remailer method to create a pseudonym for the sender
    }
    public String Remailer.findGuestRealID(String alias) {
        //introduce new Remailer method to find the real username given a pseudonym
    }
    public void preventLeaking(Message msg) {
        //check the message content and replace any real userID with pseudonym
    }

    //assign a name to the sender of incoming msg.
    after(Remailer rm) returning(Message msg):
        call(Message Remailer.changeIncomingID(Message)) && target(rm) {
            msg.setSender(rm.createAnAlias(msg.getSender()));
        }
    //make sure outgoing message contains no leaking of real userID
    after(Remailer rm) returning(Message msg):
        call(Message Remailer.changeOutgoingID(Message)) && target(rm) {
            preventLeaking(msg);
        }

    //process the reply message from the inner user (define named pointcut first)
    pointcut messageArrive(Remailer rm):
        target(rm) && call(void Remailer.receive(Message));

    //continued ...
}

```

```

// ... continued
void around(Message msg, Remailer rm): messageArrive(rm) && args(msg) {
    try {
        proceed(msg, rm);
    }
    catch(IllegalArgumentException e) {
        String realReceiver= rm.findGuestRealID(msg.getReceiver());
        if (realReceiver != null) {
            msg.setReceiver(realReceiver);
            String senderPseudonym = rm.findAlias(msg.getSender());
            if (senderPseudonym != null){
                msg.setSender(senderPseudonym);
                preventLeaking(msg);
                rm.send(msg);
            }
            else throw new IllegalArgumentException();
        }
        else throw new IllegalArgumentException();
    } //end of catch
} //end of around
} //end of aspect

```

**Figure 7.** An aspect-oriented implementation of the new interaction resolution

With this more general interaction resolution, it is important to observe that the original interaction with *MailHost* can also be resolved. The reason for this is that the “no such user” reply message will be returned to *RemailMessage* first, rather than directly to the user. As a result, this provides us with an effective mechanism to ensure the userID is always hidden.

#### 4.4 Summary: aspect-oriented programming for interaction resolutions

The examples above have hopefully provided an insight into how resolution modules, implemented in AspectJ, can be used to cleanly separate the interaction resolution code (soft logic) from the feature’s core behaviour (hard logic). Benefits can be gained, especially for more complex resolution modules such as the one provided in figure 7, when compared with the more traditional approach of entwining the resolution behaviour with the feature’s core behaviour.

In particular, in this example, three new methods and a new attribute (the dual list) have been *introduced* into the remailer’s behaviour. In addition, new behaviour (*advice*) has been provided that will be *woven* into the feature code: *after* the feature’s two changeID methods and *around* the receive method. Whereas the introductions would be relatively easy to separate structurally in the original feature code (e.g. the appropriate use of comments and grouping of related code), the extra woven behaviour cannot be effectively separated. This leads to difficulties for the maintenance and evolution of the overall system.

Perhaps more importantly, the example in figure 7 also gives us an indication that one resolution module, although using a more complex algorithm, can resolve multiple feature interactions. This is where the power of our aspect-oriented approach brings further benefits. Pointcuts can be defined over any execution points in any of the feature boxes and need not be restricted to one feature box (as has been the case in our examples above). Also, pointcut definitions can use wildcard-matching techniques, thus removing the need for the tight-coupling of aspects with features via actual method names.

With further investigation, we hope that this may lead to the implementation of general interaction resolution *patterns*, although it is inevitable that more concrete (domain-specific) resolution modules will also still be required.

## 5. Evolution of the email system to further evaluate our approach

The two-level architecture we have proposed consists of:

- **Base level:** contains features' core behaviours, known as *hard logic*; implemented using *Java*
- **Meta-level:** contains interaction resolution modules, known as *soft logic*; implemented using *AspectJ*

To further evaluate the effectiveness of this architecture, we have considered evolving the email system that we have presented so far (with three features), by extending it to all ten features of [11]. In order to make a working system, we have also refactored some of the GUI modules from ICEMail, an email client written in Java and based on the new Java Mail API [15].

We classify our evaluation into five properties: cleanness of separation, re-use, faithfulness of implementation to specification, adaptability to requirement change and support for interaction avoidance, detection and resolution. It should be noted that these properties are, by their nature, more qualitative than quantitative.

A further evaluation criterion we would like to address in the future is that of performance. It can be expected that, in line with other meta-level/ reflective approaches, our approach will incur at least a minor performance overhead. We have not yet investigated this further, although information regarding the performance of AspectJ can be found on the AspectJ web-site (FAQ), see [19].

### 5.1 Cleanness of separation

As mentioned above, we believe that the cleanness of separation is a key factor for a system's effective maintenance and evolution. Object-oriented techniques already offer a widely accepted and largely effective form of separation through encapsulation and inheritance. However, feature-oriented systems implemented in this manner still tangle the resolution code, which allows a feature to work with other features, with a feature's core behaviour (e.g. as presented in figure 3).

The examples given in this paper have illustrated the cleanness of separation achieved with our two-level architecture. All other features in [11] also display this elegant separation when implemented. As the number of features increase, the importance of this separation also increases. For example, [11] has identified 26 interactions between the 10 features considered. Should an architecture based on separation of concerns *not* be employed, this clearly results in an unenviable number of extensions required to the basic feature code. A further valuable point is that not every interaction requires a separate resolution module (as illustrated by figure 7 above). Although the subject of future work, this also gives us incentive to look for more general interaction resolution *patterns*. Importantly, our approach will provide support in cases where patterns occur in, or cross-cut, multiple features.

As a further case for the cleanness of separation achieved in our approach, we have also re-factored some of ICEMail's GUI modules. The reason for this was our observation that, in the original modules, there was a great deal of code-tangling. For example, *FolderTableModel* is a GUI component for the display of messages in an email folder. It implements the *TableModel* interface in `javax.swing.table` and users can read their incoming message via this component (supported by most of today's email clients) [15]. However, the *FolderTableModel* should only be responsible for the displaying of messages; the actual management of messages should be carried by another management component,

namely the content provider for *FolderTableModel*. Similarly, the *Fetchlet* class, a class used to get messages from a user's mailbox and place them in a local folder, need not know that the messages it fetched will be displayed by *FolderTableModel* class.

In our re-factoring, a simple interaction resolution called *Compose\_FolderTableModel\_Fetchlet* has been implemented as an aspect. This aspect knows about the interactions between the two classes, significantly reducing the size of each feature module. For example, the initial *FolderTableModel* contained 800 lines of source code, whilst our re-factored version contains only 70 lines. This reduction is achieved because, in our opinion, the original classes contained too many tangled concerns or interactions. The re-factored code demonstrates a much cleaner separation of concerns and hence, we believe, will permit more effective maintenance and future evolution.

## 5.2 Re-use

Interaction resolution modules such as those identified in figures 4 and 5 above (i.e. aspects representing very specific resolutions), perhaps only offer limited opportunities for re-use. This is mainly because the chosen resolution, of which there may be a number of options, depends on factors such as an ever-improving technological base that permits improvements to existing features, the rapid development of new features, a subjective judgement as to the best resolution, etc. Hence, we believe it is more likely that, for specific resolution modules, new modules will be developed rather than re-using existing ones.

However, our two-level architecture has the best opportunities for re-use at the base level, rather than the meta-level. We have experimented with the two GUI modules mentioned above, and have found our re-factored modules easier to re-use than the original modules. The reason for this is that the interaction concerns (soft logic) have been extracted from the core behaviour (hard logic), leaving a more generic feature component. For example, the *FolderTableModel* class can be easily re-used in another application with very minimal changes.

Importantly, if we can achieve one of our goals of identifying interaction resolution *patterns*, these obviously have much greater potential for re-use. In investigating this further, we have found that many of the interaction resolutions (like those we have mentioned above) involve boundary condition checking. Furthermore, all of the resolutions for the feature interaction cases of [11] can be fitted into some kinds of boundary checking patterns. Amongst the resolutions for the 26 feature interactions identified in [11], as well as an additional feature interaction identified by ourselves (see section 5.5 below), more than half can be generalised as generic resolutions that can be applied to other feature interaction problems. Hence, the identification of resolution patterns appears a realisable aim, and has obvious benefits for re-use.

## 5.3 Faithfulness of implementation to specification

Our two-level architecture has been designed to facilitate keeping the feature's implementation faithful to its specification. Not only does this improve the readability and simplicity of the resulting code, but also allows the feature's specification to map to the implementation more directly, thus facilitating better reasoning about the mapping. This, in turn, opens the door to generative programming techniques, as widely used in component-based development to generate code (or code templates) automatically from the specification.

#### *5.4 Adaptability to requirement change*

There are two key points of our architecture that help a developer to react to changing requirements. The first of these is that the separation we provide allows the developer to integrate new features into the system, without needing to consider, or worse rewrite, the existing features.

Secondly, and the point that to our knowledge makes our approach unique, is that by adopting aspect-oriented programming for the separate resolution modules, the developer can implement a feature without considering the interactions with other features, then focus on the interaction issues separately. The power of aspect-oriented techniques makes this last step viable and effective, as has been illustrated by the examples above.

Similarly, the removal of features from a system is just as clean and effective because of the separation we provide, meaning that any interactions have been made explicit. This helps to avoid redundant code being left embedded in feature boxes, a situation that leads to unnecessary complexity and lowers efficiency.

#### *5.5 Support for interaction avoidance, detection and resolution*

The central objective in this paper has obviously been in the provision of an architecture to cleanly handle the interactions between features. Our implementation of the email system has shown how our two-level architecture and the use of AspectJ can be used very effectively to handle feature interactions.

However, a crucial part of the paper that remains to be addressed is that, whilst our approach explicitly handles features interactions by providing separate resolution modules, what happens if our resolution modules themselves interact? Far from being a theoretical question, problems with resolution modules themselves interacting in undesirable ways have been identified in [28]. This is not an unexpected discovery, since resolutions themselves can be viewed as features, which, of course, are prone to interactions. A number of highly significant issues crop up with respect to this topic that we address in turn below.

#### *Two-level versus multi-level architectures*

By proposing a two-level architecture, we have effectively ruled out a multi-level architecture that is typical of many reflective architectures. One option would be to relax this condition, effectively allowing a meta-meta layer to handle the resolution interactions, and theoretically an infinite hierarchy to handle subsequent meta-meta level interactions and beyond. However, it should be noted that, in practice, reflective architectures rarely require more than three layers to be reified.

The problem with adopting this more relaxed approach is our choice of language. As AspectJ stands at the moment<sup>1</sup>, the language does not support “aspects of aspects”, i.e. aspects cannot be defined over other aspects, thus effectively preventing adequate support for beyond a two-level architecture. It should be noted however that other AOSD approaches do provide support for “aspects of aspects”. In an email message posted on the Demeter web-site [6], a list of approaches supporting this is given, namely Incremental Programming [25], Aspectual Collaborations [22], Hyper/J [26] and DJ [24]. These require further investigation to determine if they would provide a more appropriate language choice for our work than AspectJ.

---

<sup>1</sup> To our knowledge, and after discussions in the AOSD community regarding this topic, AspectJ will remain without support for “aspects of aspects” in the foreseeable future, the developers believing the alternatives are unnecessarily complex.

### *Implicit versus explicit composition*

A further issue relating to our language choice is the fact that, in AspectJ, composition of the resolution aspects with the features is done implicitly by the aspect weaver. As a result, there is no control over the composition operator used. Other languages provide explicit support for composition, a facility that we believe would be advantageous and give more flexibility in our solution. It is conceivable, in fact, that a custom-designed composition operator would only allow valid compositions of resolution modules, thus *avoiding* the problem of such modules interacting in undesirable ways. AOSD techniques that currently offer support for explicit composition include Hyper/J [26] and Composition Filters [1].

### *Detecting feature interactions*

In our case study, we have relied heavily on the list of known interactions provided in [11], but what if these interactions are not known in advance?

In performing this case study, we have found that most of the feature subversions happen across boundary conditions. By explicitly separating out feature behaviour into our two-level architecture we are actually encouraging a systematic exploration of boundary conditions, although this is, of course, a manual process at present rather than benefiting from automated support. However, this systematic exploration led us to discover a further (undocumented) interaction as discussed below.

The *FilterMessage* feature ensures that “if a message comes from a *blacklisted address*, it must not reach the subscriber’s *user domain*”. There are two themes in this statement: “the source of the incoming address” and “the targeted user domain”. For the former, there are many possible sources of the incoming address, either human originators or mechanical originators (e.g. auto-Responder, ForwardMessage, RemailMessage, etc.).

Now consider a message from a (mechanical) forwarding feature (e.g. from party A to party B to party C). A boundary condition for this is “What if the forwarding party (B) is not filtered, but the message it is forwarding is from a blacklisted party (A)”. From this point, we identify a potential subversion of *FilterMessage* feature, as follows (undocumented in [11]):

- **Scenario:** Alice’s domain is blacklisted and filtered by Carol’s host (e.g. username carol@lancaster). Alice asks Bob, whose domain is not filtered by Carol’s host, to forward her email to carol@lancaster, thus subverting the filtering mechanism.

From Alice and Bob’s point of view, there is unlikely to be a problem here, since Bob knows Alice is using his account. However, from Carol’s point of view, her domain has been blacklisted, presumably for good reason (e.g. virus control), yet this has been subverted by the message forwarding.

By continuing to explore the boundary condition of “the source of the incoming address”, we can find other potential feature interaction problems involving *FilterMessage*.

In terms of automated support for the detection of feature interactions, there are, of course, many techniques that have been presented throughout the series of Feature Interaction Workshops, e.g. see [3]. The majority of these techniques have had a formal basis, and hence have worked from formal specifications/ models of the features. Such techniques are highly complementary to our approach.

## *Detecting resolution interactions*

Effectively, our resolution interactions are actually aspect interactions, for which the AOSD community currently has very little support. We believe that there are four possible solutions here, all of which require further investigation:

- Avoiding interactions by applying a design by contract approach (e.g. [21]) or by utilising explicit composition operators (not available in AspectJ, see discussion above).
- Relaxing our two-level architecture to allow a meta-meta layer to handle resolution interactions (this also implies a language change since AspectJ does not support “aspects of aspects”, also discussed above). Note that our two-level architecture is not a source of resolution interactions, it just restricts the way we can handle them.
- Model-checking certain properties to determine aspect interactions (in the same way that feature interactions have been identified by model-checking). This, of course, relies on the difficult problem of identifying appropriate properties for which to check.
- Investigate the recent work of [9] where execution monitors are deployed over aspects and “aspect laws” can be verified to determine the independence of aspects. Also relevant is the work of [31] where three different categorisations of aspects are identified. For certain types of aspect, they show how the proof of a particular property is relatively straightforward, yet for other types of aspect, this is more problematic. We have yet to investigate this further with respect to our work, particularly identifying which category our aspect resolutions fall into.

## **6. Conclusions**

This paper has proposed a two-level architecture to enhance the development of feature-oriented software. For this architecture, we have proposed the separation of a feature’s core behaviour (its hard logic) from extra behaviour required to allow it to adapt to other features (its soft logic). We have illustrated how the latter can be implemented at the meta-level using aspect-oriented programming techniques. Through a case study of an email system (initially very basic with just three features) we have shown the value of this clean separation.

Further, by incrementally evolving our email system from a system with just three features to a new system with ten features and a graphical user interface, we believe we have been able to critically evaluate our proposed architecture. This evaluation has raised a number of important benefits, but also areas that require further research. The most significant of these is the issue of detecting and resolving interactions between our resolution modules. These are effectively *aspect interactions*, an area that is currently under-researched in the AOSD community. Also highlighted in this evaluation has been our choice of language, AspectJ. Further investigations are required to determine whether other AOSD techniques would be more suited to our work.

## **References**

- [1] L. Bergmans and M. Aksit, “Composing Crosscutting Concerns using Composition Filters”, *Communications of the ACM*, Vol. 44, No. 10, pp. 51-57, 2001.
- [2] R.J.A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray and S. Mankovski, “Feature-Interaction Visualization and Resolution in an Agent Environment”, In *Proceedings of the 5th International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW’98)*, Lund, Sweden, IOS Press, Amsterdam, pp 135-149, October 1998.

- [3] M. Calder, E. Magill (eds), "Feature Interactions in Telecommunications and Software Systems VI", Glasgow, Scotland, IOS Press, Amsterdam, 2000.
- [4] R. Charette, "The Decision is in: Agile versus Heavy Methodologies", (results of a survey of 200 IS/ IT managers), Agile Project Management Executive Update, Vol. 2, No. 19, Cutter Consortium, see <http://www.cutter.com/freestuff/epmu0119.html>
- [5] M.A. Cusumano and R.W. Selby, "Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People", Simon & Schuster, 1998. ISBN: 0684855313.
- [6] "Aspects of aspects", email correspondence on Demeter web-site, 2002, <http://www.ccs.neu.edu/research/demeter/related-work/aspects-of-aspects/reading-list>
- [7] J. de Luca, "Feature Driven Development: The Community Portal for all things FDD", Nebulon Pty Ltd, <http://www.featuredrivendevelopment.com/>. See also linked document "Agile Software Development using Feature Driven Development (FDD)", <http://www.nebulon.com/fdd/index.html>
- [8] E. W. Dijkstra, "A Discipline of Programming", Prentice Hall, 1976. (Now re-published, Prentice Hall PTR/Sun Microsystems Press, ISBN: 013215871X, 1997).
- [9] R. Douence, P. Fradet, and M. Südholt, "A Framework for the Detection and Resolution of Aspect Interactions", SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), 2002, ACM.
- [10] FDD (Feature Driven Development) Process #5: Build By Feature, A Practical Guide to Feature-Driven Development, Step-10 Pte Ltd., 2002. See <http://www.step-10.com/FDD/FDD5BuildFeatures.html> (web site associated with [27])
- [11] R. Hall, "Feature Interactions in Electronic Mail", In [3], pp 67-82, 2000.
- [12] W. Harrison and H. Ossher, "Subject-Oriented Programming (a critique of pure objects)", in Proceedings of OOPSLA'93, pp 411-428, 1993. See also <http://www.research.ibm.com/sop/>
- [13] J. Highsmith and A. Cockburn, "Agile Software Development: The Business of Innovation," IEEE Computer, September 2001.
- [14] J. Highsmith, "Agile Software Development Ecosystems", Addison Wesley, 2002. ISBN 0-201-76043-6.
- [15] ICEMail email client, <http://www.icemail.org>. See also D. Nourie, "The Java™ Technologies Behind ICEMail: An Open-Source Project", June 2001, see <http://developer.java.sun.com/developer/technicalArticles/javaopensource/icemail/>.
- [16] M. Jackson and P. Zave, "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services", IEEE Transactions on Software Engineering, 24(10):831-847, October 1998.
- [17] G. Kiczales, "Towards a New Model of Abstraction in Software Engineering", International Workshop on New Models in Software Architecture, Reflection and Meta-Level Architecture, 1992.
- [18] G. Kiczales, "Aspect-Oriented Programming", ACM Computing Surveys, 28(4es), Article number 154, December 1996. ISSN:0360-0300.
- [19] "Getting Started with AspectJ", Communications of the ACM, Vol. 44, No. 10, pp. 59-65, 2001. See also <http://aspectj.org/>
- [20] G. Kiczales (ed), 1<sup>st</sup> International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, ACM Press, April 2002. ISBN: 1-58113-469X.
- [21] H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck, "Aspect Composition Applying the Design by Contract Principle", 2nd International Symposium on Generative and Component-based Software Engineering (GCSE), Lecture Notes in Computer Science 2177, pp. 57-69, Springer-Verlag, 2000.
- [22] K. Lieberherr, D.H. Lorenz and J. Ovlinger, "Aspectual Collaborations: Combining Modules and Aspects", Technical Report NU-CCS-02-?, 2002. Available from <http://www.ccs.neu.edu/research/demeter/papers/publications.html>
- [23] J. Myers and M. Rose, Internet Request for Comments 1725: Post Office Protocol -- version 3, 1994. See <http://www.ietf.org/rfc.html>
- [24] D. Orleans and K. Lieberherr, "DJ: Dynamic Adaptive Programming in Java", Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, Springer Verlag, 2001.
- See also <http://www.ccs.neu.edu/research/demeter/biblio/DJreflection.html>
- [25] D. Orleans, "Incremental Programming with Extensible Decisions", First International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, G. Kiczales (ed), ACM Press, 2002. See also <http://www.ccs.neu.edu/home/dougo/papers/aosd02/>
- [26] H. Ossher and P. L. Tarr, "Hyper/J: multi-dimensional separation of concerns for Java", International Conference on Software Engineering, pp. 734-737, ACM, 2001. See also <http://www.research.ibm.com/hyperspace/>
- [27] S. Palmer and M. Felsing, "A Practical Guide to Feature-Driven Development", Prentice-Hall, 2002. ISBN: 0130676152. See also <http://www.step-10.com/FDD/index.html>
- [28] J. Pang and L. Blair, "Resolving Feature Interactions with AOP and Feature Driven Software Development", draft paper available from Computing Department, Lancaster University, Lancaster, LA1 4YR, December 2002.

- [29] D.L. Parnas “On the Criteria to be used in Decomposing Systems into Modules”, Communications of the ACM, Vol. 15, No. 12, 1972.
- [30] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin, “JAC: A Flexible Solution for Aspect-Oriented Programming in Java”, 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection’01), Lecture Notes in Computer Science 2192, pp 1-25, Springer-Verlag, 2001.
- [31] M. Sihman, S. Katz, “Superimpositions and Aspect-Oriented Programming”, to appear in The Computer Journal (British Computer Society), special issue on Aspect Oriented Software Development, 2003.
- [32] P. Zave, “FAQ Sheet on Feature Interactions”, AT&T, 2001. See <http://www.research.att.com/~pamela/faq.html>