Aspect-Oriented Programming

Anya Helene Bagge

Department of Informatics
University of Bergen

LRDE Seminar, 26 Mar 2008

Quick Overview of AOP

- (you may go back to sleep after these first two slides)
- Separation of Concerns: break things down into non-overlapping encapsulated pieces (methods, classes, modules)
- Cross-Cutting Concerns: things that can't be easily encapsulated by standard abstractions
- Aspects are abstractions for cross-cutting concerns!
- Example applications: error checking and handling, synchronisation, context-sensitive behaviour, performance optimisations, monitoring and logging, debugging support, and multi-object protocols

Aspects

- *advice*: how a cross-cutting concern should be implemented
  - e.g., print a message: "foo() was called!"
- *join-points*: potential places where advice can be applied
  - e.g., when a method is called
- *pointcuts*: identifies at which join points advice should be applied
  - e.g., all calls to methods with names beginning with `set`
- *weaving*: putting together aspects and normal code into a complete program
  - done by the *aspect weaver*

Motivation – example

- Transferring money between accounts – what about
  - checking that the user has access?
  - interrupted transfers?
  - logging?

```
void transfer(Account fromAccount, Account toAccount,
              int amount){
  if (fromAccount.getBalance() < amount) {
    throw new InsufficientFundsException();
  }
  fromAccount.withdraw(amount);
  toAccount.deposit(amount);
}
```

Tangled code:

```
void transfer(Account fromAccount, Account toAccount, int amount) throws Exception
   if (!getCurrentUser().canPerform(OP_TRANSFER)) {
     throw new SecurityException();
   }
   if (amount < 0) {
     throw new NegativeTransferException();
   }
   Transaction tx = database.newTransaction();
   try {
      if (fromAccount.getBalance() < amount) {
        throw new InsufficientFundsException();
      }
      fromAccount.withdraw(amount);
      toAccount.deposit(amount);
      tx.commit();
      systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount, amount);
   }
   catch(Exception e) {
      tx.rollback();
      throw e;
 } }
```

Why is this bad?

- Less than half the code deals with the actual business logic
- Logging and access control is most likely needed (and the same) for most account methods
- Transactions are needed for all compound methods
- Getting any of it wrong may have security implications or cause subtle bugs
- AOP would separate this into:
  - Core business logic (`transfer`() method)
  - Logging aspect
  - Transaction aspect
  - Access-control aspect

Join Point Models

- Possible join points (in AspectJ):
  - method/constructor/advice `call` or `execution`
  - class/object `initialization`
  - field access (`get`/`set`)
  - exception handling (`handler`)
- Pointcuts are composed of primitive pointcut designators and combinators:
  - at call to setX(): `call(void Point.setX(int))`
  - when object is of SomeType: `this(SomeType)`
  - anywhere within MyClass: `within(MyClass`
  - in control-flow of main(): `cflow(call(void Test.main()))`
  - using wildcards: `execution(int *())`
  - composition and definition: `pointcut ioHandler(): within(MyClass) && handler(IOException);`

Advice

- Before and after (returning, throwing or both):

```
before(): move() {
    System.out.println("about to move");
}
after() returning: move() {
    System.out.println("just successfully moved");
}
```

- More complicated example:

```
pointcut setter(Point p1, int newval):
   target(p1) && args(newval) &&
   (call(void setX(int) || call(void setY(int)));
  before(Point p1, int newval): setter(p1, newval) {
    System.out.println("About to set something in " + p1 +
                       " to the new value " + newval);
  }
```

More Advice

- Around advice replaces the join point – use `proceed` to run the original action:

```
void around(Point p, int x): target(p)
                          && args(x)
                          && call(void setX(int)) {
    if (p.assertX(x))
      proceed(p, x);
    p.releaseResources();
}
```

Example

- The following aspect counts the number of calls to the `rotate()` method on a `Line` and the number of calls to the `set*()` methods of a `Point` that happen within the control flow of those calls to `rotate`:

```
aspect SetsInRotateCounting {
    int rotateCount = 0;
    int setCount = 0;
    before(): call(void Line.rotate(double)) {
        rotateCount++;
    }
    before(): call(void Point.set*(int))
              && cflow(call(void Line.rotate(double))) {
        setCount++;
}   }
```

Safe Aspect Use

- Aspects will normally preserve *binary compatibility* – method signtures may not be changed, but you may add new code (advice) and new methods and fields
- A client of an aspect-weaved class will be able to use it as before
- But there's nothing stopping you from making incompatible behaviour changes (e.g., start returning 4 instead of 2)
  - this is difficult to reason about, and is probably a bad idea!
  - aspects should preserve the 'essential behaviour' of the program – behaviour may change (that's the point), but the main results should be the same (preserve invariants / axioms / specification)
  - changing the error behaviour – signalling an error instead of returning wrong results, for example – is a different matter
  - some programs may not work at all without aspects

Domain-Specific Aspect Languages

- A *domain-specific language* (DSL) is a language specially tailored for a domain. It may lack general programming capabilities, but problems within the domain can be solved with less, and probably clearer code.
- A *domain-specific aspect language* (DSAL) is a DSL for cross-cutting concerns
  - Some useful cross-cutting concerns can't be expressed as aspects in general aspect languages (e.g., AspectJ) – they're sort of "cross-cross-cutting concerns"
  - Sometimes you want your aspects "pre-packaged" in a user-friendly way

Example: Handling Errors and Partiality in Programs

- Errors can be reported in many ways – exceptions, return codes, global flags, ...
- They can also be handled in many ways – ignore, crash the program, try again, substitute default, ...
- Handling errors is closely tied to the reporting mechanism
- Aspects can let you specify policies for handling errors (depending on how they are reported), but this can be cumbersome if you need fine-grained control

The Alert DSAL

- Declare the error reporting mechanism together with the method:

```
int f(int x)
  pre x < 0 alert ParameterError
  post value == -1 alert Aborted
```

- Declare policies, either globally

```
on ParameterError in * {
  System.out.println("Fatal Error!");
    exit(1);
}
```

or locally:

```
int x = f(5) <:Aborted: 0; // use 0 if Aborted
```

The library+notation model for DSAL implementation

- Typically, you implement a DSL as a library in an existing language (e.g., Java or C++).
- To get nice domain syntax, you can make a simple preprocessor that translates your syntax to library calls (tools like SDF2 can help you with this). Some languages like Dylan or Scheme have macro systems that can do most of this for you.
- Aspects, however, are cross-cutting, so you can't implement them as normal Java/C++/etc libraries.

The library+notation model for DSALs (cont'd)

- You can however write them as libraries in an aspect language (if powerful enough) or a program-transformation language (such as Stratego). With nice syntax on top and a simple preprocessor, you've got a DSAL.
- "Cross-cross-cutting concerns" like Alerts will typically need some global analyses, and can't be implemented by simple translation to existing aspect languages – you need the power of a general-purpose system

Related Techniques

- Dynamically scoped functions (e.g., in Lisp)
- Subclassing (e.g., in Java)
- Program Transformation and Meta-Programming
- Open Classes
- Clone & Adapt
- COME FROM (e.g., in Intercal)

Summary

Aspects...

- provide separation of cross-cutting concerns through
  - advice
  - join points
  - pointcuts
- should be kept "conservative", without incompatible or unexpected code changes
- may not be right for you...
  - some programmers find them confusing
  - may both simplify and complicate code auditing
  - may be too weak compared to meta-programming systems
- are usable for debugging, safety, convenience and maintenance
- are available for lots of languages (Java, C++, Python, Lisp, Stratego, Ruby, Cobol, PHP, ...)

Links and references

- Bergen:
  - SAGA: `http://www.ii.uib.no/saga/`
  - Multicore: `http://www.ii.uib.no/multicore/`
  - Mouldable: `http://www.ii.uib.no/mouldable/`
- AspectJ: `http://eclipse.org/aspectj/`
- Aspect-Orientation Conference: `http://aosd.net/`
- DSAL: `http://dsal.dcc.uchile.cl/2008/`
- Costanza, "Dynamically scoped functions as the essence of AOP", SIGPLAN Notices, Aug 2003
- Bagge and Kalleberg, "DSAL = library + notation", DSAL 2006.

Abstract

Separation of concerns is the idea of breaking down a program into encapsulated pieces that overlap in functionality as little as possible. Encapsulated entities, such as classes, methods or modules, are more manageable, easier to test and maintain, and may be reused more easily than a large, entangled program. A *cross-cutting concern* is something that cannot be encapsulated using normal abstraction mechanisms, thus defeating separation of concerns. A classical example of this is logging (e.g., logging calls and returns to a file while the program is running) – the logging code needs to be added to every applicable method in the program. The logging code for each method may be almost identical, creating an undesirable overlap in functionality. *Aspects* let a programmer implement a cross-cutting concern as a separate entity, through *advice* (how a concern should be implemented) and *join points* (where it should be implemented). I will give an introduction to aspect-orientation and aspect languages, and also talk a bit about *domain-specific aspect languages*.