Aspect-Oriented Solutions to Feature Interaction Concerns

Lynne Blair, Jianxiong Pang
Lancaster University, U.K.

In a nutshell …

- A two-level architecture for feature driven software development
  - One level, the ***base layer***, for a feature's core behaviour
    - Object-oriented, e.g. Java
  - Another level, the ***meta-layer***, for resolution modules to provide solutions (resolutions) to feature interaction problems
    - Aspect-oriented, e.g. AspectJ, AspectWerkz, etc.

Overview: Aspect-Oriented Programming

- Range of techniques to facilitate ***enhanced separation of concerns***
  - Historical links with reflective techniques
  - Also links with subject oriented programming, composition filters, hyperslices, superimpositions, etc.
- Recognises ***crosscutting concerns***
  - Security, logging, tracing, debugging, error handling, non-functional concerns, etc.
  - AOP elegantly captures concerns that cut across modules and avoids "code-tangling"

A simple AOP tracing example

- Using AspectJ (Kiczales et al) – http://aspectj.org
  - a mature and well-supported AOP language

```
aspect TraceAspect {
  // set up pointcut to trace all methods in all (base-level) classes
  pointcut allMethods(): execution(* *(..));

  // around - describe what happens around the allMethods pointcut
  void around (): allMethods() {
    System.out.println("Entering "+thisJoinPointStaticPart.getSignature());
    proceed();
    System.out.println("Leaving "+thisJoinPointStaticPart.getSignature());
}   }
```

Using AspectJ

… with aspects

```
C:\myjava>ajc *.java

C:\myjava>java Driver
  trace: Entering void Driver.main(String[])
    trace: Entering void ClubList.add(Club)
    trace: Leaving void ClubList.add(Club)
    trace: Entering void ClubList.add(Club)
    trace: Leaving void ClubList.add(Club)
    trace: Entering void ClubList.add(Club)
    trace: Leaving void ClubList.add(Club)
    trace: Entering void ClubList.display()
      trace: Entering void ClubElement.display()
        trace: Entering void Club.display()
The favourite night at Liquid is Friday
        trace: Leaving void Club.display()
      trace: Leaving void ClubElement.display()
      trace: Entering void ClubElement.display()
        trace: Entering void Club.display()
The favourite night at The Carleton is Thursday
        trace: Leaving void Club.display()
      trace: Leaving void ClubElement.display()
      trace: Entering void ClubElement.display()
        trace: Entering void Club.display()
The favourite night at The Sugarhouse is Wednesday
        trace: Leaving void Club.display()
      trace: Leaving void ClubElement.display()
    trace: Leaving void ClubList.display()
  trace: Leaving void Driver.main(String[])
```

```
\myjava

03    327 Club.java
09    269 ClubElement.java
7     441 ClubList.java
9     339 Driver.java
9     927 TraceAspect.java
```

Without aspects …

```
a


uid is Friday
 Carleton is Thursday
 Sugarhouse is Wednesday
```

Applying AOP to Feature Driven Development

- Requirement: features must be capable of working with other features
  - but, a feature designer cannot foresee future features that will interact with his/her feature
  - adding multiple instances of resolution code to existing code compromises structure/ elegance
  - this is a classic example of "code tangling"
- One solution: separate all resolution code from core code using AOP techniques
- Result: a two-level architecture for FDD

A classic resolution example …

- Consider standard telephony features
  - Classic interaction example: CFB vs VoiceMail
  - Implement CFB and VoiceMail as separate features
  - Simple resolution strategy:

```
if BUSY and caller in <special_user_list>
   proceed with CFB
else
   proceed with VoiceMail
```

  - Can viewed as logically belonging alone, rather than being embedded with CFB and/ or VoiceMail

Interactions and resolutions in an email system

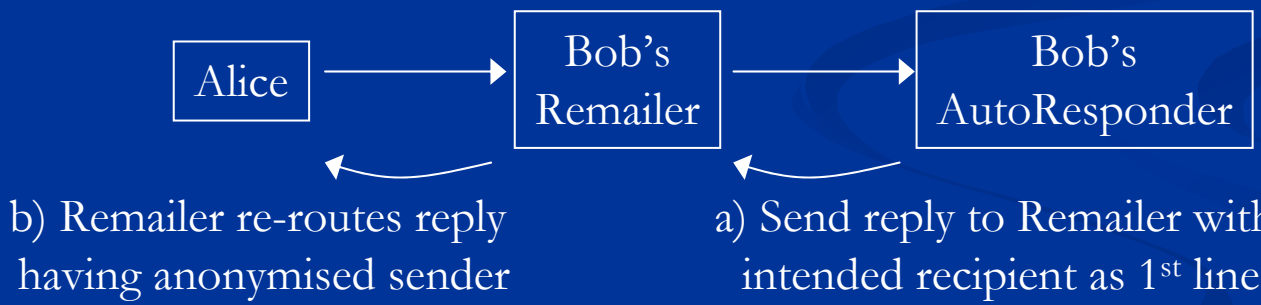
- Our study is based on the 10 email features (and 26 identified feature interactions) from:
  - R. Hall, "Feature Interactions in Electronic Mail", Feature Interactions in Telecommunications and Software Systems VI (Glasgow), pp 67-82, 2000
- Examples of features:
  - RemailMessage – provides pseudonym for sender
  - AutoResponder – automatically replies to messages
  - etc.

Feature Interaction: Remailer vs AutoResponder

- Interaction identified in Hall's paper between these 2 features - under 2 separate scenarios
- One interaction scenario:
  - Bob has a *Remail* account and *AutoResponder* feature.
  - Alice sends message to Bob's *Remail* account (pseudonym).
  - *AutoResponder* receives message via *RemailMessage* and replies automatically and directly to Alice, using Bob's real ID.
  - Because *AutoResponder* replies directly, Alice infers the *Remail* account is Bob's, thus defeating *RemailMessage's* purpose.

Resolution for Remailer vs AutoResponder

- Resolution can be achieved by (at least) two different mechanisms:
  1. In *AutoResponder*, check if answering a message from the Remailer. If so, reply using the Remailing rule:



  2. OR, modify the Remailer to ensure that all replies pass back through it (allocate sender a pseudonym)

Resolution (1) in AspectJ

```
aspect SoftenAutoResponderForRemailer {

  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 ...
}}
```

Summary so far

- We have separated interaction resolution modules from a feature's core behaviour:
  - Can *introduce* new methods and/or new attributes into feature's behaviour as required
    - Relatively easy to maintain separation w/out aspects
  - Can weave new behaviour (advice) *around*, *before* or *after* existing feature behaviour
    - Difficult to maintain separation w/out aspects
  - Pointcuts can be defined over any execution points in any feature box
    - i.e. aspects need not be specific to one feature box

Evaluation

- To evaluate:
  - Extended system to all 10 features of R. Hall's paper
  - To create running system, refactored some of GUI modules from ICEMail (email client, Java Mail API)
- Evaluation categories:
  - Cleanness of separation
  - Re-use
  - Faithfulness of implementation to specification
  - Adaptability to requirement change
  - Support for interaction avoidance, detection, and resolution

Cleanness of separation

- Object-orientation provides a widely accepted and largely effective level of separation
- Yet feature-oriented systems that require resolution modules for inter-working suffer "code-tangling"
- Two-level architecture provides elegant separation
- Increasingly important as number of features ↑
  - R. Hall's paper - 10 features, 26 identified interactions
  - Each interaction requires a resolution
  - Possible interaction resolution *patterns*
  - ICEMail system – "core" functionality: 800 lines → 70

Re-use

- The cleanness of the "core" functionality, gives good opportunities for re-use (e.g. ICEMail)
- Many resolution modules are very specific, hence only offer limited scope for re-use
- Re: patterns …
  - All of the 26 interactions arise from the need for some level of boundary checking
  - Approx. half could be classed as 'generic' (could be implemented as a pattern and applied elsewhere)
  - Still requires more work !

Faithfulness of implementation to specification

- We claim that our two level architecture:
  - Improves readability and simplicity of code
  - Allows a feature's specification to map more directly to its implementation
  - This, in turn, opens the door to generative programming techniques (for code or code templates)

Adaptability to requirement change

- New features can be added to system without consideration (or re-writing) of other features
- Also no need to consider interactions when writing a feature – can focus on these separately
- Similarly removal of features is straightforward
  - Avoids redundant code being left embedded with features

Support for interaction avoidance, detection and resolution

- This category in fact raised further questions …
- Obvious problem:
  - Resolution modules (aspects) can themselves interact
- Many reflective architectures are multi-level
  - Should we relax 2-level architecture?
  - Meta-meta level is for solving resolution interactions?
- Language choice – AspectJ?
  - Aspects of aspects not permitted
  - Composition is implicit - in other languages it's explicit

Support for aspect interactions?

- Avoidance – by 'design by contract' approach or explicit (restrictive) composition operators
- Introduce meta-meta layer
- Formal techniques (cf. FIW) e.g. model-checking properties of aspects (but which properties?)
- Aspect community's work:
  - A framework for the detection & resolution of aspect interactions, Douence, Fradet & Südholt, GPCE'02
  - Superimpositions and aspect-oriented programming, Sihman, Katz, BCS Computer Journal 2003.

Performance?

- Coming soon!
  - See FAQ on http://aspectj.org

Final comment

- Aspect-oriented approaches provide powerful language-level support for resolution modules
- Can be used to enhance traditional software development practices
- This is **not** domain specific – examples from:
  - Telephony and email
  - Middleware's system/ technical services
  - Tracking systems
  - Building Control Systems (coming next) …