Face-off: AOP+LMP vs. legacy software

Kris De Schutter, Bram Adams

{Kris.DeSchutter,Bram.Adams}@UGent.be
Ghislain Hoffman Software Engineering Lab
INTEC, Ghent University, Belgium

ABSTRACT

This paper applies a mix of aspect-oriented programming (AOP) and logic meta-programming (LMP) to tackle some concerns of/in legacy environments. We present four different problems, and illustrate —with code— how far AOP+LMP gets us. The legacy environments subjected to this treatment encompass the two major players: C, and (of course) Cobol. The aspect code is based on, respectively, Aspicere and Cobble, two aspect languages (being) developed by the authors.

1. INTRODUCTION

Modern business applications are characterized by a need for restructuring and integration, and this at a much larger scale than ever before [8, 12]. This follows from the restructuring and integration of organizations themselves, as they, among other things, strive to merge their activities and, hence, their ICT infrastructure.

In doing so, there is one major obstacle which keeps popping up: *legacy systems*. The term *legacy* brings with it an intuitive understanding of the problem, yet it is hard to find a single, fitting definition. Bennet defines legacy systems informally as,

> "*large software systems that we don't know how to cope with but that are vital to our organisation.*" [1]

Brodie and Stonebraker describe it as,

> "*any information system that significantly resists modification and evolution to meet new and constantly changing business requirements.*" [3]

Nicholas Gold puts it as follows,

> "*Legacy software is critical software that cannot be modified efficiently. A legacy system is a socio-technical system containing legacy software.*" [6]

More definitions exist, but we will focus on two properties shared between them:

1. The software (or system) is in a state where any modification requires a large (disproportionate) amount of effort.
2. The software (or system) is of such value to its stakeholders that they can not put it aside.

This paper will now present a few of these problems, and discuss whether a mix of aspect-oriented programming (AOP) and logic meta-programming (LMP) may bring relief. Our discussion will take place in the context of the two major languages in legacy environments: C and Cobol. The aspect languages used here are Aspicere and Cobble. We choose not to re-iterate the design of these —preferring to point the reader to [13, 7]—, but will present the necessary background when needed.

```
static FILE* fp;

Type around tracing (Type) on (Jp):
  call(Jp,"ˆ(?!.*printf$|.*scanf$).*$")
  && type(Jp,Type) && !is_void(Type)
{
  Type i;

  fprintf (fp, "before␣(␣%s␣in␣%s␣)\n",
       Jp->functionName, Jp->fileName);

  i = proceed ();

  fprintf (fp, "after␣(␣%s␣in␣%s␣)\n",
      Jp->functionName, Jp->fileName);

  return i;
}

Type around cleanup (Type,Name) on (Jp):
  execution(Jp,"main")
  && type(Jp,Type) && !is_void(Type)
  && logfile(File) && stringify(File,Name)
{
  Type i;

  fp = fopen (Name, "a");
  i = proceed ();
  fclose (fp);

  return i;
}
```

Figure 1: Part of a generic tracing aspect.

2. OBLIGATORY TRACING

Tracing is one of the classic examples of (non-functional) crosscutting concerns, due to its extremely scattered nature, and corroborated by its apparent simplicity. In legacy environments however, it might actually prove useful too, as such dedicated tools as DTRACE [5] do not necessarily exist there.

2.1 The reasoning behind AOP+LMP

What sets procedural programming apart from object-orientation (OO), is its primitive support for subtyping. Whereas classes in Java are related to each other by inheritance and implementation relationships, and whereas C++ depends on its powerful template mechanism, C has none of this. Admittedly, one can always cast to `void`-pointers, but this is not as safe as OO-mechanisms.

Such limitations severely impact the introduction of AOP: writing one generic advice capable of advising calls to procedures with varying parameter lists and return types, is impossible without blessing Aspicere with reflective powers. Cobol, which does not even have the "luxury" of void-pointers, needs this even more.

We can access these reflective capabilities through the application of logic meta-programming (LMP), complemented with a simple template mechanism. Briefly, basing the pointcut language on a logic language (such as Prolog) gives access to a powerful unification mechanism, allowing binding of data during the matching phase of the pointcut. This way, information encountered at each individual advised join point can be bound to logic variables (or "bindings" for short). Using these bindings both in the advice's signature as well as in the advice body, solves the problem of writing generic advice. For more information on this, see [7, 13].

2.2 A generic tracing aspect . . .

In figure 1, we have shown part[1] of a generic tracing aspect written in Aspicere. The idea is to trace calls to all procedures except for the printf- and scanf-families (line 4) and stream output into a file (fp, declared on line 1) before and after each call (lines 9 and 14). Opening and closing of the file pointer on line 1 is achieved by advising the main-procedure (line 21).

Aspects are encapsulated in plain compilation units able to hold advice constructs. Advice itself features a signature (lines 3 and 20), a pointcut (lines 4–5 and 21–23) and a body (lines 7–17 and 25–31). The advice body is written in C, with additions as explained in the previous subsection.

In figure 1, for example, the return type of the advised procedure call is bound on lines 5 and 22. These bindings are then used in the advices' signatures (lines 3 and 20) as well as in their bodies (lines 7 and 25). This way, the tracing advice is not limited to one particular type of procedures, and the file pointer management is oblivious to the specific signature of the main procedure. The well-known thisJoinPoint construct from AspectJ-like languages, can also be accessed through a join point-specific binding (Jp on lines 3 and 20) and used as such (lines 10 and 15).

Applied to reverse-engineering contexts, using LMP and a template mechanism allows non-invasive and intuitive extraction of knowledge hidden inside legacy systems, *without* prior investigation or exploration of the source code [13]. One does not first have to extract all available types and copy the tracing advice for all of them, as was experienced in [4].

2.3 . . . crosscuts the build system

In [13], we applied this tracing aspect to a large case study (453 KLOC) to enable dynamic analyses. Although we did not need to delve into the source code (thanks to the generic advice), integrating Aspicere's weaving framework into the existing build system did prove more of a problem.

As source code is the most portable representation of C programs across several platforms, Aspicere's weaver transforms base code and aspects into woven C code and as such acts as a preprocessor to a normal C compiler. Because the original makefile hierarchy drives the production of object files, libraries and executables, using a myriad of other tools and preprocessors (e.g. embedded SQL), and all of these potentially process advised input, it turns out that Aspicere's weaver crosscuts the makefile system. We therefore need to find out what is produced at every stage of the build and unravel accompanying linker dependencies.

Although one does not need to delve into source code for applying AOP to a legacy system, one does need to find out about the build system in use. First, an inventory of the included tools, and their interplay, has to be made. For each of these we must then find a way to plug in the weaver.

[1]We do not show advice for void procedures, as these are equivalent to the advices shown, less the need for a temporary variable to hold the return value.

More specifically, Aspicere's weaver needs one preprocessed input at a time and its output will be another tool's input. Additionally, the normal weaving habit is to transform aspects into genuine C compilation units by converting the advices into (multiple) procedures. This enables the normal C visibility rules in a natural way, i.e. the visibility of fp on figure 1 is tied to the module containing the aspect. To accomplish this modularisation, we need to link this single transformed aspect into each advised application.

In case all makefiles are automatically generated using, for instance, automake, one could try to replace (i.e. alias) the tools in use by wrapper scripts which invoke the weaving process prior to calling the original tool. The problem here is that this is an all-or-nothing approach. It may be that in some cases weaving is needed (e.g. a direct call to gcc), and in others not (e.g. when gcc is called from within esql). Making the replacement smart enough to know what to do when is not a trivial task.

In the case of [13], many of the 267 makefiles were indeed generated. Still, some were manually adapted afterwards, while others were written from scratch. Due to issues with the embedded SQL preprocessor and the irregular presence of certain environment variables, we wrote some scripts to directly alter the makefiles and "weave in" the right operations on Aspicere's weaver instead of aliasing the tools themselves. However, detecting where exactly our tool failed (due to the heterogeneously structured makefiles) and making the necessary manual adaptations still took several hours.

Without intimate knowledge of the build system, it was hard to tell whether source files are first compiled before linking all applications, or (more likely) whether all applications are compiled and linked one after the other. As such, our weaving approach could not be applied. As an ad hoc solution, we opted to move the transformed advice into the advised base modules themselves. This meant that we had to declare fp as a local variable of the tracing advice, resulting in huge run-time overhead due to repeated opening and closing of the file. Using a shared library in the first place seems to be a better solution.

3. BUSINESS RULE MINING

In [9], Isabel Michiels and the first author discuss the possibility of using dynamic aspects for mining business rules from legacy applications. Some suggestions as to how this may be done are presented based on the following fictitious, though realistic case:

> "*Our accounting department reports that several of our employees were accredited an unexpected and unexplained bonus of 500 euro. Accounting rightfully requests to know the reason for this unforeseen expense.*"

We will now revisit this case, showing the actual Cobble advices which may be used to achieve the ideas set forth there.

We start off by noting that we are not entirely in the dark. The accounting department can give us a list of the employees which got "lucky" (or rather unlucky, as their unexpected bonus did not go by unnoticed). We can encode this knowledge as facts:

```
META-DATA DIVISION.
 FACTS SECTION.
   LUCKY-EID VALUE 7777.
   LUCKY-EID VALUE 3141.
   *> etc.
```

Furthermore, we can also find the definition of the employee file which was being processed, in the copy books:

```
DATA DIVISION.
```

```
  FILE SECTION.
3   FD EMPLOYEE-FILE.
    01 EMPLOYEE.
5       05 EID PIC 9(4).
        *> etc.
```

Lastly, from the output we can figure out the name of the data item holding the employee's total salary. This data item, BNS-EUR, turns out to be an edited picture. From this we conclude that it is only used for pretty printing the output, and not for performing actual calculations. At some time during execution the correct value for the bonus was moved to BNS-EUR, and subsequently printed. So our first task is to find what variable that was.

We go at this by tracing all moves to BNS-EUR, but *only while processing one of our lucky employees*:

```
  FIND-SOURCE-ITEM SECTION.
2   USE BEFORE ANY STATEMENT
    AND NAME OF RECEIVER EQUAL TO "BNS-EUR"
4   AND BIND LOC TO LOCATION
    AND IF EID EQUAL TO LUCKY-EID.
6  MY-ADVICE.
    DISPLAY EID, ":␣", LOC.
```

In short, this advice states that before all statements (line 2) which have BNS-EUR as a receiving data item (line 3), and if EID (id for the employee being currently processed; see data definition higher up) equals a lucky id (runtime condition on line 5; similar to AspectJ's if condition), we display the location of that statement as well as the current id.

We now find the possibilities to be one of several string literals (which we can therefore immediately disregard) and a variable named BNS-EOY, whose name suggests it holds the total value for the end-of-year bonus.

Our next step is to figure out how the end value was calculated. This would allow us to check the figures and maybe spot an error. We set up another aspect to trace all statements modifying the variable BNS-EOY, but again only while processing a lucky employee. We do this in three steps. First:

```
1 TRACE-BNS-EOY SECTION.
    USE BEFORE ANY STATEMENT
3   AND NAME OF RECEIVER EQUAL TO "BNS-EOY"
    AND BIND LOC TO LOCATION
5   AND IF EID EQUAL TO LUCKY-EID.
   MY-ADVICE.
7   DISPLAY EID, ":␣statement␣at␣", LOC.
```

Before execution of any statement (line 2) having BNS-EOY as a receiving data item (line 3), and when processing a lucky employee (line 5), this would output the location of that statement. Next:

```
1 TRACE-BNS-EOY-SENDERS SECTION.
    USE BEFORE ANY STATEMENT
3   AND NAME OF RECEIVER EQUAL TO "BNS-EOY"
    AND BIND SENDING TO SENDER
5   AND BIND SENDING-NAME TO NAME OF SENDING
    AND IF EID EQUAL TO LUCKY-EID.
7  MY-ADVICE.
    DISPLAY SENDING-NAME, "␣sends␣", SENDING.
```

This outputs the name and value for all sending data items (lines 4 and 5) before execution of any of the above statements. This allows us to see the contributing values. Lastly, we want to know the new value for BNS-EOY which has been calculated.

```
  TRACE-BNS-EOY-VALUES SECTION.
2   USE AFTER ANY STATEMENT
    AND NAME OF RECEIVER EQUAL TO "BNS-EOY"
4   AND IF EID EQUAL TO LUCKY-EID.
   MY-ADVICE.
6   DISPLAY "BNS-EOY␣=␣", BNS-EOY.
```

We now find a data item (cryptically) named B31241, which is consistently valued 500, and is added to BNS-EOY in every trace. Before moving on we'd like to make sure we're on the right track. We want to verify that this addition of B31241 is only triggered for our list of lucky employees. Again, a dynamic aspect allows us to trace execution of exactly this addition and helps us verify that our basic assumption holds indeed.

We start by recording the location of the "culprit" statement as a usable fact:

```
  META-DATA DIVISION.
2   FACTS SECTION.
      CULPRIT-LOCATION VALUE 666.
4     *> other facts as before
```

The test for our assumption may then be encoded as:

```
  TRACE-BNS-EOY-SENDERS SECTION.
2   USE BEFORE ANY STATEMENT
    AND LOCATION EQUAL TO CULPRIT-LOCATION
4   AND IF EID NOT EQUAL TO LUCKY-EID.
   MY-ADVICE.
6   DISPLAY EID, ":␣back␣to␣the␣drawing␣board.".
```

This tests whether the culprit statement gets triggered during the process of any of the other employees. If it does, then something about our assumption is wrong. Or it may be that the accounting department has missed one of the lucky employees.

Given the verification that we are indeed on the right track, the question now becomes: why was this value added for the lucky employees and not for the others? Unfortunately, the logic behind this seems spread out over the entire application. So to try to figure out this mess we would like to have an execution trace of each lucky employee, including a report of all tests made and passed, up to and including the point where B31241 is added. Dynamic aspects allow us to get these specific traces.

First, some preliminary work:

```
  WORKING-STORAGE SECTION.
2   01 FLAG PIC 9 VALUE 0.
       88 FLAG-SET     VALUE 1.
4      88 FLAG-NOT-SET VALUE 0.
```

The FLAG data item will be used to indicate when tracing should be active and when not. For ease of use we also define two "conditional" data items: FLAG-SET and FLAG-NOT-SET. These reflect the current state of our flag.

Our first advice is used to trigger the start of the trace:

```
  TRACE-START SECTION.
2   USE AFTER READ STATEMENT
    AND NAME OF FILE EQUAL TO "EMPLOYEE-FILE"
4   AND BIND LOC TO LOCATION
    AND IF EID EQUAL TO LUCKY-EID.
6  MY-ADVICE.
    SET FLAG-SET TO TRUE.
8   DISPLAY EID, ":␣start␣at␣", LOC.
```

I.e., whenever a new employee record has been read (line 2 and 3), and that record is one for a lucky employee (line 5), we set the flag to true (line 7). We also do some initial logging (line 8).

The next advice is needed for stopping the trace when we have reached the culprit statement:

```
  TRACE-STOP SECTION.
2   USE AFTER ANY STATEMENT
    AND LOCATION EQUAL TO CULPRIT-LOCATION.
4  MY-ADVICE.
    SET FLAG-NOT-SET TO TRUE.
6   DISPLAY EID, ":␣stop␣at␣", LOC.
```

Then it is up to the actual tracing. We capture the flow of procedures, as well as execution of all conditional statements:

```
1 DISPATCHING SECTION.
    USE AROUND PROGRAM
3   AND BIND PARA TO PARAGRAPH
    AND BIND PARA-NAME TO NAME OF PARA
5   AND IF METHOD-NAME EQUAL TO PARA-NAME.
   MY-ADVICE.
7   PERFORM PARA.

9 ENCAPSULATION SECTION.
    USE AROUND PROGRAM.
11 MY-ADVICE.
    PERFORM ERROR-HANDLING.
13  EXIT PROGRAM.
```

Figure 2: Aspect for procedure encapsulation.

```
TRACE-PROCEDURES SECTION.
  USE AROUND PROCEDURE
  AND BIND PROC TO NAME
  AND BIND LOC TO LOCATION
  AND IF FLAG-SET.
 MY-ADVICE.
  DISPLAY EID, ":␣before␣", PROC, "␣at␣", LOC.
  PROCEED.
  DISPLAY EID, ":␣after␣", PROC, "␣at␣", LOC.

TRACE-CONDITIONS SECTION.
  USE AROUND ANY STATEMENT
  AND CONDITION
  AND BIND LOC TO LOCATION
  AND IF FLAG-SET.
 MY-ADVICE.
  DISPLAY EID, ":␣before␣condition␣at␣", LOC.
  PROCEED.
  DISPLAY EID, ":␣after␣condition␣at␣", LOC.
```

From this trace we can then deduce the path that was followed from the start of processing a lucky employee, to the addition of the unexpected bonus. More importantly, we can see the conditions which were passed, from which we can (hopefully) deduce the exact cause.

This is where the investigation ends. For those curious, we refer to the original paper for the solution [9]. Whatever the cause of the problem, AOP+LMP provided us with a flexible and powerful tool to perform our investigation.

4. ENCAPSULATING PROCEDURES

In [11], Harry and Stephan Sneed talk about creating web services from legacy host programs. They argue that while there exist tools for wrapping presentation access and database access for use in distributed environments,

> "*the accessing of [...] the business logic of these programs, has not really been solved.*"

In an earlier paper, [10], Harry Sneed discusses a custom tool which allowed the encapsulation of Cobol procedures, to be able to treat them as "methods", a first step towards wrapping business logic. Part of that tool has the responsibility of creating a switch statement at the start of the program, which performs the requested procedure, depending on the method name.

4.1 The basic wrapping aspect . . .

Figure 2 shows how encapsulation of procedures (or "business logic") can be achieved, in a generic way, using AOP and LMP. The aspect shown here, written in Cobble, consists of two advices: one named `DISPATCHING`, the other `ENCAPSULATION`.

The first advice (lines 1–7) takes care of the dispatching. It acts around the execution of the entire program (line 2), and once for every paragraph in this program (line 3). The latter effect is caused by the ambiguousness of the `PARAGRAPH` selector, as it applies to any paragraph. Rather than just picking one, what Cobble does is *pick them all*: the advice gets activated for every possible solution to its pointcut, one after the other.

Furthermore, the `DISPATCHING` advice will only get triggered when `METHOD-NAME` matches the name of the selected paragraph (extraction of this name is seen on line 4). This is encoded in a runtime condition on line 5. Finally, the advice body, when activated, simply calls the right paragraph (`PERFORM` statement on line 7).

The second advice (lines 9–13) serves as a generic catch-all. It captures execution of the entire program (line 10), but replaces this with a call to an error handling paragraph (line 12) and an exit of the program (line 13). The net effect is that whenever the value in `METHOD-NAME` does not match any paragraph name in the program, the error will be flagged and execution will end. This, together with the first advice, gives us the desired effect.

We are left with the question of where `METHOD-NAME` is defined, and how it enters our program. The answer to the first question is simply this: any arguments which get passed into a Cobol program from the outside must be defined in a *linkage section*. I.e.:

```
LINKAGE SECTION.
 01 METHOD-NAME PIC X(30) VALUE SPACES.
```

Furthermore, the program division needs to declare that it expects this data item as an input from outside:

```
PROGRAM DIVISION USING METHOD-NAME.
```

4.2 . . . is hard to wrap up

This begs the question as to how this input parameter was inserted in an AOP-like way. Simply: it was *not*. We tacitly assumed our aspect to be defined *inside* the target program (a so-called "intra-aspect"), which dismissed the need for any added introduction mechanism. Of course, for a truly generic aspect (an "inter-aspect") we need to remedy this.

Definition of the `METHOD-NAME` data item is no big problem. We can simply define it within an aspect module, which, upon weaving, would extend the target program (modulo some alpha-renaming to prevent unintended name capture):

```
IDENTIFICATION DIVISION.
  ASPECT-ID. PROCEDURE-WRAPPING.

DATA DIVISION.
LINKAGE SECTION.
   01 METHOD-NAME PIC X(30) VALUE SPACES.
```

From this, it becomes pretty obvious that `METHOD-NAME` should be used as an input parameter. The concept of a linkage section makes no sense for an external aspect module, as an aspect will never be called in such a way. Indeed, we might even say that it *should not* be used that way. Therefore the appearance of a linkage section is a sufficient declaration of intent.

The hard part lies with the semantics of declaring extra input data items on another program. What do we expect to happen?

- Does the introduction of an input data item by the aspect replace existing input items in the advised program, or is it seen as an addition to them?

- If it is added to them, then where does it go into the existing list of inputs? At the front? At the back?

- What happens when multiple aspects define such input items? In what order do they appear?
- How do we handle updating the sites where the woven program gets called? The addition of an extra input item will have broken these.

Consider the C/Java/... equivalent of this: what does it mean to introduce new parameters on procedures/methods? More to the point, *should* we allow this?

4.3 Full encapsulation

The complexity of the problem increases when we consider another important feature of Sneed's tool (ignored until now):

> "*For each [encapsulated] method a data structure is created which includes all variables processed as inputs and outputs. This area is then redefined upon a virtual linkage area. The input variables become the arguments and the output variables the results.*" [10]

Put another way, we must find all data items on which the encapsulated procedures depend. These are then gathered in a new record (one per procedure), which redefines a "virtual linkage area" (in C terms: a union over all newly generated typedefs). This linkage area must then also be introduced as an input data item of the whole program.

Such a requirement seems far out of the scope of AOP. While it has a crosscutting concern in it (cfr. "for *each* method"), this concern can not be readily defined using existing AOP constructs.

Instead, figure 3 shows a different approach to the problem. It is encoded neither in Cobble or Aspicere, opting for a different view on the AOP+LMP blend. Whereas the previous examples were based on LMP embedded in AOP, figure 3 is based on embedding AOP in LMP, similar to the approach in [2].

The code can be read as follows. Whatever you find enclosed in curly brackets ({...}) is (aspect-)code which is to be generated. This can be further parameterized by placing variables in "fishgrates" (<...>), which will get expanded during processing. Everything else is Prolog, used here to drive the code generation.

Let us apply this to the code in figure 3. Lines 1 and 2 declare the header of our aspect, while lines 4–6 define the linkage section as discussed before. Lines 8–15 calculate all slices (`slice/2` on line 11) for all paragraphs (`paragraph/2` on line 10). From each of these we extract the working-storage section (`wss/2` on line 12), which gives us the required in- and output parameters, collected in `AllInOut` (line 14). From this we extract the size of the largest one (`max_size/2` on line 17) which is used next in the definition of the virtual storage space (line 18).

Next, for each paragraph (i.e. for each member of `AllInOut`), we generate a redefinition of the virtual space to include all data items on which that paragraph depends (lines 20–26). The redefinition can be seen on line 21, where it is given a unique name (i.e. `SLICED-`*paragraph-name*). Its structure is defined by going over all records in the working-storage section for that paragraph (line 22), cloning each record under a new, unique name while updating the level number (line 23), and then outputting this new record (line 24). This concludes the data definition.

Next, the procedure division is put down, declaring the necessary parameters (line 28). We then generate advice similar to figure 2, but now some extra work is needed. First, the data from the virtual storage space as redefined for the paragraph must be transferred to the original records defined for the program (lines 37–39). The original paragraph may then be called without worry (line 40). Afterwards, the calculated values are retrieved by moving them back

```
  { IDENTIFICATION DIVISION.
2     ASPECT-ID. PROCEDURE-WRAPPING.

4   DATA DIVISION.
    LINKAGE SECTION.
6   01 METHOD-NAME PIC X(30) VALUE SPACES. },

8 findall(
    [Name, Para, Wss],
10  ( paragraph(Name, Para),
      slice(Para, Slice),
12    wss(Slice, Wss)
    ),
14  AllInOut
  ),
16
  max_size(AllInOut, VirtualStorageSize),
18 { 01 VSPACE PIC X(<VirtualStorageSize>). },

20 all(member([Name, Para, Wss], AllInOut), (
    {  01 SLICED-<Name> REDEFINES VSPACE.},
22  all( (record(R, Wss), name(R, RName)), (
      clone_and_shift(R, "<RName>-<Name>", SR),
24    { <SR> }
    ))
26 )),

28 { PROGRAM DIVISION USING METHOD-NAME, VSPACE.
    DECLARATIVES. },
30
  all(member([Name, Para, Wss], AllInOut), (
32  { WRAPPING-FOR-<Name> SECTION.
        USE AROUND PROGRAM
34      AND IF METHOD-NAME EQUAL TO "<Name>".
       WRAPPING-BODY.
36  },
    all( (top_record(R, Wss), name(R, RName)),
38    { MOVE <RName>-<Name> TO <RName>.}
    ),
40  {     PERFORM <Name>.}
    all( (top_record(R, Wss), name(R, RName)),
42    { MOVE <RName> TO <RName>-<Name>.}
    )
44 )),

46 { ENCAPSULATION SECTION.
      USE AROUND PROGRAM.
48   MY-ADVICE.
      PERFORM ERROR-HANDLING.
50    EXIT PROGRAM.
    END DECLARATIVES. }
```

Figure 3: Full procedure encapsulation.

to the virtual storage space, again as redefined for the paragraph (lines 41–43). All that is left is the generic catch-all (lines 46–50), and the closing of the aspect (line 51).

Despite the inherent complexity of the problem, AOP+LMP allowed us to write down our crosscutting concern with certain ease. LMP was leveraged to define our aspect by reasoning over the program. AOP was leveraged to tackle the actual weaving semantics, unburdening us from writing program transformations.

Granted, we quite happily made use of a slicing predicate to do most of the hard work (line 11). Still, the use of libraries which hide such algorithms is another bonus we can get from LMP.

5. YEAR 2000 SYNDROME

The Y2K-bug is probably the best-known example of problems related to legacy systems. It is important to understand that at the heart of this was not a lack of technology or maturity thereof, but rather the understandable failure to recognize that code written as early as the sixties would still be around some forty years later.

So might AOP+LMP have helped us out? The problem statement certainly presents a crosscutting concern: whenever a date is accessed in some way, make sure the year is extended.

This presents our first problem: how do we recognize data items for dates in Cobol? While Cobol has structured records, and stringent rules for how data is transferred between them, they carry no semantic information whatsoever. Knowing which items are dates and which are not requires human expertise. The nice thing about LMP is that we could have used it to encode this.

In C, where a disaster is expected in 2038[2] (hence Y2K38), the recognition problem is less serious because of C's more advanced typing mechanisms. A date in (ANSI-)C could be built around the standard time provisions (in "time.h"), or otherwise some (hopefully sensibly named) custom typedef. In the former case, recompiling the source code on a system using more than 32 bits to represent integers solves everything immediately. Whereas all variables in Cobol have to be declared in terms of the same, low-level Cobol primitives, C allows variables to be declared as instances of user-defined types. In this sense, the latter case (custom date type) represents much less of a problem. The check for a date would be equivalent to a check for a certain type.

Second problem for Cobol: given the knowledge of which data items carry date information, how do we know which part encodes the year? It may be that some item holds only the current year, or that it holds everything up to the day. A data item may be in *Gregorian* form (i.e. "yyddd") rather than standard form ("yymmdd"). Of course, that "standard" may vary from locale to locale (the authors would write it as "ddmmyy"). But again, we could use LMP to encode this knowledge.

Let us assume we can check for data items which hold dates, and that these have a uniform structure (in casu "yymmdd"). Then we might write a date expansion aspect using a century window, like:

```
1 AN-YYMMDD-FIX SECTION RETURNING MY-DATE.
    USE AROUND SENDING-DATA-ITEM
3   AND SENDING-DATA-ITEM IS DATE.
   MY-ADVICE.
5   MOVE PROCEED TO MY-DATE(3:8).
    IF MY-DATE(3:4) GREATER THAN 50 THEN
7     MOVE 19 TO MY-DATE(1:2)
    ELSE
9     MOVE 20 TO MY-DATE(1:2).
```

This advice has two problems. One is the definition of `MY-DATE` (referred to as a return value on line 1, and assumed to have a "yyyymmdd" format). In Cobol, all data definitions are global. Hence, `MY-DATE` is a unique data item which gets shared between all advices. While this is probably safe most of the time, it could lead to subtle bugs whenever we have nested execution of such advice.[3] The same is true for all advices in Cobble. Only, in this case, the need for a specific return value makes it surface more easily. Of course, in this case, the fix would be to require duplication of this data item for all advice instantiations.

The greater problem lies in the weaving. When committed to a source-to-source approach, as we are with Cobble, weaving anything below the statement level becomes impossible. As Cobol lacks the idea of functions[4], we can not replace access to a data item with a call to a procedure (whether advice or the original kind) as we could do in C. The remedy for this would be to switch to machine-code weaving, but we are reluctant to do so, as we would lose platform independence. Common virtual machine solutions (e.g. as with ACUCobol) are not widespread either.

[2]More details on `http://www.merlyn.demon.co.uk/critdate.htm`

[3]Though not in this case, as the structure of the advice body *only* refers to the data item *after* the `PROCEED` statement.

[4]Functions can be written in later versions of Cobol. Our focus on legacy systems, however, rules these out for use here.

6. CONCLUSION

We discussed four problems with legacy software, and showed how three of these, might be aided through a mix of AOP and LMP. Tracing in C and business rule mining in Cobol went smoothly, using LMP as a pointcut mechanism in AOP. Encapsulation of procedures in Cobol required a more generative approach, by embedding AOP in LMP.

As for the Y2K problem in Cobol, only very advanced, nearly weaver-level pointcuts in synergy with various cooperating introductions might manage this. As it is, the semantics of Cobol, especially its lack of typing, present too much of a limitation. In C, the Y2K38 problem can still be managed reasonably, precisely because it does feature such typing.

All in all, AOP+LMP proves a useful, flexible and strong tool to tackle the ills of legacy software.

7. ACKNOWLEDGEMENTS

Kris De Schutter received support within the Belgian research project ARRIBA, sponsored by the IWT, Flanders. Bram Adams is supported by a BOF grant from Ghent University.

8. BIBLIOGRAPHY

[1] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995.

[2] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In *GPCE*, pages 110–127, 2002.

[3] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems*. Morgan Kaufmann, 1995.

[4] M. Bruntink, A. van Deursen, and T. Tourwé. An initial experiment in reverse engineering aspects. In *WCRE*, pages 306–307. IEEE Computer Society, 2004.

[5] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[6] N. Gold. The meaning of "legacy systems", 1998.

[7] R. Lämmel and K. D. Schutter. What does Aspect Oriented Programming mean to Cobol? In *AOSD '05*, pages 99–110, New York, NY, USA, 2005. ACM Press.

[8] I. Michiels, D. Deridder, H. Tromp, and A. Zaidman. Identifying problems in legacy software: Preliminary findings of the ARRIBA project. In *ELISA, ICSM'03*.

[9] I. Michiels, T. D'Hondt, K. De Schutter, and G. Hoffman. Using dynamic aspects to distill business rules from legacy code. In *Dynamic Aspects Workshop*, pages 98–102, 2004.

[10] H. M. Sneed. Encapsulating legacy software for use in client/server systems. In *WCRE*, page 104, 1996.

[11] H. M. Sneed and S. H. Sneed. Creating web services from legacy host programs. In *WSE*, pages 59–65, 2003.

[12] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, Berne, 2001.

[13] A. Zaidman, B. Adams, K. De Schutter, S. Demeyer, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. In *CSMR*, 2006.