Aspicere: AOP for C
– A Primer –

Bram Adams
bram.adams@ugent.be

October 30, 2005

Contents

1 Introduction

Aspect Oriented Programming (AOP for short) is a rather young programming paradigm, going beyond Object Oriented Programming's (OOP) modularisation limitations. As OOP was more or less the successor of procedural programming languages like C, Pascal, Cobol, Fortran, ... and as such borrows a lot of their concepts, it's no bad idea to propagate AOP's ideas back to them. This is also motivated by the fact that the procedural programming style still dominates the programming world, such that AOP in legacy languages even becomes economically viable. In the remainder of this tutorial, we suppose that the reader (you!) has some coarse understanding of what AOP is all about as well as the basic concepts. If not, there's a wealth of information available (e.g. [7] and [6]).

So, what does Aspicere offer you? It's an aspect language for C, started as a spin-off of Cobble [8], an aspect language for Cobol, to generalize its framework to other languages. Originally, Aspicere was influenced by AspectC [4] that had borrowed its syntax and semantics from AspectJ, the best known aspect language (targeted to Java). As argued in [5, 1], AspectJ's dependence on naming conventions limits the expressivity of the pointcut language, resulting in fragile aspects.

So, we opted for a dual aspect language, containing a fixed advice syntax with flexible Prolog pointcuts. Advice code is pure C with logic variables in it. We'll explain this terminology using the example application in 2, and the two crosscutting concerns of 3 and 4. More documentation can be found on our website[1].

[1] `http://users.ugent.be/~badams`

2 Our example application

We consider a simple database C application in which a company's employees are paid their wages[2]. Fig. 1 shows the essential two methods. The `_iqc*()`-methods are offered by an external SQL-library, whose interface is shown in Fig. 2. Looking at Fig. 1, we see that method `monthlyPayment()` does most of the work. There's an initial query to the database, followed by cursoring through the fetched data and closing the cursor. In each loop there's also a call to `displayName()`, which performs a separate SQL query to display the name of the relevant employee.

The crucial difference between both methods[3], is that `monthlyPayment()` is much more critical than `displayName()`. The former quits the entire program (drastic exception handling) if something goes wrong, while the latter just prints "UNAVAILABLE" on the screen and continues in case of trouble. To illustrate, the output of our example program is given in Fig. 3[4]. Although there should be five people paid, only two really are. Also notice the lack of information about the actual error encountered.

3 Our first crosscutting concern: logging (surprise!)

As a first illustration of Aspicere's power, we'll add logging to our base program. The easiest way to do this, is to add `printf()`-statements in every method. This is not only very tiresome, but we have to log calls to our (external!) SQL-library at every call site. It's easy to forget certain events.

It should be clear by now that the logging concern crosscuts the main payment concern. We notice both:

code scattering nearly identical `printf()`-statements in every method

code tangling in a certain method logging code is intermingled with payment code

We'd really want to put everything involved with logging into a separate module, making code maintenance and evolution much easier. AOP allows this, by encapsulating the whole logging concern into an aspect. Fig. 4 shows a possible implementation as an aspect.

It's easy to see that an aspect is just a plain C module, but enriched with some AOP-constructs[5]. As such, it can contain static/global variables and functions. Aspicere's advice constructs, are made up of:

advice signature Our advice implicitly denotes so-called "around"-advice, i.e. advice that can decide whether and when the execution of the advised join point should continue. This can be considered somehow as a special C method, invoked automatically when some condition is fulfilled (see pointcut). That's why we opted for a normal method-like signature, containing a return type, name, binding list[6] and current join point name[7]. The last two could be considered as a meta-argument list with "arguments" representing types, strings, ... and as a name of the join point we want to advise respectively.

pointcut This is a logic query selecting the join points which should be advised by the current advice and binding the logic variables of the binding list to a corresponding value[8]. It's in fact a conjunction or disjunction of Prolog predicates with some syntactic sugar to resemble more closely C-style operators like `&&`, `||` and `!`. The predicates used in Fig. 4 are defined in separate Prolog modules provided with the system. Nothing prevents the user from

[2]The source code of this example can be found in the file $ASPICERE/sandbox/database/*/main.c.

[3]One can argue why the two SQL-queries aren't performed at the same time, but we did this for educational purposes.

[4]Any references to real people are accidentally and not making any sense.

[5]Currently, we don't provide inter-type declarations like addition of extra fields to a struct, enum or union.

[6]These logic variables have names starting with uppercase.

[7]Analogously.

[8]Notice that we currently don't deal with dynamic bindings, i.e. bindings whose value relies on run-time information. This is future work.

```
1  int monthlyPayment(int lowerID,int upperID){
2    sqcursor* cursor=_iqcprep("select␣ID,wage,accountnr␣from␣accounting␣
       where␣%d␣<=␣ID␣<=␣%d",lowerID,upperID);
3    sqlda odesc;
4
5    int id;
6    int counter=0;
7    int amount;
8    char* rekNr;
9
10   while(_iqcftch(cursor, &odesc)){
11     if(sql_code!=0){
12       printf("Oops,␣some␣error␣occurred.␣Abandoning␣after␣%d␣successful
           ␣fetches␣out␣of␣%d...\n",counter,upperID-lowerID+1);
13       exit(0);
14     }
15
16     counter++;
17     id=odesc.id;
18     amount=odesc.wage;
19     rekNr=odesc.accountnr;
20     displayName(id);
21     transfer(amount,rekNr);
22     printf("-----------------\n");
23   }
24
25   _iqcfin(cursor);
26
27   return counter;
28 }
29
30 void displayName(int ID){
31   sqcursor* cursor=_iqcprep("select␣ID,name␣from␣population␣where␣ID␣=␣
       %d",ID);
32   sqlda odesc;
33   char* name;
34
35   _iqcftch(cursor, &odesc);
36   printf("ID:\t%d\n", ID);
37   if(sql_code!=0){
38     printf("Name:\tUNAVAILABLE\n");
39   }else{
40     name=odesc.name;
41     printf("Name:\t%s\n", name);
42   }
43
44   _iqcfin(cursor);
45 }
```

Figure 1: The two most important methods of our example data-driven application.

```
1  int sql_code;
2
3  typedef struct _sqcursor{
4    int next_index;
5    int upper_index;
6    char* query;
7  }sqcursor;
8
9  typedef struct sqlda{
10   int id;
11   double wage;
12   char* name;
13   char* accountnr;
14 }sqlda;
15
16 sqcursor* _iqcprep(char* query,...);
17 int _iqcftch(sqcursor *cursor, sqlda *odesc);/*return current ID*/
18 void _iqcfin(sqcursor *cursor);
```

Figure 2: Interface of the SQL-library in use.

```
ID:     7
Name:   Theo D'hondt
Wiring 2010 Euro to account 437123123456.
----------------
ID:     8
Name:   Kris De Schutter
Wiring 1234 Euro to account 979123123456.
----------------
Oops, some error occurred. Abandoning after 2 successful fetches out of 5...
```

Figure 3: Output of base program.

```
#include <stdio.h>

static int depth = 0;

static void indent () {
  int ax;

  for (ax = 0; ax < depth; ax++)
    {
      printf ("  ");
    }
}

void advice logging_void (FName) on (Jp):
 call(Jp,FunctionName,_)
 && type(Jp,"void")
 && stringify(FunctionName,FName) {
  indent ();
  printf ("before %s\n", FName);

  depth++;
  proceed ();
  depth--;

  indent ();
  printf ("after %s\n", FName);

  return;
}

ReturnType advice logging_nonvoid (ReturnType,FName,RName) on (Jp):
 call(Jp,FunctionName,_)
 && type(Jp,ReturnType)
 && !str_matches("void",ReturnType)
 && stringify(FunctionName,FName)
 && stringify(ReturnType,RName) {
  ReturnType i;

  indent ();
  printf ("before %s : %s\n", FName, RName);

  depth++;
  i = proceed ();
  depth--;

  indent ();
  printf ("after %s : %s\n", FName, RName);

  return i;
}

int advice printf_indentation () on (Jp):
 call(Jp, "printf",_){
  indent ();
  printf ("## ");
  proceed ();

  return;
}
```

Figure 4: Aspect model of our logging concern.

implementing and using his own predicates to compose more concrete pointcuts. The Prolog layer also allows access to a whole range of (meta)data-containers (at weave-time).

advice body This is pure C code, except for appearances of the logic variables[9] of the binding list and the current join point name, as well as occasional calls to `proceed()` and manipulation of a struct named "thisJoinPoint". The former lets the normal execution flow continue (perhaps first to the next advice in the chain), the latter provides all kinds of information about the currently advised join point.

So, in Fig. 4 we have two kinds of advice, the pure logging ones and a more layout-related one. Advices `logging_void` and `logging_nonvoid` provide the (indented) logging functionality for methods returning void and others respectively. Notice the use of a logic variable (`ReturnType`) on line 31 to hide the specific return type of `logging_nonvoid`, illustrating the fact that due to the lack of inheritance-like relationships in C (non-OO!) these logic variables are invaluable to yield a useful aspect language[10]. Advice `printf_indentation` marks the base program's output to differentiate it from the logging output.

Switching to your favorite console window, you can type in the commands of Fig. 5 and get the output shown. We can see that Aspicere acts as a preprocessor to GCC. It gets an input base file (main.c), one at a time, and a file listing all applicable aspects (aspects.lst). The latter mechanism allows easy (un)plugging of aspects in the build process. As a result, one gets a woven version of the base file (main-woven.c). These woven programs are then compiled by GCC the usual way. *Note that weaving an already woven file should be avoided at all costs!* Aspect modules are transformed into woven versions (logging-woven_ac.c) and need to be included into the compilation process.

In short, the weaving process consists of transforming advice into real methods of the woven aspects (who retain their pure C variables and functions) and inserting the right method calls and `#include`-statements in the woven base program. Sometimes this approach cannot be incorporated easily into existing legacy applications' build systems. That's why we also provide a "legacy" mode, in which the advice instantiations are inserted in every base module, such that the aspect modules don't need to be linked in separately. Sharing of state between all advice of a module or even within different executions of a single piece of advice by way of static/global variables becomes impossible this way. We need to investigate how to overcome these problems, or (equivalently) how to fit the full Aspicere more easily into existing build systems.

4 More advanced concern: recovering from trivial database errors

Although our developers are pleased by being freed of inserting logging statements as they go, our company's manager isn't. The fault-intolerance of the application is striking, as will be the employees if they don't get their money. Now, even if a database is really buggy, there are always some errors which can be ignored or are only occurring temporarily. Retrying erring SQL-queries could solve quite some errors, but this is only really needed for `monthlyPayment()`. For our database administrator (DBA), it's also interesting to know which errors disturb normal functioning of the payment application. Of course, we don't want to lose the logging concern either.

We settle with the additional aspect of Fig. 6, implementing the recovery concern together with the Prolog module depicted in Fig. 7, which also needs the Prolog-library (written in Java) of Fig. 8 and the properties file of Fig. 9.

[9] Again, please take note that these logic variables should have names starting with uppercase!

[10] Indeed, without these logic variables, we'd need to write down separate advice definitions `logging_int`, `logging_char`, ... for every possible return type.

4.1 Recovery metadata

Let's start at the beginning (Fig. 7). Our current database vendor provided the DBA with an (electronic) fact sheet of error codes and their meaning. Using a simple (Perl) script, this sheet could be transformed into a collection of Prolog facts like `sql_code(-666,'...')`. For some reason, there are also some error codes explained in property files like Fig. 9. Because our Prolog engine[11] is written in Java, it allows us to define Prolog predicates using Java. This makes it easy to hide the reading of a properties file behind a predicate, as illustrated in Fig. 8. Although the implementation is TuProlog-specific, it's quite obvious: A Java class extending Library, groups a collection of predicates with names ending in "_arity". There are also Prolog-specific data structures, but this is out of scope for this tutorial. We need to make sure however that the library is loaded in our Prolog module (line 1 in Fig. 7).

Based on his expertise, the DBA assigns a number of recovery attempts to the most relevant error codes (`sql_redo(-666,2)`). Again, there could be property files with additional information, or perhaps a database could be used, but there's no Java predicate implemented yet (left as an exercise to the reader ;-)).

The data we specified thusfar, is so-called metadata ("data about data"). It acts as some sort of configuration data for the recovery concern during the weaving process.

4.2 Recovery aspect

Now that most of the needed metadata is provided, we can concentrate on the recovery aspect of Fig. 6. Basically, we want to create a chain of filters, each looking for a specific database error code. If no recovery is needed, control flow is passed to the previous filter, otherwise a new database query is performed. Fig. 10 shows our intentions. We'll first describe the recovery logic, then we discuss the pointcut construction.

For a particular error code E (with its description D and number of attempts A), we want to implement the following algorithm:

1. Initialize variable `currentAttempt` on 0 (we didn't invoke `_iqcftch()` yet).
2. If `currentAttempt` is less or equal to A (there are still attempts left), call `proceed()` in order to go to the next recovery filter (back to 1), the next other advice or the `_iqcftch()`-call. In the last two cases, control will eventually return to the last recovery filter.
3. If error E has occurred, then:
   (a) If `currentAttempt` equals A (last attempt failed), we print out a failure message and exit.
   (b) Otherwise, we print an error message containing D, increment `currentAttempt` and announce a retry attempt (back to 2).
4. Otherwise, we continue normal execution of the control flow (aka natural exit). If this was a recovery attempt, then we print out success of this attempt on the screen and go back to the previous recovery filter if there is one (back to 3).
5. Database query was successful.

This is easy to implement, as our advice code is largely plain C code (lines 10—31 in Fig. 6). However, we need some binding variables, specific to every single error code: `ErrorCode` (E), `ErrorString` (D) and `Iterations` (A). As these depend on the metadata provided by the DBA, we should bind their value in the pointcut.

Besides the three mentioned logic variables, our pointcut must localize the right join points in the base program. Because our SQL-library only has one fetch method, the `recover`-advice must only apply to calls to `_iqcftch()`. As we're tied to this library due to a restrictive license, this won't

[11] `http://lia.deis.unibo.it/research/tuprolog/`

change in the near future and we literally use this method's name in the pointcut. Alternatively, this could be put into metadata[12].

We have seen that while the `monthlyPayment()`-method is very critical, the `displayName()`-method isn't. We want to reflect this in the recovery concern too, so we need some extra meta-information in our pointcut (lines 13—19 in Fig. 7). We define a critical call as being a method invoked in the body of a critical method, of which the `monthlyPayment()`-method is definitely an example (the only one currently). Combining all this, we get the pointcut of Fig. 6 (lines 5—9).

5 TODO

- binding of args
- thisJoinPoint-struct

References

[1] Bram Adams and Tom Tourwé. Aspect Orientation for C: Express yourself. In 3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD 2005, 2005.

[2] Bram Adams. Language-independent aspect weaving. Extended abstract, GTTSE '05 Summer School (Braga).

[3] Bram Adams, Kris De Schutter and Andy Zaidman. AOP for Legacy Environments, a Case Study. In EIWAS '05, 2005.

[4] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. 26(5):88-98, 2001.

[5] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In AOSD '03: Proceedings of the 2nd international conference on Aspect-Oriented Software Development, pages 60-69, New York, NY, USA, 2003. ACM Press.

[6] Gregor Kiczales et al. An overview of AspectJ. In Proceedings of ECOOP '01, LNCS 2072, 2001.

[7] Ramnivas Laddad. I want my AOP!, part 1 [electronic version]. Available (on 17th of August 2005) at http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html, 2003.

[8] Ralf Lämmel and Kris De Schutter. What does Aspect-Oriented Programming mean to Cobol? In AOSD '05: Proceedings of the 4th international conference on Aspect-Oriented Software Development, pages 99-110, New York, NY, USA, 2005. ACM Press.

[12] It's important to make wise decisions as to whether certain information is hardcoded/wildcarded in a pointcut or put aside in metadata. As a rule of thumb, metadata should contain more semantic data, i.e. things that can't be captured easily/correctly by (wildcarded) name patterns.

```
~$ aspicere -verbose -i main.c -o main-woven.c -aspects aspects.lst 2&> /dev/null
~$ gcc -g -I$ASPICERE/lib/runtime -o db $ASPICERE/lib/runtime/_aspicere.c main-woven.c \
> sql.c logging-woven_ac.c 2&> /dev/null
~$ ./db
before monthlyPayment : int
  before _iqcprep : sqcursor*
  after _iqcprep : sqcursor*
  before _iqcftch : int
  after _iqcftch : int
  before displayName
    before _iqcprep : sqcursor*
    after _iqcprep : sqcursor*
    before _iqcftch : int
    after _iqcftch : int
    before printf : int
      ## ID:    7
    after printf : int
    before printf : int
      ## Name:  Theo D'hondt
    after printf : int
    before _iqcfin
    after _iqcfin
  after displayName
  before transfer
    before printf : int
      ## Wiring 2010 Euro to account 437123123456.
    after printf : int
  after transfer
  before printf : int
    ## ----------------
  after printf : int
  before _iqcftch : int
  after _iqcftch : int
  before displayName
    ...
  after displayName
  before transfer
    before printf : int
      ## Wiring 1234 Euro to account 979123123456.
    after printf : int
  after transfer
  before printf : int
    ## ----------------
  after printf : int
  before _iqcftch : int
  after _iqcftch : int
  before printf : int
    ## Oops, some error occurred. Abandoning after 2 successful fetches out of 5...
  after printf : int
  before exit
```

Figure 5: Output of logged base program.

```
1  #include <stdio.h>
2  #include "sql.h"
3
4  int advice recover(ErrorCode, Iterations, ErrorString) on (Jp):
5   call(Jp,"_iqcftch",_)
6   && critical_call(Jp)
7   && sql_redo(ErrorCode,Iterations)
8   && sql_code(ErrorCode,ErrStr)
9   && stringify(ErrStr,ErrorString) {
10    int res=0;
11    int i;
12
13    for(i=-1;i<Iterations;i++){
14      res=proceed();
15
16      if(sql_code==ErrorCode){
17        if(i<Iterations-1) {
18          if (i == -1)
19            fprintf(stderr,"!! Error (%d): %s\n", ErrorCode, ErrorString)
                ;
20          fprintf(stderr,"!! Retrying (attempt %d/%d)...\n",(i+2),
                Iterations);
21        } else {
22          fprintf(stderr,"Retry limit reached. Bailing out.\n");
23        }
24      } else {
25        if (i > -1)
26          fprintf(stderr,"!! Retry %d was successful.\n", (i+1));
27        break;
28      }
29    }
30
31    return res;
32  }
33
34  int advice showsqlcodes() on (Jp):
35   call(Jp,"_iqcftch",_) {
36    int res=proceed();
37
38    if (sql_code != 0)
39      fprintf (stderr,"****** %d ******\n", sql_code);
40    return res;
41  }
```

Figure 6: Aspect model of our recovery concern.

```
1  :- load_library('KnowledgeLibrary').
2
3  sql_code(-666,'The␣statement␣cannot␣be␣executed,␣because␣a␣utility␣or␣a
       ␣governor␣time␣limit␣was␣exceeded.').
4  sql_code(-910,'The␣object␣cannot␣be␣used,␣because␣an␣operation␣is␣
       pending.').
5  sql_code(N,Message):-
6      readSQLCodeFromFile(N,Message).
7
8  sql_redo(-666,2).
9  sql_redo(-357,3).
10 sql_redo(N,Times):-
11     readSQLRedoFromFile(N,Times).
12
13 critical_method('monthlyPayment').
14
15 critical_call(Jp) :-
16     enclosingMethod(Jp,EncMethod),
17     name(EncMethod, Name),
18     critical_method(Name),
19     print('CRITICAL␣call␣found!!!'),nl.
```

Figure 7: User-provided metadata and Prolog rules.

```
1  public class KnowledgeLibrary extends Library {
2      /**
3       *
4       */
5      public KnowledgeLibrary() {
6          super();
7      }
8
9      /**
10      * readSQLCodeFromFile(-N,-Message): read SQL code from property
            file
11      *
12      * @param jp
13      * @param methodNode
14      * @return
15      */
16     public boolean readSQLCodeFromFile_2(Term N,Term Message) {
17         Properties prop=new Properties();
18         try {
19             prop.load(new BufferedInputStream(new FileInputStream("/
                   Users/bram/workspace/aspicere/branches/only_parse_to_xml
                   /sandbox/database/recovery/prop.txt")));
20         } catch (FileNotFoundException e) {
21             e.printStackTrace();
22         } catch (IOException e) {
23             e.printStackTrace();
24         }
25         Iterator it=prop.keySet().iterator();
26         if(it.hasNext()){
27             String key=(String)it.next();
28             String value=(String)prop.get(key);
29             System.out.println("KEY:\t<"+key+">␣of␣"+key.getClass().
                   getName()+"\t<"+value+">␣of␣"+value.getClass().getName()
                   +"");
30             return unify(N,TUPrologDynamicMapping.mapToTuProlog(new
                   Integer(key), getEngine()))
31                 && unify(Message,TUPrologDynamicMapping.mapToTuProlog(
                       value, getEngine()));
32         }
33
34         return false;
35     }
36 }
```

Figure 8: User-created TuProlog library.

```
-357=The file server is currently not available.
```

Figure 9: Metadata stored in properties file.

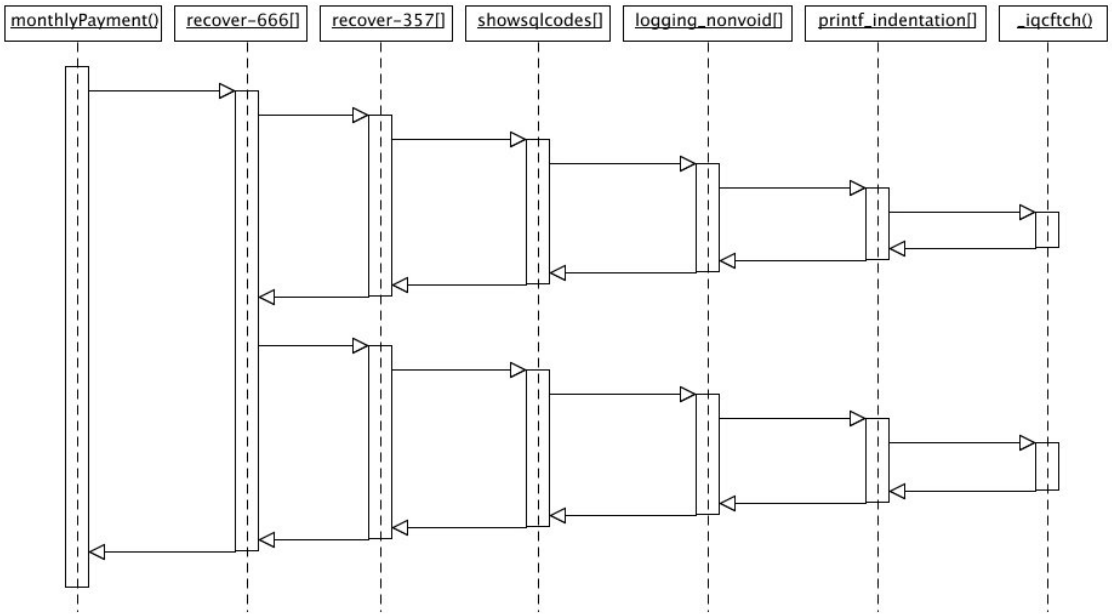

Figure 10: Sequence diagram showing flow of control for the example application of Fig. 1, the logging aspect of Fig. 4 and the recovery aspects and metadata of Fig. 6 to 9. To get the sequence shown, the logging aspect is woven in after the recovery aspect, and the -666 error occurred at run-time (recovered successfully during the first recovery attempt). Instead of objects, we show methods (name ends in parentheses) and advice instances (brackets at the end of the name).