Mixin Composition Strategies for the Modular Implementation of Aspect Weaving

The EPP preprocessor and its module description language Ld-2

Aspect Oriented Programming workshop at ICSE'98, April 20th, 1998, Kyoto

Yves ROUDIER[1], Yuuji ICHISUGI[2]
{roudier,ichisugi}@etl.go.jp
[1]STA Fellow - Electrotechnical Laboratory, [2]Electrotechnical Laboratory
Computer Science Division
1-1-4 Umezono, Tsukuba, Ibaraki 305 JAPAN

Abstract: EPP (the Extensible Preprocessor) is an extensible language preprocessor; it was designed to introduce new language constructs by the mere addition of plugin modules that can also define new syntaxes and macro definitions. The relation of EPP to AOP is twofold. First, the EPP kit has been programmed using the Ld-2 language which introduces aspect-oriented constructs: system mixins. Second, we suggest that EPP could be used to program aspect-oriented weavers in a modular fashion. We describe here the new Java version of EPP. We also discuss the constructs needed for providing additional capabilities regarding the composition of extension modules that could serve in writing weavers.

1 Issues in AOP

The traditional approach to software programming has been to modelize a problem by defining independent components, be it under the form of objects or functions. Global interactions between components and with the program environment arise, but only implicitly, from their local calls: this is a major drawback when this global interaction (or other mechanisms involving parts of several components) is the most important definition in the software, especially if it must be fine-tuned. For instance, parallel or reactive systems often exhibit these characteristics.

Reflective systems are good at solving some of these issues but do not seem so well adapted for dealing with composition problems or syntactic issues in a simple manner.

These problems get worse when the size of the system grows. The programmer needs to isolate parts of its components in relation with others (untangling). Moreover, introducing specific syntax is very important, would it only be because many formalisms and formal systems have already been developed and could improve the quality of programs.

Aspectual approaches (e.g. [LK97]) directly target these problems and thus shed a new light on possible solutions. However, this approach delegates many problems to the aspect weaver which has to gather aspectual code, process it according to aspect descriptions and generate the final program.

If the size of a system becomes quite large, complexity will also increase. Aspects seem to be somehow domain dependent: if the system grows, it is rather likely that more aspects will be needed to describe it. For now, we can only think of rather simple aspects, but there could be a need for more complex ones as well (although not as complex as the systems they will help to describe). Decomposing the aspect weaver into smaller composable units would both simplify the task of programming a weaver and allow to reuse parts of the aspect description. But to date, a very limited number of tools has been developped that could readily address, in a generic manner, the definition of weavers with such kind of modular units.

2 The module description language Ld-2

We first introduce an object-oriented language, Ld-2 (Language for differential description) [IHT+96, RI97b]. Ld-2 provides a mechanism, *system mixin*,

which enables the program to be split according to different points of view into small reusable modules.

2.1 System Mixins

The first program in Fig.1 defines a method named `m` which contains nested `if` statements. This method is written in a traditional style, in other words, in a "monolithic" way. When a programmer wants to add a new `else-if` clause to this method, the only way to do so is to edit the source code. By using the inheritance mechanism, some editing can be avoided, but this changes the name of the class where the method belongs. It is indeed impossible to extend the behavior of the method without defining a new subclass. If at some other place in the program, there exists the definition `class Bar extends Foo` $\{\cdots\}$ or the definition `new Foo()`, all `Foo` occurrences have to be changed to the name of the subclass.

```
class Foo {
  void m(String d){
    if (d.equals("B")){
      doB();
    } else if (d.equals("A")){
      doA();
    } else {
      doDefault();
    }
}
```

```
SystemMixin Skeleton {
  class Foo {
    define implement void m(String d){
      doDefault();
    }
  }
}
SystemMixin A {
  class Foo {
    extend void m(String d){
      if (d.equals("A")) { doA();} else { original(d);}
    }
  }
}
SystemMixin B {
  class Foo {
    extend void m(String d){
      if (d.equals("B")) { doB();} else { original(d);}
    }
  }
}
```

Figure 1: Method definitions with and without system mixins.

Ld-2 solves this problem by introducing system mixins. Bracha [BC90] showed the flexibility and reusability of *mixin-based* inheritance. System mixins are similar to mixins but the unit of inheritance is not a class but the whole program.

The second program in Fig.1 uses the system mixin feature and has the same meaning as the first one; however, its structure is much more modular. The keyword `SystemMixin` defines a system mixin, that is, a fragment of the program. All system mixins are composed into a complete program before starting up. The expression `original(d)` is similar to the method invocation to the super class in traditional object-oriented languages. When the method `m` is called, the fragment of method `m` defined in `B` will be called. If the fragment evaluates `original(d)`, the next fragment defined in `A` will be called.

If there are subclasses of class `Foo`, the extension of method `m` will be propagated to all of them.

2.2 An AO layout in Ld-2

Among other styles, system mixins allow programming "by difference", a form of aspect-oriented programming.

Intuitively, the API of the application should be defined as a description of small components. The base object-oriented language is very well adapted for expressing these matters. Classes should remain the common reference, especially for teamwork.

After this first description has been written, cross-cutting concepts can be introduced by other system mixins. Classes or parts of classes are distributed among system mixins, then programmed. A system mixin thus roughly corresponds to an aspect or to part of an aspect (either if it is complex or for reuse reasons, in a mixin style).

The choice of classes or parts to be added to a mixin is equivalent to chosing join points in other approaches (decided inside of the aspect weaver). Of course, each system mixin can introduce new join points by defining new classes, which could be regarded as defining aspects of aspects. We generally find it preferable to define a kind of abstract system mixin containing all these classes definitions (but only class definitions). All other system mixins simply introduce additional aspects.

As usual in object-related designs, this double definition of components interface and cross-cutting system mixins should probably follow an iterative design (and should in fact already take place in the requirements specification). The final composition depends on the order of specification of system mixins, mostly decided for dependency reasons, but can also be a deliberate choice of the

Name	Function
AutoSplitFiles	Splits multiple public classes in one file to multiple files.
defmacro	Macro definition with the same style as #define .
EmbedCopyright	Embedding a message into all class files.
enum	Constant integer definition.
OptParam	Optional actual parameters.
UserOp	Operators for user defined data types.
Math	Using java.lang.Math with easier way.
assert	Assert macro.
noassert	Translates assert macros to null statements.
SystemMixin	Supporting programming-by-difference.
Symbol	Same as lisp's symbols.
BackQuote	Same as lisp's back quote macros.

Table 1: Plugins currently provided.

programmer to select features from a special system mixin.

2.3 Related constructs

Flavors and CLOS [Ste90] provide before and after daemon methods that are related to the system mixins because they can add extra behavior to existing methods without changing their class name and method name. However, daemon methods are a more restricted mechanism than system mixins. For example, it is impossible to add multiple before (or after) daemon to one method.

Objective-C has a notion of "category". Using categories, the programmer can add new methods to the existing class hierarchy. A third party programmer can add methods to existing class libraries even if their source code has not been supplied. However, it is impossible to extend existing methods without subclassing.

Some design patterns [GHJV93] which enhance extensibility offer an analogy to system mixins. For example, decorators are units of extension and multiple decorators can be composed simultaneously. However, applying the Decorator pattern requires explicit method delegation which is not required by the system mixin construct. In addition, the Decorator pattern requires a fixed interface for decorators. Therefore, flexibility is worse than system mixins. Of course the comparison is difficult because decorators are just patterns and not language constructs. Decorators also define the way they can be dynamically composed during the execution of the program. As opposed to this design, the configuration of the system mixin list is fixed just before starting up the program.

3 A generic extensible preprocessor kit: EPP

To implement new languages or extend existing ones, pre-processors or translators are often used rather than native compilers (e.g. many tools for C/C++). The merits of this style of implementation are its simplicity and high portability; moreover, preprocessors offer a form of reflexivity since the introduced extensions are ultimately described in the extended language. We argue that the interest for such tools is symptomatic of their power of description of (admittedly often simple) abstractions of the code, and sometimes the possibility to deal, at least partly, with tangling issues. Syntactic issues are especially often adressed and preprocessors clearly help increase programs declarativeness.

3.1 Preprocessing tools

The Java language [GJG96] has recently become very popular among programmers but lacks facilities for language extension offered by many other object-oriented languages. For instance, C++ provides at least a macro pre-processor and has operator overloading and template facilities; Smalltalk and CLOS [Ste90] have closures and metaclass facilities. These features extending language constructs and operators supplement the class mechanism which extends data types.

Many extensions have been proposed for the Java language and often implemented as pre-

processors. Although there are potentially many useful language extension systems, extensions are often exclusive: it is generally impossible to merge several language extensions or eliminate harmful features from the extended system.

Systems with a meta-object protocol (MOP) such as CLOS [KdRB91], MPC++ [Ish94], OpenC++ [Chi95], EC++ [C$^+$97] or JTRANS [KK97] have solved this problem. These systems allow the implementation of language extensions as modules that can be selected by the users. Yet, extensibility of syntax is slightly restricted in these systems; composition of extensions is often not the main interest of these systems either.

3.2 EPP: the preprocessor kit

The Extensible PreProcessor kit, *EPP*, is a generic extensible preprocessor: it provides an application framework for writing preprocessor type language extension systems. EPP basic behavior is to transform a Java source code into the same program. By using the hooks provided by EPP recursive descent style parser the extension programmer can extend this basic behavior: he can introduce new features, possibly associated with new syntax without editing the existing source code of EPP. Because all grammar rules are defined in a modular way, it is also possible to remove some original grammar rules from standard Java.

Once the parsed program has been transformed into a tree, the pre-processor programmers can easily manipulate it from their program. The usefulness of such kind of tool has already been proven by Lisps and adapted to C++ by various systems like Sage++ [B$^+$94], MPC++ [Ish94] or OpenC++ [Chi95].

EPP enables preprocessor programmers to write extensions as separate modules that are finally composed to form a complete preprocessor. We call these extension modules *plugins*.

High composability of EPP plugins can be realized thanks to the module description language *Ld-2* and the *system mixin* feature.

The inheritance mechanism of object-oriented languages makes it easy to implement extensible applications because all methods of objects can be considered as hooks for extensions. In addition to the traditional inheritance mechanism, system mixins encourage the development of extensible and modular software.

The first prototype of EPP (lisp-epp) was written in Common Lisp [Ste90]. Using this system, several plugins have been implemented including a simple parallel language: Tiny Data-Parallel Java [IR97] and a reactive programming extension [RI97a].

```
#epp "enum"

public class EnumTest {
  enum {RED, GREEN, BLUE}
  enum {
    MALE,
    FEMALE,
  }
  public static void main(String argv[]){
    System.out.println(EnumTest.RED);
  }
}
```

Figure 2: A program using a enum plugin.

```
/* Generated by EPP 1.0beta3 (by lisp-epp1.0beta3) */

public class EnumTest {
  public static final int RED = 0;
  public static final int GREEN = 1;
  public static final int BLUE = 2;
  public static final int MALE = 0;
  public static final int FEMALE = 1;

  public static void main(String argv[]){
    System.out.println(EnumTest.RED);
  }
}
```

Figure 3: The translated program.

The source code of lisp-epp is now re-written in "Java extended by EPP" itself. The bootstapped byte-code is available on any platform where Java is supported. Plugins also work on any platform, therefore they can circulate and be used as casually as class libraries written in Java.

Although the current target of pre-processing is only Java, the architecture of EPP is applicable to pre-processors for other programming languages.

3.3 An Example of an EPP plugin

The user of the pre-processor can specify an *EPP plugin* at the top of his Java source files. The plugin will be dynamically loaded by EPP before starting the pre-processing. Fig. 2 is a simple example program using an EPP plugin that defines an enum macro.

Fig. 3 is the translated program. Each element of the enum declaration is expanded into static final field declarations.

The plugin is a Java class file written in "Java extended by EPP". Fig. 4 shows how the source

code of the enum plugin looks like. The source code makes use of four plugins (see Table 1 which explains the function of each plugin).

The source code consists of two parts: the definition of the extended syntax and the definition of the macro expansion function. The first part will add a "patch" to the recursive descent parser in order to add a new grammar rule of enum declaration. The second one defines a function which translates an enum declaration into a static final field declaration.

3.4 Parser design

Code generation is not the only part of EPP written in Ld-2. The parser itself is basically written in a recursive descent style [ASU87], using the system mixin feature. Each function which parses a non-terminal consists of nested `if-then-else` branched by the value of a look-aheaded token. The plugin programmer can add new `else-if` clauses to the methods in order to add new operators or new statements. As shown in figure 4, it is possible to manipulate the abstract syntax tree of the program without special problems. The final parser is composed of the original parser plus several patches in the order chosen for the composition of system mixins.

4 Beyond system mixins: an architecture for the weaver

From their primary objectives, most aspects seem to be domain dependent. Most probably, aspects can be grouped in families from which basic bricks could be extracted for describing their common points. Aspects are also fairly specialized, hence avoiding combinatory explosion should also be a concern of weaver programmers. Our position is that these basic bricks should not be classes but system-wide mixins (used in an AOP fashion or not). As a side effect, using mixins is a good way to preserve alternative designs for weavers, and potentially to fine-tune aspect implementations at run-time. For these reasons, we will essentially focus on the use of system-wide mixins for designing weavers.

```
#epp "Symbol"
#epp "SystemMixin"
#epp "AutoSplitFiles"
#epp "BackQuote"

package enum;
import epp.*;

SystemMixin enum {
  class Epp {

    extend Tree classBodyDeclarationTop() {
      if (lookahead() == :enum) {
        matchAny();
        TreeVec tvec = new TreeVec();
        match(:"{");
        while (true){
          if (lookahead() == :"}") break;
          tvec.add(identifier());
          if (lookahead() == :"}") break;
          match(:",");
        }
        matchAny();
        return new Tree(:enum, tvec);
      } else {
        return original();
      }
    }

    extend void initMacroTable() {
      original();
      defineMacro(:enum, new EnumMacro());
    }
  }
}

class EnumMacro extends Macro {
  public Tree call(Tree tree){
    Tree[] args = tree.args();
    int c = 0;
    TreeVec top =
      (TreeVec)Dynamic.get(:beginningOfClassBody);
    for (int i = 0; i < args.length; i++){
      top.add(
              `(decl
                  (modifiers
                    (id public) (id static) (id final))
                  (id int)
                  (vardecls
                   (varInit ,(args[i])
                    ,(new LiteralTree(:int,
                            Integer.toString(c++))))))
              );
    }
    return `(emptyClassBodyDeclaration);
  }
}
```

Figure 4: The source code of enum plugin.

We have experimented with Ld-2 system mixins first in the Lods compiler kit [IHT+96] and more recently for programming EPP extension plugins and have found them a very suitable and simple tool for improving program organization.

However, they provide only a limited form of compositionality. The standard composition rule of mixins is a simple, additive fusion of constructs in the order explicitly determined by the programmer (system mixin dependencies). Of course, each system mixin can itself define additional classes which can be extended by other system mixins: this slightly enhances the model compositionality and could permit to program aspects of system mixins (themselves aspects or parts of aspects). Improving composition capabilities should enhance the reusability of system mixins.

Moreover, defining a general architecture of aspects is very desirable: it would document the weaver and could keep knowledge about good or bad designs (for performance improvement or compatibility for instance: e.g. which aspects should not be used together ?).

4.1 Successive refinement approach

A first approach to introducing more compositionality in our design is to build a weaver as the successive application of smaller weavers, that is by applying a refinement process: the original program (possibly written using several languages) is transformed into simpler forms by the repeted application of simple transformations. In this approach, the final program is the result of a series of rewritings.

This approach can be reflective if the process applied at each step is the same: see for instance compile-time MOPs with an recursive generation of meta-objects.

The specification of the different refinements is quite simple in itself, especially in a meta-object architecture where each meta-object is a modular unit. However, global interactions of several mixins/aspect descriptions is not explicitable (except maybe by introducing group reflection, but which might then result in different implementations) and the composite architecture is not directly visible or even representable in a detailed way, apart from the succesive weavers applied.

4.2 Relational approach

This is why we are currently studying the introduction of a notion of relation connecting system mixins and explicitly conveying the composition description.

The purpose of this approach should be to expose all connections between components of a weaver, which we feel is especially important if we adopt a mixin-based solution.

Similar approaches have been undertaken like pattern languages [KC97] which can be used to define relations between different patterns; semantic nets are another example of such composition model.

In EPP, plugins must be assembled in a determined order which is specified in a setup list; this list can be seen as a set of precedence relations between the different plugins. Other relations are most probably interesting: policy for merging several plugins (addition, replacement, partial replacement, etc.), contracts to ensure the delivery of information between system mixins (in a dataflow-driven fashion), logical assertions, and so on. Relations are probably not only two-party but more probably multi-party: think for instance of how to combine several overlapping system mixins to form an aspect.

It is possible to use relations to represent possible or unacceptable design alternatives, especially in a mixin approach as we noted; the programmer could be able to select among a set of mixin strategies to implement his plugins (aspects descriptions).

Introducing such complex relations would probably also mean adding an interface to system mixins for connecting related types of relations. Since extended plugins can be very general, these interfaces should also be parametric on the relations in input: this would permit the most abstract definition to be possible. For instance, if an information is expected by a plugin, it could be parameterized to offer pattern-like generative properties. Component litterature should probably be reread with aspects in mind instead of direct composition of components.

The granularity of extended plugins certainly influences their reuse, but in this approach, their composability essentially depends on the kind of relations that they can support and establish with other extended plugins. That way, building an aspectual weaver would be nearly transformed into merely connecting them with relations.

A relational approach is probably more difficult to design than a refinement one (that we can program now with existing tools), but would surely bring important benefits for code organization.

As a side note, it might well be possible to use

this approach with the previous one. The former approach could be an especially easy way to implement aspects able to survive weaving and providing run-time adaptations as suggested in [MJV+97].

4.3 Example sketch

In parallel object-oriented languages, guards are good instances of aspects. They can either be implemented as an usual extension for active object languages (possibly using reflexive techniques); or guards in themselves can be a program describing the synchronization of all components of the system, separately from the program, in relation with objects decomposition (for a more concrete example, have a look at [LK97]).

Many kinds of guards can be described: the term essentially specifies the protection of a routine by some test, but it is possible to specify that restrictions be incrementally added (e.g. [Fro92]) or to define guards conditions exclusively expressed on synchronizations variables, etc.

Defining an aspect language for each of these different guards is possible but clearly restrictive. Moreover it is not informative as to what are the possibilities for implementing guards. On the contrary, it is rather natural to construct more basic modules for describing all possible guard languages, like for instance:

- *methods listing*, defining how to relate some piece of code with any method;
- *synchronization variables*, providing a declarative framework for defining synchronization variables;
- *condition restriction*, defining how to check that conditions are increasingly strengthened;
- etc.

These modules correspond to different choices of implementation for guards, but they cannot be combined in an arbitrary manner; they are also more general than a direct implementation of a specific type of guard since no commitment to a complete combination of their different features is done by default. A guard aspect would provide a language for associating the effective service of a method request with conditions: such an aspect language could, among other possible solutions, depend on modules like *method listing* and *synchronization variable*.

Fig. 5 represents a possible configuration where we would define the *synchronization variables based guard* with a parameter concerning the method it controls. A relation would be defined to express that the aspect language has to take the *method listing* plugin's output and use it to replace the parameter in the guard plugin: for every method listed, a new guard would be produced (in the absence of a specification, the default behavior being to preserve the method service untouched). The input and output arrows do not have the same meaning as the other ones, since they in fact represent files that are used as input/ouput by run-time components (run-time meaning weaver run-time, not final program run-time).

The usefulness of a relational model depends probably on what relations are expressible and easy to understand at the same time (and on what kind is not).

5 Visualization and documentation issues

An important purpose of modularizing the weaver description is to be able to represent and document aspects interactions. Using diagrams and providing a visual interface for assembling them into a weaver is important in this respect.

Many examples (e.g. JavaBeans for components assembly and event dispatch description, KLIEG [STST97] for describing behavioural patterns, ...) show the interest of visual environments for component assembly. In an aspect-oriented approach, visual tools could be developped for programming aspects (user-oriented) or for coordinating aspect descriptions in the weaver (for advanced users and for user documentation).

In relation with the study of relations, we intend to provide an environment for composing system mixins with a graphical interface close to Fig. 5. For instance, in EPP, instead of writing the plugin setup by hand, the user would be able to graphically build it as a graph of precendence relations between plugins.

Is a graphical description really enough if we want to deal with libraries of elementary components for aspect description (especially, what about the size of such libraries) ?

Textual documentation is a very important part of software. Can documentation or specification (think of assertions for instance) be considered as aspects ? For instance, literate programming [Knu92] relies on separate definition of documentation through a kind of anchoring technique which allows to write pseudo-code and to reference other parts of a "program". Its intent is however notably different from AOP, since it is presented as

Utrecht, the Netherlands, July 1992. European Conference on Object-Oriented Programming, Springer-Verlag.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse in object-oriented designs. In O. Nierstrasz, editor, *Proceedings of ECOOP'93*, Berlin, 1993. Springer-Verlag.

[GJG96] J. Gosling, B. Joy, and Steele. G. *The Java Language Specification*. Java Series. Sun Microsystems, 1996.

[IHT+96] Yuuji ICHISUGI, Satoshi HIRANO, Hitoshi TANUMA, Kuniyasu SUZAKI, and Michiharu TSUKAMOTO. Compiler Widgets — Reusable and Extensible Parts of Language System — (in Japanese). In *The 11th workshop of object oriented computing WOOC'96, Japan Society of Software Science and Technology*, March 1996.

[IR97] Yuuji Ichisugi and Yves Roudier. The extensible java preprocessor kit and a tiny data-parallel java. In Yutaka Ishikawa, Rodney R. Oldehoeft, John V. W. Reynders, and Marydell Tholburn, editors, *proceedings of ISCOPE'97 (International Scientific Computing in Object-Oriented Parallel Environments Conference), December 8 - 11, 1997, Marina del Rey, California, USA.*, number 1343 in LNCS, pages 153–160. Springer-Verlag, 1997.

[Ish94] Y. Ishikawa. Meta-level Architecture for Extendable C++, Draft Document. Technical Report Technical Report TR-94024, Real World Computing Partnership, 1994.

[KC97] Norman L. Kerth and Ward Cunningham. Using patterns to improve our architectural vision. *IEEE Software*, pages 53–59, January/February 1997.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Dan Bobrow. *The Art of the Meta Object Protocol*. MIT Press, 1991.

[KK97] A. Kumeta and M. Komuro. Meta-Programming Framework for Java. In *The 12th workshop of object oriented computing WOOC'97, Japan Society of Software Science and Technology*, March 1997.

[Knu92] Donald E. Knuth. Literate Programming. Technical report, Stanford University Center for the Study of Languages and Information, Leland Stanford Junior University, 1992.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A Language Framework for Distributed Programming. Technical Report SPL97-010 P9710047, Xerox PARC, February 97.

[MJV+97] Frank Matthijs, Wouter Joosen, Bart Vanhaute, Bert Robben, and Pierre Verbaeten. Aspects should not die. In *ECOOP'97 workshop on aspect-oriented programming*, 1997.

[RI97a] Yves Roudier and Yuuji Ichisugi. Integrating data-parallel and reactive constructs into java. In *proceedings of the 2nd France-Japan Workshop on Object-Based Parallel and Distributed Computing (OBPDC'97), Toulouse, France*, October 1997.

[RI97b] Yves Roudier and Yuuji Ichisugi. Java data-parallel programming using an extensible preprocessor. In *proceedings of SWOPP'97 (Summer united WOrkshops on Parallel, distributed and cooperative Processing), Kumamoto, Japan*, pages 85–90, June 1997.

[Ste90] G.L. Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 1990.

[STST97] Etsuya Shibayama, Masashi Toyoda, Buntarou Shizuki, and Shin Takahashi. Visual Abstractions for Object-Based Parallel Computing. In *proceedings of the 2nd France-Japan Workshop on Object-Based Parallel and Distributed Computing (OBPDC'97), Toulouse, France*, October 1997.