Reengineering Cocoon with AspectJ

Leeann Bent

Abstract

AspectJ is a new Aspect Oriented extension to Java. This study attempts to quantify how appropriate AspectJ is for a large body of open source code (the Cocoon project) by reengineering it. In the process we discovered a number of things. The first is a set of patterns or templates for creating new or converting to AspectJ code. These are outlined and evaluated in the paper. The second is that AspectJ fundamentally changes and simplifies the "Knows About" relationship in a body of code. This simplification has the benefit of fixing the "Reverse Inheritance" problem found in real code. With AspectJ, "Knows About" relationships are localized, and often reduced. This localization removes the need for classes to "Know About" and import secondary behavior. It was also found that AspectJ is a good tool for the expression of Layered programming [5]. This is shown in the implementation of the Layered Aspect type. Finally, it appears that the Cocoon code base was designed with "Aspects" in the abstract, even though they were implemented using Java functionality. This is made especially apparent by the prevalence of reverse inheritance in the code.

1 Introduction

AspectJ [1] has proven to be a valuable programming tool for developers. It has been used in the Apache system [7], as well as in other work [2, 3]. However, questions still remain about the use of AspectJ in existing systems, including how and when to use AspectJ. To answer these questions an existing project was re-engineered. The qualitative and quantitative impact of AspectJ on modularity was measured, and the original code was examined for existing crosscutting concerns. These were used to characterize when and how AspectJ might be helpful. A secondary goal of this project was to utilize the tools available (compiler, development environment, visualization tools, etc.), both exercising them, and attempting to assess their contribution to design and implementation. The code base that was chosen for reengineering is Cocoon [6], an open source web-publishing framework. Cocoon is a moderate sized project in later stages of development, well suited to reengineering.

Over the course of this project a total of nine aspects were created. These aspects were often easy to find, and reduced tangling significantly. We discovered that these nine aspects could be divided into three different structural types: the Reverse Inheritance (RI) type, the Layer type, and the Filter type (these names refer to the AspectJ solution). All of these structural types convert easily to AspectJ. The most common of these Aspects was the Reverse Inheritance type. The prevalence of this type made it easy to convert much of Cocoon's functionality to AspectJ code and revealed that the designers of Cocoon designed Cocoon with aspects in mind.

Focus and Scope of the Paper

This paper will focus on the three different Aspect types. At least one example of each will be given, as well as an explanation of the original design problem that motivated the change. We will define the design problem, discuss the AspectJ design/implementation, and evaluate the implementation. We will attempt to quantify the changes by measuring several things. These measures are designed to assess tangling and code size. The first is size or the number of classes in the original object. This is simply a measure of coding effort. The second is modularization (or lack thereof), or the number of parts involved in implementing the original design *not* within their module. This is a traditional measure of *code tangling* -- how a module's code is spread across non-implementing classes. The third is localization, or the number of objects that reference the original object. This is another measure of tangling, but it refers to tangling in the "Knows About" relationship (or *"Knows About" tangling*). All of these measures are compared with the same measures for the AspectJ version of the code. Additional measures that cannot be easily quantified, but are nonetheless important, are reuse of original code, and the reusability of the AspectJ code. These are discussed in a qualitative fashion.

The rest of the paper is organized as follows. Section three introduces Cocoon and motivates the decisions for aspectizing the code. Sections four through six discuss the different aspect types. Section seven summarizes and concludes.

2 Cocoon

Cocoon [6] is an open source web-publishing framework from the Apache group. Cocoon is a moderately sized project (197 Classes, 19KLOC) that is well supported. The current release is version 1.7.4, and development is in process for Cocoon 2. Because of the extensive restructuring in version 2, we focused our efforts on Cocoon 2, pre-beta. Cocoon's basic functionality is to turn XML content into web palatable formats, including

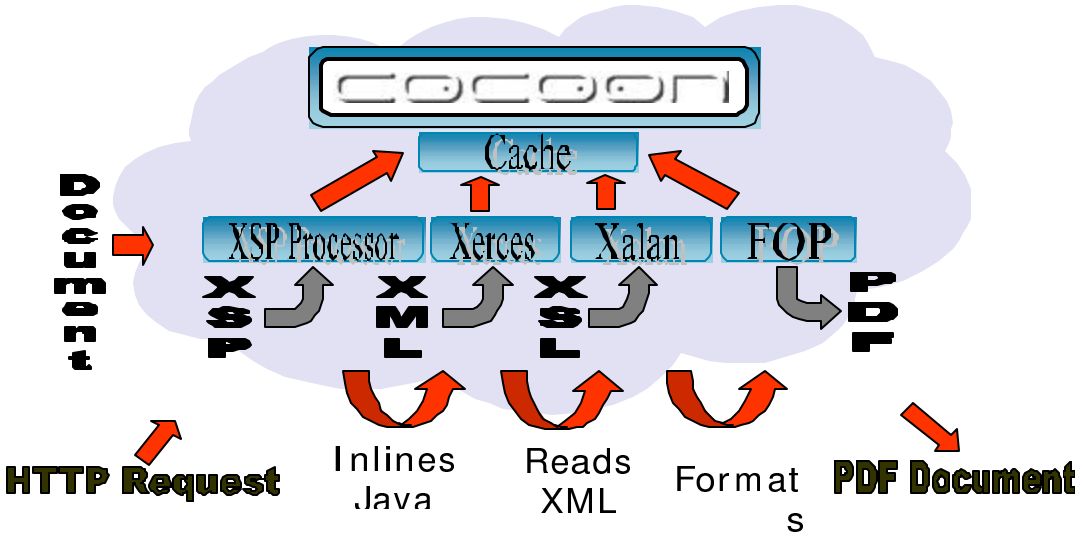

Figure 1: Cocoon serving a PDF document with Java code inserted using XSP.

HTML, XHTML, and PDF. Cocoon does this using other Apache modules. A sample use would transform some XML content into HMTL format. To read in this XML content, Cocoon would use the Xerces XML processor. To format this document in HMTL, Cocoon would use the Xalan stylesheet formatter, along with a stylesheet written in XSL. Cocoon has more complicated functionality, as well. It uses a variant of XML called XSP to allow Java code to be inserted into the original XML document. This Java code is compiled and executed, with the code's output inserted back into the XML before the document is served. The XSP Processor module in Cocoon takes care of this. Cocoon is also able to format output for PDF, using the FOP module. See **Figure 1** for a picture of Cocoon 2 serving a request for a PDF document with dynamic Java content.

Cocoon is partitioned into two different packages: the Avalon package and the Cocoon package. The Avalon framework does component management within both packages, while the Cocoon package implements Web functionality. Consequently, objects often inherit from many interfaces and objects to get aspects of behavior from Avalon and Cocoon. This leads to two different types of tangling: *code tangling* and *"Knows About" tangling*. An example of this is shown in **Figure 2**.

```
public class ProgramGeneratorImpl
  implements ProgramGenerator, Composer, Configurable {
            ...
  protected ComponentManager manager;
   public void setComponentManager(ComponentManager manager) {

       this.manager = manager;
       this.factory = (NamedComponentManager);
       this.manager.getComponent("factory");
   }
      ...
}
```

Figure 2: ProgramGeneratorImpl inheriting from ProgramGenerator, Composer, and Configurable.

As shown, ProgramGeneratorImpl inherits behavior from ProgramGenerator (Cocoon), Composer (Avalon), and Configurable (Avalon). This is an example of the "Reverse Inheritance" problem. Every subclass of ProgramGeneratorImpl must "Know About" Composer and Configurable, even though the functionality of these modules is different from the functionality of ProgramGeneratorImpl and ProgramGenerator. (Additionally, these are services provided by ProgramGeneratorImpl to the Avalon package) We would prefer that there be a "Reverse Inheritance" instead, that the Composer object could impose its behavior on ProgramGeneratorImpl, without ProgramGeneratorImpl knowing about Composer. The code in **Figure 2** exhibits the second type of tangling*: "Knows About" tangling*. In addition, this example code exhibits the first type of tangling as well: *code tangling*. ProgramGeneratorImpl must implement functionality for Composer and Configurable. The actual implementation (or code) for Composer and Configurable is

spread into ProgramGeneratorImpl. This is shown in **Figure 3.** The Composer code is shown in blue, while the Configurable code is shown in green. This is an example where the designers of Cocoon have used the Avalon framework (of which Composer and Configurable are both a part) to capture "Aspects" of behavior. These "Aspects" implement unrelated behavior, but are tied to class implementation because of Java's inheritance hierarchy.

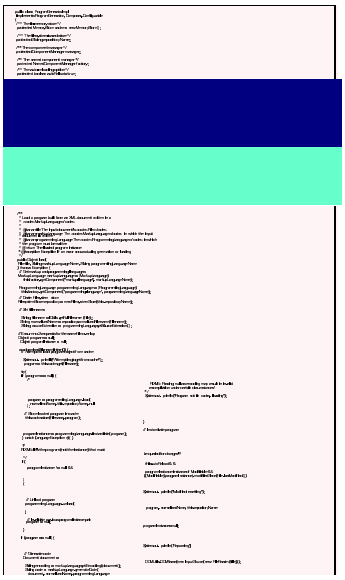

Figure 3: ProgramGeneratorImpl. The blue is Composer code; the green is Configurable code.

3 Reverse Inheritance Aspects

The Reverse Inheritance type (**Figure 4**) is the most common found in Cocoon 2. An example Aspect of the Reverse Inheritance type is the AspectJ version of the Composer interface from **Figure 2.** The Composer interface is responsible for tracking an object's manager. Because of the semantics of interfaces, Composer must be implemented in the subtype. This results in generic code that is tangled across many classes and has a couple

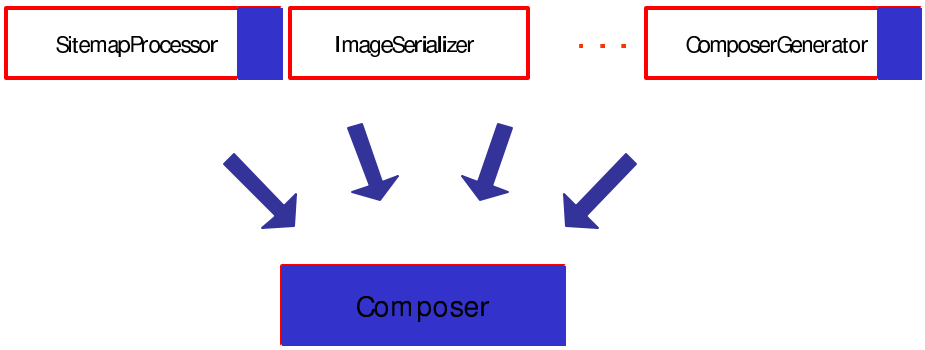

Figure 4: The Reverse Inheritance Type problem. Every object (33 of them) that needs Composer functionality must know about Composer. In addition, some objects may need to implement Composer behavior.

of negative effects. The tangling foils independent maintenance of the two functionalities. In addition, this inheritance means that subtypes must *know about* Composer, even if their use of Composer is minimal. Thus, Composer shows two types of tangling: tangling of *code* implementation and tangling of *"Knows About"* relationships. While the Composer object is inherited into many objects, the behavior of Composer is almost always independent of the behavior of the subtype. Composer's functionality is weakly related to the subtype. This structure, a secondary behavior that must be implemented in a tangled fashion, is ideal for conversion to Aspects.

It is the "Know About" tangling that characterizes the Reverse Inheritance (RI) Aspect type. Rather than requiring an object to know about and (possibly) implement desired (yet unrelated) behavior, it would be preferable to have the behavior pushed out into the object. This is reverse inheritance, and can be implemented using AspectJ. We can remedy both the case where *code* and *"Knows About"* tangling exist.

Composer: Code Tangling and "Knows About" Tangling

Description of Composer

The Composer Aspect is an example of an Aspect that fixes both code tangling and "Knows About" tangling. In the original Cocoon code, the Composer interface is used by classes that provide access to their manager (frequently necessary for reusing instantiated components). Since it is the client of an object that requires access to the manager, the client or the manager is the best-suited to impose the management. In Java, the object squeezes it in because the language makes it hard for the client or manager to impose management behavior. This manager (stored through the Composer interface) is subsequently used to set up object fields and reference other objects. Thus, the Composer interface in Avalon simply declares the abstract method **setManager**; the implementing classes declare the **manager** variable and define this method. In most cases, **setManager** simply sets the manager. This is shown in **Figure 5**. In other cases this method was much more complex, including code that set up object fields dependent on the **manager** variable. An example of this type of code is shown in **Figure 6.** Both of these behaviors must be captured in the Aspect design.

```
public class XalanTransformer extends DocumentHandlerWrapper
  implements Transformer, Composer, {
    ...
    protected ComponentManager manager;

    public void setComponentManager(ComponentManager manager) {
        this.manager = manager;
    }
  ...
}
```

Figure 5: A generic implementation of Composer with variable manager and method setComponentManager.

Aspect Design

The Aspect design was intended to capture the basic functionality of a Composer. The **manager** variable itself is moved into the Aspect (Composer), and this variable is set when the object is created using the Aspect's **setManager** method. Creating a pointcut for **setManager** presented some difficulties because the **manager** variable is set during object initialization using an argument from the constructor. Since AspectJ does not allow pointcuts before constructors, a pointcut on the creator of the object was used. A second aspect was used to encapsulate the pointcut that sets the manager. This design reflects the fact that the manager is an aspect of each Composer inheritor object, while setting the manager is an aspect of those objects that initialize Composer objects. Additionally, it is the pointcut aspect that *imposes* the manager behavior on objects.

```
public void setComponentManager(ComponentManager manager) {
   this.manager = manager;
   Enumeration t = this.types.elements();
   while (t.hasMoreElements()) {
      Hashtable components = (Hashtable) t.nextElement();
      Enumeration c = components.elements();
      while (c.hasMoreElements()) {
        NamedComponent component = (NamedComponent)
                                          c.nextElement();
        if (component instanceof Composer) {
          ((Composer) component).
                        setComponentManager(this.manager);
        }
      }
   }
}
```

Figure 6: An implementation of Composer where setComponentManager does more than set the manager variable.

Implementation

While the original design confined **manager** to the aspect, in the implementation introduction was used to move it back into the classes; **setManager** was moved back into the classes using introduction as well. This allows us to reintroduce the Composer interface to allow for type checking. **Figure 7** shows this.

When an object that requires Composer functionality is created (using **new**, **newInstance**, or **load**) the pointcut in the caller checks for the Composer type. If the object being created is a Composer type, **setManager** is called. This pointcut is shown in **Figure 8**.

This aspect now looks very similar to the original code. The Composer interface contains the declaration of **setManager**. The introduction aspect (AComposer) then defines

```
aspect AComposer of eachobject(instanceof( SitemapProcessor ||
                                Block || ImageSerializer ||
                                Sitemap ||... )) {

  introduction (SitemapProcessor ||
                Block || ImageSerializer || Sitemap || ...) {
       implements ComposerO;

    private ComponentManager my_manager;

    public ComponentManager getManager() { return my_manager; }
    public void setManager(ComponentManager manager)
       { my_manager = manager;}
  }
```

Figure 7: AComposer introduces the Composer interface into classes directly.

setManager, and declares the manager. Finally, AComposerSetter contains the pointcut that imposes manager behavior.

Code from the original **setManager** that was not directly involved in setting the manager (such as that in **Figure 6**) was moved into initialization methods. These are also introduced into classes using **introduction**. These are called (depending on type) after **setManager**.

```
aspect AComposerSetter of eachobject (instanceof(Sitemap ||
                                                 ...)) {

  pointcut setupComponentManager(Object cs) returns Object:
    instanceof(cs) && calls(*, new(..)) || ...;

  around (Object cs) returns Object:
                              setupComponentManager(cs) {
    Object o = thisJoinPoint.runNext(cs);

    if (o instanceof Composer) {
      this.setupManager(cs, o);
    }
    return o;
  }
```

Figure 8: Pointcut on calls of new(), newInstance() and load() used to set the manager for Composer objects. setupManager() calls setManager() and init().

Evaluation

This implementation changed one interface into one Aspect composed of two AspectJ aspects and one interface. There were 9 uses of **getManager**() outside the Aspect, where referencing the manager was necessary. This aspect eliminated 33 classes that inherited or implemented Composer interface. In addition to a decrease in size, this reduction indicates that this Aspect fixes the code tangling problems discussed in the introductory section. There is no longer any code tangling, because all of the Composer code is

implemented in one place. The AspectJ code is also fairly localized, with only 9 references to the manager object outside of the aspect itself. In addition, redundant code has been eliminated though the use of one **setManager** function, used for all Composers.

The "Knows About" tangling has been removed as well. In the AspectJ version, only **AComposerSetter** must know which classes require Composer behavior. This localizes the "Knows About" relationship, and introduces a "Reverse Inheritance" where the Composer functionality is imposed on a class. **Figure 9** shows both of these improvements (over **Figure 4**).

This implementation turned out to be fairly similar to the original Avalon design, except that the "Knows About" relationship has been simplified, allowing the code to be localized and condensed. Because the implemented behavior was identical to the original code, refactoring from Java to AspectJ was very easy. While almost every method was changed, the majority of lines (33 of 36) in the (often redundant) original code's method bodies went unchanged. Because of the similarity to the original code, refactoring from Java to AspectJ was very easy. Additional refactoring will also be easy due to the new

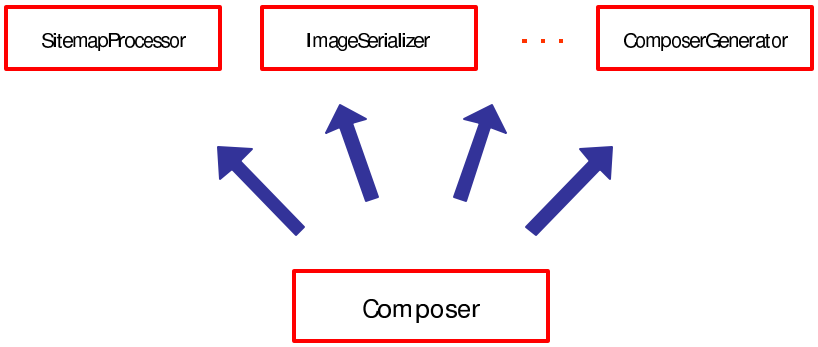

Figure 12: Now only Composer "Knows About" the 33 objects. The uses of Composer are localized, and behavior is separately encapsulated.

design's encapsulation. Now that the Composer functionality has been distilled out, it can be introduced into any class. Additionally, any class may be reused with or without Composer functionality.

There are some unresolved details regarding this aspect. Moving the initialization code into a separate method (**init**()) does not appear to be an optimal solution. While it is true that the original design mixed the functionality of setting the manager and initializing the object variables, there are other alternatives to a simple initialization method. The first is to create another aspect with initialization and hang it on the **setManager** method. This aspect would take care of the additional setup. The second is to create a **setManager** method for each different object type. This would require typing the pointcut (to avoid casts) which in turn would require creating an aspect for all types that have managers (plus exception types). I have not implemented either of these due to time constraints. A second potential problem is the use of the Composer type. While using the Composer

interface defines the behavior of management, the use of the Composer interface could have been avoided by using **hasAspect** to check if objects have the manager aspect (**o.hasAspect(AComposer)**). This design would have simplified the Aspect, and decreased the number of classes involved.

NamedComponent: "Knows About" Tangling

The NamedComponent is an example of an Aspect where only *"Knows About" tangling* is present in the original code. Aspectizing NamedComponent does not decrease the number of unrelated classes that implement functionality, and gives no noticeable decrease in program size or references. Instead, the "Knows About" relationship is localized and simplified, again providing Reverse Inheritance that imposes NamedComponent functionality.

The NamedComponent is an interface in Avalon that defines an access method (**getName**) for the name of an object. NamedComponent is introduced into components in order to give them "names"; it introduces a second functionality orthogonal to an object's primary behavior. The inheritance is trickier than in Composer however, because the introduced name is object dependent (one name per object), and should not collide with any other object name. This means that each implementation of NamedComponent must be specialized according to its class.

Like in Composer, the object requiring a name must "Know About" NamedComponent to have that behavior. The "Knows About" relationship is further tangled by the requirement that names be unique. Each object that implements NamedComponent must "Know About" every other object that implements NamedComponent. This relationship is shown in **Figure 13.**

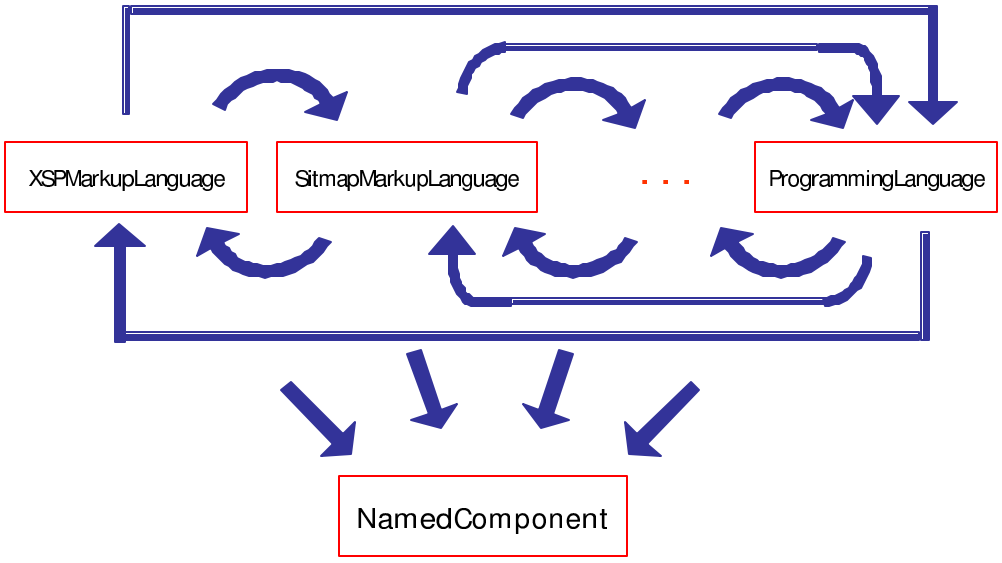

Figure 13: The original "Knows About" relationship for NamedComponent. All components that use NamedComponent must know about NamedComponent and each other.

Aspect Design

NamedComponent's Aspect design is also very simple. The inheritance from NamedComponent is removed explicitly, and reintroduced using introduction. A String variable for name and its access method for each object are placed into one large aspect.

Implementation

This aspect was implemented using introduction to preserve the NamedComponent object. NamedComponent was preserved, with the abstract definition of **getName**(). This was done so that the calls to **getName**() do not need to be typed. The NamedComponent

```
aspect Introductions of eachJVM() {

  introduction XSPMarkupLanguage {
    private String name = "xsp";
    public String getName() { return this.name; }
  }
  introduction SitemapMarkupLanguage {
    private String name = "map";
    public String getName() { return this.name; }
  }
  introduction JavascriptLanguage {
    private String name = "javascript";
    public String getName() { return this.name; }
  }
  introduction JavaLanguage {
    private String name = "java";
    public String getName() { return this.name; }
  }
  introduction (MarkupLanguage ||
            ProgrammingLanguage) {
    extends NamedComponent;
  }
 ...
}
```

Figure 14: Introductions for NamedComponent aspect, including specific string names. Because we're only using this aspect for introductions, we only require one. Thus eachJVM() is used.

interface was introduced to each class that originally inherited from NamedComponent. Then introductions were used to define the value the name String would have, and the **getName**() method for the different types. These were all grouped into one aspect, as shown in **Figure 14**.

Evaluation

Evaluations according to the first two metrics (code size, and external implementations) defined in the introduction do not provide a clear view of the advantages of this Aspect. This is because they measure *code tangling*. Still, we state them for completeness. The NamedComponent aspect changed one interface into two AspectJ aspects (the introductions shown in **Figure 14** and the NamedComponent interface that we have retained). We have succeeded in localizing the NamedComponent: the nine different places where NamedComponent was inherited or implemented have now been consolidated into one area. However, this aspect still has five different implementations for each class that needs a different string. There are also five different classes that inherit the NamedComponent interface (either directly, or through other classes). Additionally, there are still 22 classes with 50 references to **getName**() interspersed the code, even though **getName**() itself is "hidden" in an aspect. These calls cannot be removed because they are used in very diverse locations: from accessing components to printing error messages.

In spite of this, the detangling has succeeded. Now, the only component that needs to know how NamedComponent works is the Aspect. This is a change in the *"Knows About"* relationship shown in **Figure 15**. The "Knows About" relationship has been localized, and NamedComponent now imposes its behavior onto other objects. Again, we have introduced a kind of Reverse Inheritance. Not only does this eliminate the need for other components to "Know About" NamedComponent, but they also do not need to "Know About" each other. This detangling also helps reuse. Classes can be used with or without NamedComponent functionality (for example reusing SitemapMarkupLanguage without the name). None of the behaviors defined in the introductions can be reused, however, because they are class specific. This Aspect is a great example of how Aspects can simplify the *"Knows About"* relationship tremendously (by order n^2), through the use of Reverse Inheritance.

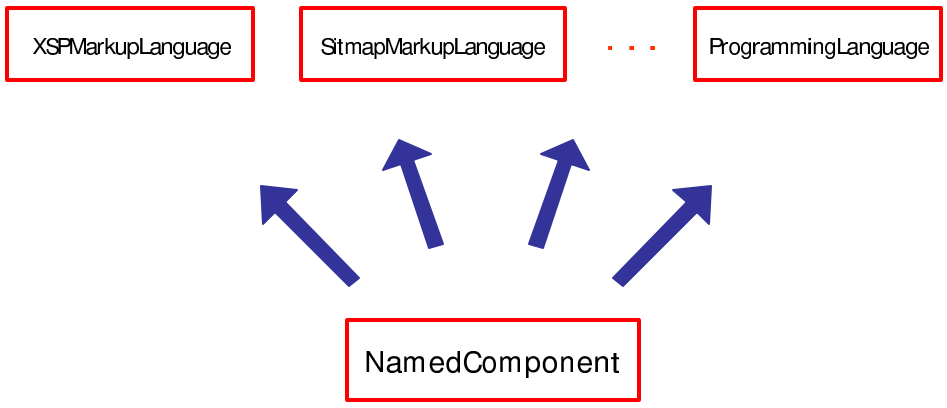

Figure 15: The "Knows About" relationship after AspectJ modification. Now, only NamedComponent must know about other components.

4 Layered Aspects

The PGICache Aspect is an example of a Layer aspect. The PGICache was created by layering the behavior previously found in ProgramGeneratorImpl (PGI). The PGI object is responsible for loading code from disk (using **load**()). However, the PGI also has a local cache for code that has been generated recently, and a check for out-of date programs (which must be regenerated). The cache is checked before the program is loaded, and an out-of-date check is performed after the program is loaded. If the code is out of date, it is regenerated. Thus, PGI has several functionalities, some of which are special cases of others. The PGI method **load**() is primarily responsible for loading code from disk. Layered on top of this functionality are the check for the cache and the check for out of date code. Because these functionalities are all implemented in one function, they must all *"Know About"* each other. This relationship is shown in **Figure 16.**

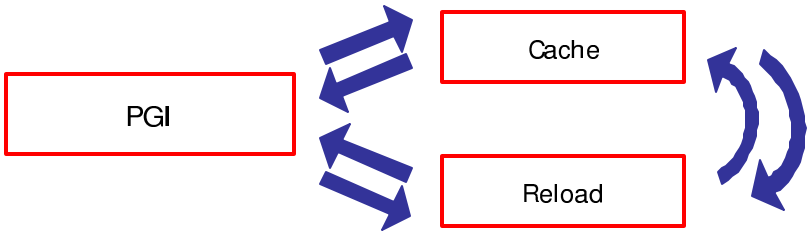

Figure 16: Because the cache, load from disk, and reload are implemented in the same method, every functionality must know about every other.

This Aspect was less prevalent in Cocoon 2. The PGICache is the only implemented Aspect of this type, though it appears that there may be others (including the high level architecture of Cocoon itself). This Aspect type is defined by its layered functionality. In PGI's case, we found like PGI to implement a primary behavior (loading from disk), and allow special case behavior to impose itself when necessary, without requiring PGI's knowledge. The amalgamation of these special cases into one method is also a form of code tangling. This tangling has effects similar to the Reverse Inheritance, pre-Aspect code above. PGI is now responsible for implementing all of the cache and recompilation logic, as well as its own loading logic. It is required to *"Know About"* these functionalities and implement them. Additionally, no part of PGI can be removed or reused without the other parts. This means that the cache cannot removed from PGI without modification of PGI, and program invalidation and recompilation cannot be reused in another part of the code. Again, using AspectJ we can improve upon this, creating a layered system that is not entangled and allows component reuse.

Description of ProgramGeneratorImpl

ProgramGeneratorImpl originally contained all of the logic for caching, loading, and regenerating code in the body of the **load**(..) method. The cache check was performed before loading, and the out-of-date check was performed after loading. Since multiple

Cocoon threads could be running, all of this code is protected by a synchronize block -- meaning only one thread can access code storage at any instant. This implied that all advice needed to be placed within this synchronize block, which presented additional constraints in the Aspect design. **Figure 17** contains a condensed version of the original load method.

```
public Object load(File, String, String) throws Exception {
         Object program = null; Object programInstance = null;

    synchronized(filename.intern()) {

       // Cache check
      program = this.cache.get(filename);

       // Load check
      if (program == null) {
          program = progLang.load(normalizedName,repositoryName);
          this.cache.store(filename, program);
      }

       // Modified check
      if ((Modifiable)
            programInstance).modifiedSince(file.lastModified()) {
        progLang.unload(program, normalizedName, repositoryName);
        program = null;
      }

       // Regenerate
      if (program == null) {
        Document document = DOMUtils.DOMParse(new InputSource(
                                                    new FR(file)));
        . . .
        repository.store(sourceFilename, code);
        program =
         programmingLanguage.load(normalizedName,repositoryName);
         this.cache.store(filename, program);
     }
     programInstance = programmingLanguage.instantiate(program);
   }
   return programInstance;
}
```

Figure 17: Condensed version of original ProgramGeneratorImpl code.

Aspect Design

The Aspect design attempted to strip PGI down to its most basic functionality: loading. Then, two pointcuts are added: one for checking whether the cache has the requested code and one for checking whether a program instance is out of date. Both have advice that imposes the appropriate action should either check be true. Note that the cache storage is moved into the aspect. The actual implementation is discussed below.

Implementation

```
public Object load(File file, String markupLanguageName, String
     programmingLanguageName) throws Exception {

    Object program = null;
     Object programInstance = null;

    synchronized(filename.intern()) {
        program = progLang.load(normalizedName,
                        repositoryName);
        programInstance =
          progLang.instantiate(program);
    }
    return programInstance;
}
```

Figure 18: Revised load method. Variable initialization condensed.

The first, and most difficult problem, encountered in this aspect is the fact that programs are cached, loaded, and regenerated via different references (i.e. file ids, explicit filenames, relative filenames). Since this Aspect accesses both the cache and the disk (to regenerate code), all of these references must be stored in the Aspect with a minimum of overhead. A PGICache object was created to store web code on a per PGI basis, and a cflow reception (See **Figure 19**) was used to capture the state information on a per PGI **load** request basis. This state information included all of the necessary references. PGI **load** was reduced to its simplest functionality, loading (See **Figure 18**). The CFlow captures the call of load and all control flow from the entry to the exit of **load(...)**. Within the cflow aspect, state (e.g. a stored filename) is available and valid throughout load's execution on a per load-call basis. Thus, there is no concern about potential cross-contamination of the references. By intersecting this cflow with pointcuts, we can use the

```
static public aspect loadInterface of
      eachcflow(loadInterface.loader(ProgramGeneratorImpl, File,
                                     String, String)) {
    MarkupLanguage markupLanguage;
    ProgrammingLanguage programmingLanguage;
    FilesystemStore repository;
    String filename;
    String normalizedName;
    String sourceExtension;
    Object program;
    File file;

    pointcut loader(ProgramGeneratorImpl pgi, File file,
                    String markupLanguageName, String
                    programmingLanguageName) returns Object :
      instanceof(pgi) &&
           receptions( Object load(file, markupLanguageName,
                                   programmingLanguageName));
    }
```

Figure 19:CFlow aspect (loadInterface) and pointcut (loader) used to store load request information in cflow local variables (i.e. markupLanguage). The static keyword is used because this aspect is inside the PGICache aspect.

stored state information for a particular call for different behaviors.

```
pointcut plLoadCall() returns Object :
      instanceof(ProgramGeneratorImpl) &&
      calls(ProgrammingLanguage, Object load(String,
                                        String, String));

around() returns Object : plLoadCall() {
      program = cache.get(filename);
      if (program == null) {
            program = thisJoinPoint.runNext();
            cache.store(filename, program);
      }
      return program;
    }
}
```

Figure 20: Code to check and update the cache, wrapped around programmingLanguage.load. Note that program and filename are cflow local variables.

```
 pointcut aroundInstantiate(ProgramGeneratorImpl pgi)
       returns Object : instanceof (pgi) &&
            calls(ProgrammingLanguage, Object instantiate(Object));

 around (ProgramGeneratorImpl pgi) returns Object :
    aroundInstantiate(pgi) {

      Object programInstance = thisJoinPoint.runNext(pgi);

     if ((Modifiable)
          programInstance).modifiedSince(file.lastModified())) {
        progLang.unload(program, normalizedName, repositoryName);
        program = null;
       }

      if (program == null) {
        Document document = DOMUtils.DOMParse(new
                                InputSource(new FR(file)));
        . . .
        repository.store(sourceFilename, code);
        program=
           programmingLanguage.load(normalizedName,repositoryName);
        this.cache.store(filename, program);
      }
      programInstance =
                  programmingLanguage.instantiate(program);
    }
    return programInstance;
 }
```

Figure 21: Code to check for programInstance validity is wrapped around programmingLanguage.instantiate. file, normalizedName, repositoryName, sourceFilename, filename, and program are local to the cflow aspect.

The cache is checked and updated by using advice on the call to programmingLanguage **load.** This advice is intersected with the cflow, allowing it to use the information stored in the cflow to access (look up and update) the program in the cache. See **Figure 20** for this pointcut.

The validity of each programInstance is checked on return from instantiation. Around advice is again intersected with the cflow to allow the return value of programmingLanguage.instantiate to be checked. If this programInstance is out of date, it is reparsed and recompiled using values stored in the cflow. See **Figure 21** for an example of this.

Evaluation

This implementation changed one class (in the original) into one class and one Aspect in the AspectJ version. Because the ProgramGeneratorImpl is already localized, there are no localization effects. The same number of objects reference PGICache and PGI as did before the aspectization.

While it appears that Aspectization has changed little about this design, in actuality both *code tangling* and *"Knows About"* tangling has been removed. This is because in the original design one object implemented several functions. The different function's code was tangled into one object. Though the cache and re-parse from disk are currently implemented together (to save effort), it would be a simple matter to separate these into two aspects. Once they are separated, the three different components are no longer tangled. Any component of this functionality could be reused, or left unused. This aspect also changes the "Knows About" relationship (See **Figure 22**), decreasing the number of "Knows About" relations, and limiting them to the aspects. This means that once again the Aspect imposes the behavior. The PGI component is no longer required to "Know About" caching and re-parsing from disk, and the Aspects are unaware of each other.

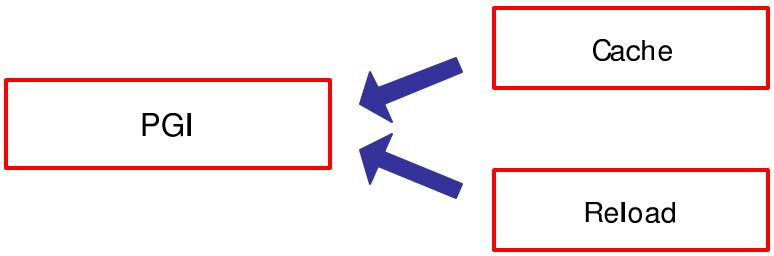

Figure 22: After reengineering, only Cache and Reload must know where they are placed with respect to PGI.

Additionally, this aspect showcases a technique that can be used to store state information on a per function call basis: the use of cflow. CFlow provides a mechanism that captures information on a functional basis, rather than an object basis, allowing this information to

be accessible deeper in the call tree. The cflow allows access to PGI load's local variables; cflow local information stores method scoped information. This requires some care, since variables in the cflow must mirror those in **load** to remain valid over the course of the method. In this case, cflow allows access to variables that would otherwise be hidden to the Aspect.

A final note about this aspect is that the "default" behavior of the original PGI was difficult to ascertain. I chose loading from disk, but the default behavior could easily be considered checking the cache, or re-compiling from code. It would be fairly straight-forward to code either of these in a similar fashion using AspectJ.

5 Filter Aspect

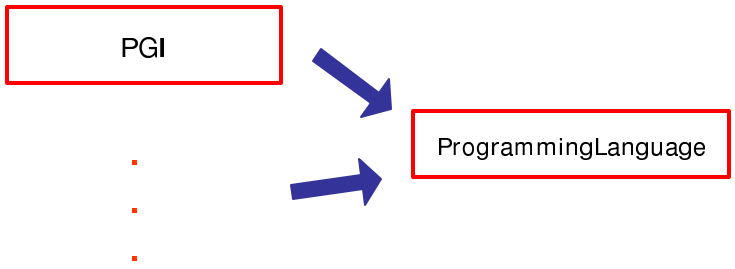

Figure 23: PGI, and potentially other callers, must know about the format for arguments to Programming Language.

A filter aspect is used to replace translation routines, where a method is expecting to receive information in a specific format, but it is available to the caller in another format. An example is shown in **Figure 24**. One String type, filename, is translated into another String type of a different format, normalizedFilename. The caller only has available filename therefore it must translate filename into a normalized filename. The difficulty with this type of implementation is shown in **Figure 23**. Every module that wishes to call **load** must know the format required by **ProgrammingLanguage**. This conversion can be made transparent however, using Aspects. While the only example of this Aspect type in Cocoon is Filename translation, it has been seen in other work [4], and thus deserves mention.

```
program = programmingLanguage.load(
                  repository.normalizedFilename(filename),
                  this.repositoryName, encoding);
```

Figure 24: An example of where a filter aspect is appropriate. The caller has filename, however programmingLanguage.load expects a normalized filename.

Filter Aspects do not suffer from the *code tangling* problems discussed with some of the other Aspects. It is easy to reuse the transformational method and, assuming the

format is correct, programmingLanguage.load can be reused. The problem with Filter Aspects is that the caller is required to know that programmingLanguage.load requires a normalized filename. The caller is also required to know about the conversion routine. This is a burden on the caller, and constitutes *"Knows About"* tangling. It would be better to make this conversion automatic and transparent to the caller. Then the conversion could happen when and where appropriate as dictated by the Aspect.

Aspect Design

The design of this aspect was rather simple: simply remove the translation call, capture the original filename with a pointcut, transform it to normalizedFilename, and pass this on to the method that required it, using runNext. Originally, I planned to perform this type of filtering on other calls as well. This turned out to be infeasible, because type conversions were required for some of these calls (e.g. File to String).

Implementation

```
public aspect PGIFileUtils dominates PGICache
      of eachobject(instanceof(ProgramGeneratorImpl)) {

  FilesystemStore repository;

  pointcut catchRepository() :
                instanceof(ProgramGeneratorImpl) &&
                calls(FilesystemStore, new(..));

  after () returning (FilesystemStore fss) :
                catchRepository() {
      repository = fss;
  }

  pointcut normalizeFilename(String name) returns Object :
      instanceof (ProgramGeneratorImpl) &&
      calls(ProgrammingLanguage, Object load(name, String,
                                             String));

  around (String name) returns Object :
                       normalizeFilename(name) {
      String normalizedName =
                repository.normalizedFilename(name);
      Object program =
                thisJoinPoint.runNext(normalizedName);
            return program;
  }
}
```

Figure25: Around advice on load captures the filename and transforms it into a normalized filename. Note that a copy of the repository must also be captured for filename lookup.

The implementation of this aspect was also rather simple. The translation call was removed, and a pointcut was inserted to capture the call to load, and perform the translation.

One complication with this code is that the repository is needed to transform the filename. A variable to reference the repository is added to the aspect, and a pointcut is added to capture and set it. This is also shown in **Figure 25.** The filter Aspect takes care of the translation, including the use of repository for lookup. For each class that needed filename conversion one aspect was created. This could have easily been implemented as only one aspect, however.

Evaluation

Because I chose to have one filename conversion per conversion type, this implementation required two aspects on two different calls of load. This actually increased the size of the code base, since no classes were removed for this aspect (the filtering was previously handled by a method in repository). However, the effects of filename translation are now completely transparent. The burden of the *"Knows About"* relationship has been removed from the caller of **load** and potentially other methods that require filename conversion. The "Knows About" relationship has been pushed into the aspect (See **Figure 27**), which imposes the transformation when it is required to.

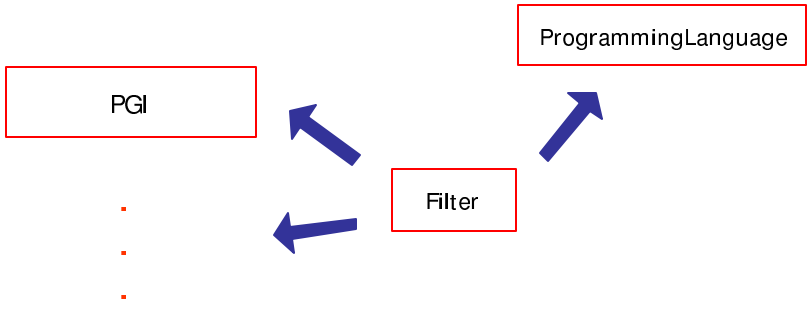

Figure 27: After reengineering, only Filter must know the required format, regardless of the number of classes that call ProgrammingLanguage.load.

This aspect was implemented as one aspect per conversion for convenience. These two aspects share a repository though, and these two filter aspects could have been implemented as one. This would have simplified the design by allowing fewer pointcuts, and allowing all **load** calls to be checked for conversion. The two-class design is more reusable, however, since it does not require a shared repository.

7 Conclusion

The original goal of this project was to reengineer the Cocoon code to gain insight, both quantitative and qualitative, into the effect of AspectJ on modularity. During this process

a total of nine aspects were created, and these were grouped into three different categories. In all of these cases, modularity was improved and the resultant code showed less *code tangling* and *"Knows About" tangling*.

I. AspectJ Patterns

An important result of this paper is that the Aspects in the reengineered Cocoon code fit into three generic categories. These three categories provide patterns for the use of AspectJ in other systems, and guidance to AspectJ programming.

- The Reverse Inheritance (RI) type should be used when functionality separate from the object's functionality is desired. When this behavior is fairly generic across a body of code, a large reduction in complexity and code size can be gained. This is because the Reverse Inheritance type fixes both *code tangling* and *"Knows About" tangling*.
- The Layered type should be used for optional or special case behavior. When this behavior has been amalgamated with the standard behavior, AspectJ can be used to separate the special cases into Aspects. This type of Aspect also fixes both *code tangling* and *"Knows About" tangling*.
- The Filter type can be used to make type to type transformations transparent. This decreases the complexity of code by making the Aspect, instead of the objects, responsible for the transformation. This type of aspect fixes *"Knows About" tangling*.

Another pattern was discovered over the course of this project as well: a usage pattern for cflow. In the PGICache, cflow was used to mirror method local variables for use in the Aspect. AspectJ only allows access to object variables but using cflow, the Aspect gained access to method local variables. Thus, cflow can be used to capture method local variables, and AspectJ can behave in a functional capacity, creating objects and executing code on a per-method-call basis.

II. AspectJ provides Reverse Inheritance

The successful use of AspectJ for the Reverse Inheritance highlights another discovery. AspectJ classes can be used to introduce behavior into a class without tangling the *"Knows About"* relationship. This is because Reverse Inheritance can be used to *impose* behavior on an object, confining the "Knows About" relationship to the Aspect introducing the behavior.

In searching for aspects in Cocoon, it was discovered that many classes inherited behavior from multiple sources. Secondary behavior was often transparent and unrelated to the behavior of the inheriting (and sometimes implementing) class. Using AspectJ, we can change the implementation so that there is no *code tangling* (because code is implemented in the Aspect), and no *"Knows About" tangling* (because it is the Aspect that contains the "Knows About" relationship.

III. Changing the Knows About Relationship

This project also verified that AspectJ is successful in improving the *"Knows About"* relationship in a project. AspectJ both improves the directionality (leading to Aspects "Knowing About" and introducing behavior), and localizes the "Knows About" relationship. This is especially pronounced in the **NamedComponent** Aspect where potentially N "Knows About" relationships are removed, in addition to other improvements. However, even in Aspects with fewer gains, the "Knows About" relationship is pushed into the Aspect. This leads to a reduction in *"Knows About" tangling*. Components are no longer required to know about behaviors unrelated to their own.

IV. Layered Programming

Another discovery was the ease with which AspectJ allows layered extension programming [5], or subset programming. This is seen in the Layered Aspect. Once the minimal subset is found and coded, the other behaviors can be layered on using AspectJ. The Layered Aspect in this project also highlighted the difficulties one can encounter in finding the minimal subset of behavior. In the Layered Aspect, it was unclear whether the cache, disk, or regeneration of code should be the default behavior. While this difficulty was not resolved, it is clear that AspectJ provides a mechanism for implementing any of these as default behavior.

Several other observations were made during the course of this project that are mentioned only briefly here. The first is that Aspects were easy to find. Several mechanisms used in the code pointed to easy conversion to AspectJ. Examples of these are the use of many interfaces, trivial abstract classes whose behaviors were implemented many times (often with the same functionality), and the division of Cocoon into a management package (Avalon) and a functional package (Cocoon). Interfaces were used to introduce tangential behaviors (defined by trivial abstract classes), or "Aspects" of an object. Most of these interfaces were candidates for conversion to RI Aspects, though not all were simple candidates (i.e. mixed functionality made it hard to separate out the functionality of the abstract class alone). Even where interfaces were not used, the layout of code and method calls clearly indicates that some behaviors are "Aspects", and not primary functionality. It appears that Aspects are an abstract structure in the minds of the programmers, even if they cannot be explicitly implemented. The second observation is that AspectJ often allowed the conversion of existing code into Aspects with minimal effort. This appears to be both because Aspects were easy to find and because they were simply re-implementing behavior. The final observation is that the tools for AspectJ works well for "real" code bases. This includes the compiler itself (though slow), as well as the development environment (AJDE Emacs) and the Aspect Browser.

Acknowledgements
The author would like to thank William G. Griswold, Cristina Videira Lopes, the AspectJ group, and Xerox PARC.

References

1. Recent Developments in AspectJ™
Cristina Videira Lopes and Gregor Kiczales
In ECOOP'98 Workshop Reader, Springer-Verlag LNCS 1543.
Corresponding AspectJ version*: pre 0.1

2. A Study on Exception Detection and Handling Using Aspect-Oriented Programming
Martin Lippert and Cristina Videira Lopes
In proceedings of ICSE'2000. Limmerick, Ireland. Corresponding AspectJ version*: 0.4

3. Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming
Mik Kersten and Gail C. Murphy
In OOPSLA'99 proceedings. Denver, CO, USA. ACM Press, pp. 340-352, 1999.
Corresponding AspectJ version*: 0.2

4. Personal Conversation with Jeff Palm
Jeffery Palm
Xerox PARC Summer 2000

5. The Structure of the ``THE''-Multiprogramming System
E.W. Dijkstra,.
Communications of the ACM, 11(5), 1968.

6. The Cocoon Project
The Apache Software Foundation
http://xml.apache.org/cocoon/index.html

7. The Apache Project
The Apache Software Foundation
http://www.apache.org