Unplugging Components using Aspects

Sergei Kojarski David H. Lorenz*
Northeastern University
College of Computer & Information Science
Boston, Massachusetts 02115 USA
{kojarski,lorenz}@ccs.neu.edu

ABSTRACT

Normally, Aspect-Oriented Programming (AOP) is used for plugging and not for unplugging. Aspects (concerns) are weaved into a system, thus providing a modularized way to add crosscutting features to a base program. For example, given a base program without logging and given a logging aspect, the AOP weaver produces from the untangled code an executable with the desired crosscutting logging behaviors. The benefit is that the logging code is modularized, e.g., you can easily activate or deactivate logging. However, what if a legacy system written without AOP technology already contains logging functionality hard-coded in the base program? How do you deactivate logging then? In this paper, we explore the use of AOP for writing aspects that transform exiting hard-coded calls into plugs that can be used to retarget the client to use other subcomponents. Applying AOP for unplugging has potential usage in connection with components. You are given a monolithic system. You apply AOP to non-intrusively decouple the system's components. You may then replace some of the legacy components with alternative third-party components.

1. AOP FOR UNPLUGGING

Consider a client that is strongly coupled with a component C_1, for which third-party alternatives, $C_2, C_3, \ldots, C_n$, exist. Presumably, $C_1, C_2, \ldots, C_n$ present a spectrum of trade-offs for the client to choose from, e.g., on certain methods, C_2 or C_3 may be more efficient than C_1. Suppose that the client calls methods m_1, m_2, and m_3 directly on C_1, and that C_2 and C_3 were developed independently by third-parties and provided without their source code [13]. AOP [4] can help to selectively unplug the client from certain calls to C_1, and plug those calls back into C_2 or C_3 (Figure 1).

Normally, one would need to change the code of the client or the code of the components. Changing the client code is intrusive and the result is not retargetable. Changing the component is also undesired, because it will affect other clients too. Moreover, it may be impossible to change a third-party component.

Instead one can employ AOP to achieve pluggability. We first motivate this approach using the standard logging example in AspectJ [6, 3]. In Section 2 we illustrate this approach for the complex case of retargeting a client of reflection in Java [1].

1.1 Logging Unplugged

The components of the system may already be coupled and their code tangled. Consider a legacy system where the client code (class

*Supported in part by the National Science Foundation (NSF) under Grants No. CCR-0098643 and CCR-0204432, and by the Institute for Complex Scientific Software at Northeastern University.

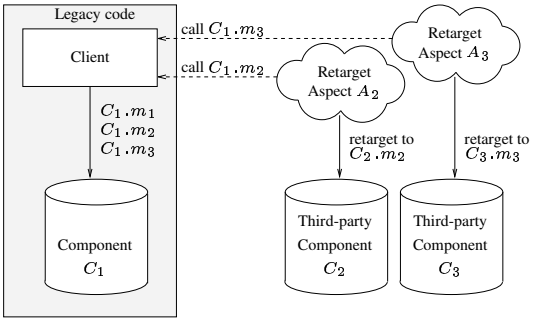

Figure 1: Retargeting

`Tangled` in Listing 1) is tangled with the logging code (hard-coded calls to `System.out.println`). Using AOP, we can separate them, providing the following benefits:

1. Ability to activate or deactivate the logging code.
2. Ability to modify the logging, e.g., spell check the messages before printing, or translating the messages to a different language.
3. Ability to add to the message reflective information from the join point, e.g., line number (lexical information), receiver's type (static information), or receiver's identify (dynamic information).
4. Ability to transform explicit-invocation to event-based implicit invocation, thus allowing pluggability in a component builder.

The "decoupling" `Unlogger` aspect (Listing 2) uses an **`around`** advice to divert calls from `println` to `Logger.log` (Listing 3).[1] The `Logger` class (Listing 3) extends `TextEventNotifier` (Listing 5), providing interested `TextListener`s (Listing 4) with the logging message as a `java.awt.event.TextEvent` event. Thus, transforming the explicit invocation of `log` into implicit invocation [10] of `textValueChanged`.

As a result, adding a TRANSCEIVER [7] component between the client and the logging facility, e.g., a spell checker (Listing 6), can be easily achieved.

[1]We assume knowledge of AspectJ [3].

Listing 1: Tangled.java

```
package program;

public class Tangled {

 public void method1() {
  System.out.println("method 1 is called");
 }

 public void method2() {
  System.out.println("method 2 is called");
 }

 public void method3() {
  System.out.println("method 3 is called");
 }

 public static void main(String[] args) {
  Tangled t = new Tangled();
  t.method1();
  t.method2();
  t.method3();
 }
}
```

Listing 2: Unlogger.java

```
package aspects;
import connector.Logger;
import connector.TextEventNotifier;

aspect Unlogger {

 void around(String message,Object source):
  call(void java.io.PrintStream.println(String))
  && args(message) && target(source) {
   logger.log(message,source);
 }

 public static TextEventNotifier
     getTextEventNotifier() {
   return Unlogger.aspectOf().logger;
 }

 private Logger logger = new Logger();
}
```

Listing 3: Logger.java

```
package connector;

public class Logger extends TextEventNotifier {

 public void log(String message,Object source) {
  LogTextEvent event = new LogTextEvent(message,
      source);
  fireTextEvent(event);
 }
}
```

Listing 4: TextListener.java

```
package connector;
import java.awt.event.TextEvent;

public class TextListener implements java.awt.
    event.TextListener {

  public void textValueChanged(TextEvent e) {
    System.out.println(e.paramString());
  }
}
```

Listing 5: TextEventNotifier.java

```
package connector;
import java.awt.event.TextEvent;
import java.awt.event.TextListener;
import java.util.Vector;

public class TextEventNotifier {

 public void addTextListener(TextListener listener
     ) {
  if (!listeners.contains(listener)) listeners.add
      (listener);
 }

 public void removeTextListener(TextListener
     listener) {
  listeners.remove(listener);
 }

 void fireTextEvent(TextEvent event) {
  for (int i=0;i<listeners.size();i++)
   ((TextListener)listeners.get(i)).
       textValueChanged(event);
 }

 private Vector listeners = new Vector();
}
```

Listing 6: SpellChecker.java

```
package connector;
import java.util.Vector;
import java.awt.event.TextListener;
import java.awt.event.TextEvent;

public class SpellChecker
 extends TextEventNotifier
 implements TextListener {

 public void textValueChanged(TextEvent e) {
  String checked = spellCheck(e.paramString());
  LogTextEvent event = new LogTextEvent(checked,e.
      getSource());
  fireTextEvent(event);
 }

 private String spellCheck(String str) {
  // spell checking omitted
  return str;
 }
}
```

Listing 7: MetaClient.java (Core reflection)

```
import java.util.Vector;
import java.lang.reflect.Field;

class MetaClient {

 public Object[] getFields(Object host) {
  if (host==null) return null;
  Vector result = new Vector();
  collectFields(host.getClass(),result,host);
  return result.toArray();
 }

 private void collectFields(Class cl,Vector result
     ,Object host) {
  if (cl==null) return;
  Field[] declared = cl.getDeclaredFields();
  for(int i=0;i<declared.length;i++)
   try{
    result.add(declared[i].get(host));
   } catch (IllegalAccessException e) {}
  collectFields(cl.getSuperclass(),result,host);
 }
}
```

Listing 8: MetaClient.java (Mirrored reflection)

```
import java.util.Vector;
import edu.neu.ccs.mirror.java.lang.Class;
import edu.neu.ccs.mirror.java.lang.reflect.*;

class MetaClient {

 public Object[] getFields(Object host) {
  if (host==null) return null;
  Vector result = new Vector();
  collectFields(Class.getClass(host),result,host);
  return result.toArray();
 }

 private void collectFields(Class cl,Vector result
     ,Object host) {
  if (cl==null) return;
  Field[] declared = cl.getDeclaredFields();
  for(int i=0;i<declared.length;i++)
   try{
    result.add(declared[i].get(host));
   } catch (IllegalAccessException e) {}
  collectFields(cl.getSuperclass(),result,host);
 }
}
```

2. RETARGETING REFLECTION

For a more complex non-trivial example, consider the `MetaClient` code (Listing 7)—a client of reflection [11, 12, 9].

A first step to achieve pluggable reflection [8] is to convert from core to a mirrored reflection by changing the imports, and by changing `host.getClass()` to `Class.getClass(host)`. Making the transition to mirrored reflection is required because classes in the `java.lang.reflect` package and the `java.lang.Class` class are final and because only the JVM can instantiate these classes. The mirrored reflection has a default implementation, which falls back on Java [1] Core Reflection to provide the reflective information. Basically, it acts as a wrapper [2] of Java Core Reflection (for an overview see [8]).

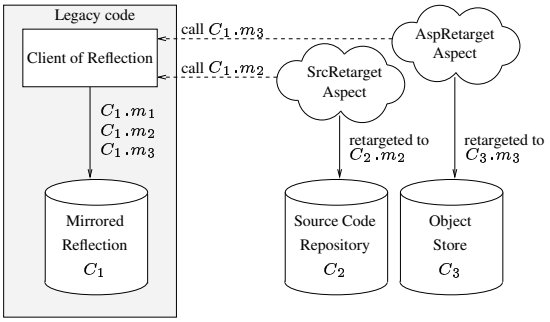

Figure 2: Retargeting a client of reflection

For the remainder of the paper, we shall assume that the client code is written against `edu.neu.ccs.mirror.java.lang.Class` and `edu.neu.ccs.mirror.java.lang.reflect.*`. The corrected code of `MetaClient` is shown in Listing 8.

2.1 Goal

Given a client that uses the mirrored reflection component C_1, the goal in this example is to non-intrusively retarget the client to use alternative providers of meta-information, C_2 and C_3 (Figure 2). Specifically, let C_2 be a source code repository capable of providing static class-level reflective information. Assume C_2 has complete static information but no dynamic information. That is, C_2 implements the reflection interface on top of a source-code repository; C_2 presents an alternative component to C_1 for class-level functionality; but C_2 cannot provide information which is not available statically in the source code.

Let C_3 be an object store capable of providing dynamic object-level reflective information, such as values of fields. Assume C_3 has complete dynamic information but very limited static information. For example, the type of each object (i.e., the name of its class) is known in C_3, but the inheritance hierarchy in unknown. That is, C_3 presents an alternative component to C_1 for object-level functionality; but C_3 has incomplete static information, and the limited static information it has is associated with the dynamic functionality.

Next, we show that aspects can retarget clients of C_1 to use either C_2 or C_3 or both as an alternative source of meta-information.

2.2 Retargeting class-level reflective calls to C_2

`Class` objects are the gateway to class-level reflective information. There are exactly two methods in the reflection API that allow clients to obtain a handle to a `Class` object, namely:

1. `Class.forName(str,...);`
2. `Class.getClass(obj).`

Two aspects do the job of retargeting:

`classstore.ForName` (Listing 9). Given a class name, the method `Class.forName(str,...)` returns the `Class` instance for the class named `str` . Since classes are represented in the source-code repository, class names can be mapped to a corresponding class representation in C_2 by the method `Class-Repository.forName(name)`. The **`around`** advice found

Listing 9: classstore.ForName.java

```
package edu.neu.ccs.mirror.retarget.classstore;
import edu.neu.ccs.mirror.java.lang.reflect.*;
import edu.neu.ccs.mirror.java.lang.Class;
import edu.neu.ccs.mirror.repository.
    ClassRepository;

aspect ForName {

 Object around(String name):
  && call(public static Class Class.forName(String
      ,..))
  && args(name) {

   try{
    // return ClassRepository−based Class instance
    return ClassRepository.forName(name);
   } catch (Exception e) {
    return proceed(name);
   }
 }
}
```

Listing 10: classstore.GetClass.java

```
package edu.neu.ccs.mirror.retarget.classstore;
import edu.neu.ccs.mirror.java.lang.reflect.*;
import edu.neu.ccs.mirror.java.lang.Class;

aspect GetClass {

 Class around(Object obj):
  && call(public static Class Class.getClass(
      Object))
  && args(obj) {

   try{
    Class cl = proceed(obj);
    // return ClassRepository−based Class instance
    return Class.forName(cl.getName());
   } catch (Exception e) {
    return proceed(obj);
   }
 }
}
```

in `classstore.ForName` intercepts calls to `Class.forName(name)` and redirects them to `ClassRepository.forName(name)` to return a `ClassRepository`-based `Class` instance instead.

`classstore.GetClass` (Listing 10). Given an object `obj`, the method `Class.getClass(obj)` returns a `Class` instance corresponding to the type of `obj`. Here, the source code repository cannot be used directly, because it has no knowledge of run-time values. Therefore, in the **`around`** advice, calls to `Class.getClass(obj)` are forwarded to the default underlying reflective mechanism via **`proceed`**`(obj)`. However, the class instance returned by **`proceed`**`(obj)` needs to be converted to the C_2-based class representation. This is accomplished by extracting the class name and using `Class.forName(String)`. Like the calls in the base program, the call to `Class.forName(String)` in Listing 10 is also intercepted by the `classstore.ForName` aspect. Consequently, the desired class representation in C_2 (`ClassRepository`-based `Class` instance) is returned.

Listing 11: objectstore.GetClass.java

```
package edu.neu.ccs.mirror.retarget.objectstore;
import edu.neu.ccs.mirror.java.lang.reflect.*;
import edu.neu.ccs.mirror.java.lang.Class;
import edu.neu.ccs.mirror.aspects.internal.Type;
import edu.neu.ccs.mirror.aspects.internal.Value;
import edu.neu.ccs.mirror.aspects.internal.Store;

aspect GetClass {

 Object around(Object obj):
  && call(public static Class Class.getClass(
      Object))
  && args(obj) {
  // obtain internal representation for host
  Value val = Store.getValue(obj);
  // obtain internal host type
  Type type = val.getType();
  // return Class provided by forName
  return Class.forName(type.getName());
 }
}
```

2.3 Retargeting object-level reflective calls to C_3

Object-level introspection is provided in reflection via two methods:

1. `Field.get(obj)`

   This method reflects the object graph by mapping the receiver `Field` meta-object and the argument `obj` object to the value of the corresponding field in `obj`.

2. `Class.getClass(obj)`

   Finding the type of an object using the static method `Class.getClass(obj)` in C_1 (which is the equivalent to `obj.getClass()` in core reflection.)

Two aspects do the job of retargeting:

`objectstore.GetClass` (Listing 11). The `GetClass` aspect intercepts calls to `Class.getClass(Object)`. The mapping from objects to their class names that is supported by the object store can be utilized in `Class.getClass(Object)`. To achieve that, `Class.getClass(Object)` is implemented in two steps. First, the class name of the object argument is found in C_3. Second, the returned class name is used in `Class.forName(String name,...)` to provide the desired result.

`objectstore.FieldGet` (Listing 12). The `FieldGet` aspect intercepts calls to `Field.get(Object)`. The object store has complete object-level reflective information. Therefore, The **`around`** advice in the `FieldGet` aspect simply retrieves the information from the object store.

Note that we only retarget calls, but the underlying classes are the same, i.e., we only replaced the functionality. The client still uses C_1 but two specific pieces of functionality are performed by C_3 (class names of objects, and values of fields.)

Listing 12: objectstore.FieldGet.java

```
package edu.neu.ccs.mirror.retarget.objectstore;
import edu.neu.ccs.mirror.java.lang.reflect.*;
import edu.neu.ccs.mirror.java.lang.Class;
import edu.neu.ccs.mirror.aspects.internal.Store;
import edu.neu.ccs.mirror.aspects.internal.
    ObjectReference;
import edu.neu.ccs.mirror.aspects.internal.
    ReferenceType;

aspect FieldGet {

 Object around(Field field,Object host):
  call(Object Field.get(Object))
  && args(host)
  && target(field) {
  // obtain aspectual store representation for host
  ObjectReference internalHost =
   (ObjectReference)Store.getValue(host);
  // convert field to edu.neu.ccs.mirror.aspects.internal.Field
  ReferenceType enclosingType =
   (ReferenceType)Store.getType(field.
       getDeclaringClass().getName());
  edu.neu.ccs.mirror.aspects.internal.Field
      internalField =
   enclosingType.getDeclaredField(field.getName())
       ;
  // obtain value from aspectual repository
  return internalField.getFieldValue(internalHost)
      .getObjectVal();
 }
}
```

2.4 Aspect Composition

There are two independent third-party components: `classstore` and `objectstore`. Each component comes with two aspects, which must be applied in concert: `ForName` and `classstore.GetClass` are always used together; similarly, `objectstore.GetClass` and `FieldGet` are always used together. The two pairs of aspects, however, are designed to work independently.

The combined effect of applying the two components concurrently is particularly interesting, because

- their aspects target the same set of join points: both `classstore.GetClass` and `objectstore.GetClass` independently advise calls to `Class.getClass(Object)`;
- `classstore.ForName` intercepts calls to `Class.forName` not only in the base program, but also in the `objectstore.GetClass` and in the `classstore.GetClass` aspects.

Generally, there are two explicit mechanisms powering aspect interaction.

Advising join points within another aspect's advice One possible scenario is that calls to `Class.getClass(Object)` in the base program are intercepted by the `objectstore.GetClass` aspect. That aspect first looks up the argument object class name in the object store. Then, the `Class` instance is obtained from `Class.forName(String)` and returned. When both the class and the object store aspects are applied simultaneously, the call to `Class.forName(String)` is always intercepted by the `classstore.ForName`. This collaboration guarantees that the `objectstore.GetClass` aspect correctly returns the `ClassRepository`-based `Class` instance.

Proceeding to another aspect's advice The second scenario is the dominance of `classstore.GetClass` over `objectstore.GetClass`. In this case, calls to `Class.getClass(Object)` trigger the **`around`** advice in `classstore.GetClass`. The collaboration is achieved by calling **`proceed`** for forwarding the control to the **`around`** advice in `objectstore.GetClass`, which in turn executes as explained above and correctly returns a `ClassRepository`-based `Class` instance.

In sum, handling `Class.getClass(Object)` calls involve the functionality of both C_2 and C_3, unplugging C_1 completely.

3. CONCLUSION

This paper presented the use of AOP to weave plug points into monolithic code, allowing componentization (unplugging) of application code.

Using the well known AOP example of logging, we illustrated that aspects can be applied for untangling rather than entangling. Using the reflection example, we have demonstrated that a naive client does not realize that it uses a different source of meta-information. The client thinks it is using C_1 and is unaware that it is in fact using C_2 and C_3. Moreover, C_2 doesn't realize that it uses C_3; and C_3 doesn't know that it uses C_2. This is crucial for achieving pluggability.

The benefit of having a mirrored reflection and the implementation of pluggable reflection without AOP techniques are explained elsewhere [8]. In [5] we illustrate that AOP can serve as an object store which can offer an alternative source of dynamic meta-information. In this paper, we build on these two works and show that AOP can also help in implementing pluggable reflection.

More generally, aspects can help unplug even strongly coupled components and help retarget calls to alternative third-party components. There are various degrees of coupling. If the client uses the implementation directly, the two are strongly coupled. If both rely only on interfaces—it is a weak coupling: client and implementation may evolve separately as long as the interface doesn't change. In component-based programming, interfaces are discovered using introspection and adaptation allowing also third-party components to be connected. Aspects could help migrate strongly coupled components to pluggable components.

By drawing on the code-transformation abilities of AOP, we showed that the established techniques (as found in AspectJ) can be used to transform in "the other way" from the aspectually woven to the unwoven (unplugged) code. This idea may be even more useful then AOP in its usual form: this unusual transform approach can help to normalize and refactor existing code.

4. REFERENCES

[1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison–Wesley Publishing Company, 1996.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*.

Professional Computing. Addison-Wesley, 1995.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 18-22 2001. ECOOP 2001, Springer Verlag.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 9–13 1997. ECOOP'97, Springer Verlag.

[5] S. Kojarski and D. H. Lorenz. Reflective mechanisms in AOP languages. Technical Report NU-CCIS-03-07, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Mar. 2003.

[6] C. V. Lopes and G. Kiczales. Recent developments in AspectJ. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology. ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science, pages 398–401. Workshop Proceedings, Brussels, Belgium, Springer Verlag, July 20-24 1998.

[7] D. H. Lorenz and J. Vlissides. Designing components versus objects: A transformational approach. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 253–262, Toronto, Canada, May 12-19 2001. ICSE 2001, IEEE Computer Society.

[8] D. H. Lorenz and J. Vlissides. Pluggable reflection: Decoupling meta-interface and implementation. In *Proceedings of the 25th International Conference on Software Engineering*, pages 3–13, Portland, Oregon, May 1-10 2003. ICSE 2003, IEEE Computer Society.

[9] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 147–155, Orlando, Florida, Oct. 4-8 1987. OOPSLA'87, ACM SIGPLAN Notices 22(12) Dec. 1987.

[10] M. Shaw and D. Garlan. *Software Architecture, Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[11] B. C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT LCS TR-272, Jan. 1982.

[12] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 11th ACM SIGPLAN symposium on Principles of programming languages*, pages 23–35, 1984.

[13] C. Szyperski. *Component-Oriented Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.