Autonomics Development: A Domain-Specific Aspect Language Approach, 41–66
Book Series: Autonomic Systems
© 2010 Springer Basel AG

3 An Aspect-Oriented Approach

3.1 Introduction

Object-orientation has been presented as the technology that will finally make software reuse a reality as the object model provides a better fit with domain models than procedural programming [63]. Object-orientation, currently the dominant programming paradigm, allows a programmer to build a system by decomposing a problem domain into objects that contain both data and the methods used to manipulate the data, thereby providing both abstraction and encapsulation. In addition, object-oriented languages typically provide an *inheritance* mechanism that allows an object to reuse the data and methods of its parent, thereby enabling polymorphism.

There are, however, many programming problems where the object-oriented programming (OOP) technique, or the procedural programming technique it replaces, are not sufficient to capture the important design decisions a program needs to implement. Kiczales et al. [59] refer to these design decisions as *aspects* and claim the reason they are so difficult to capture is because they crosscut the systems basic functionality. Kiczales et al. claim that AOP makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code.

3.2 Crosscutting Concerns and Aspects

Separation of concerns has long been a guiding principle of software engineering as it allows one to identify, encapsulate and manipulate only those parts of software that are relevant to a particular goal or purpose [79]. Unfortunately, there can be many concerns that crosscut one another leading to tangled code that is difficult to understand, reuse and evolve. Concerns are said to crosscut if the methods related to those concerns intersect [31] as illustrated in the UML for a simple picture editor[1] in Figure 3.1.

Figure 3.1 illustrates two implementations of the `FigureElement` interface, `Point` and `Line`. Although these classes exhibit good modularity, consider the concern that the screen manager must be notified whenever a `FigureElement` moves. In this case every time a `FigureElement` changes, the screen manager must

[1]This example is reproduced from Kiczales et al. [60].

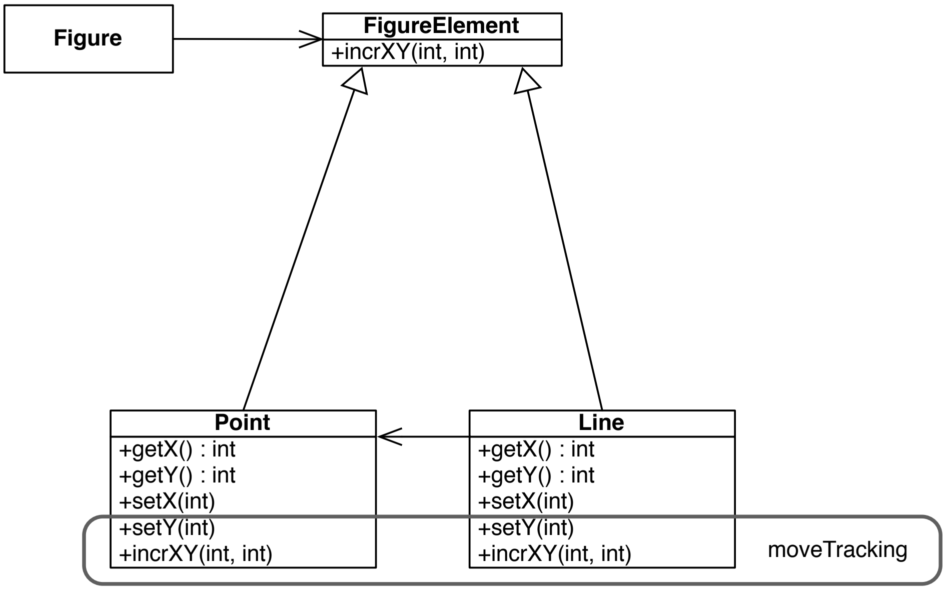


FIGURE 3.1. Aspects crosscut classes in a simple figure editor.

be informed by calling the screen manager's `moveTracking` method, as illustrated by the band surrounding the methods that implements this concern in Figure 3.1. This concern is called a *crosscutting concern* as it crosscuts methods in both the `Line` and the `Point` classes.

AOP is an attempt to isolate and modularise these concerns and then weave or compose them together with an existing program, thereby allowing the concern to be applied in an oblivious way (the existing code is unaware of the crosscutting concern). An aspect can therefore be considered as a modular unit of crosscutting implementation [61].

It is important to note that the goal for AOP is not as a replacement for object-orientation, it is to build on object-orientation by supporting separation of concerns that cannot be adequately expressed in object-oriented languages [31].

3.3 AOP Semantics

AOP introduces new semantics to describe crosscutting concerns and aspects. Much of the semantic model is based on the AspectJ language developed by Kiczales et al. [59] at Xerox's Palo Alto Research Centre.

According to Kiczales et al. [59], aspect-oriented languages have three critical elements: a join point model, a means of identifying join points, and a means of effecting implementation at join points. These elements can be described as:

Join points. A join point model makes it possible to define the structure of crosscutting concerns. Join points are well-defined points in the execution flow of a program [59], such as method calls, constructors, function calls etc. Join points can therefore be considered as places in a program where aspects may be applied.

Pointcuts. A pointcut is a means used to identify a join point. This is typically a filter mechanism that defines a subset of join points [59]. A `cflow` is a type of pointcut that identifies join points based on whether they occur in the dynamic context of other join points. For example, the `cflow` statement `cflow(move())` in the AspectJ language picks out each join point that occurs between when the `move()` method is called and when it returns, which may occur multiple times in the case of a recursive call [6].

Advice. The advice is used to define additional code that is run at join points. In most AOP languages there are a number of different advice declarations that define when the advice runs when a join point is reached. These are typically before, after, or around the join point.

Weaving. The process of adding aspects to existing code to produce a coherent program is known as *weaving*. Weaving is either done at compile time (static weaving) or at runtime (dynamic weaving) and some systems, such as composition filters [12], allow aspects to be added and removed dynamically. Weaving is achieved using a technique known as bytecode rewriting, which alters existing bytecode, either dynamically as it is loaded or statically by altering the bytecode contained in an existing class file.

As well as the above general AOP semantics, various aspect implementations provide their own semantics.

3.4 Static and Dynamic Weaving

Static weaving refers to the modification of the source code of a class by inserting aspect-specific statements at join points [21]. Java applications are compiled to bytecode, a portable format that is interpreted by the Java virtual machine at runtime, and consequently most Java AOP systems alter the bytecode, not the source code. This has the added advantage of allowing Java AOP systems to be used where the source code is not available.

Dynamic weaving refers to the ability to weave and unweave aspects at runtime without having to restart the application [84].

A number of methods have been proposed to implement aspect-oriented functionality. These can be broadly classified as language-based implementations, framework-based implementations and domain-specific language implementations.

3.5 Language-Based Implementations

A number of language-based approaches have been proposed to implement aspect-oriented programming. Many of these languages, such as AspectJ [60], Caesar [71], Jiazzi [68], AspectC [20], and JAsCo [106] have been designed as extensions to existing languages. Other novel language-based concepts, such as composition filters that manipulate the messages passing between objects, have also been proposed [12].

These languages and systems have typically been produced by the research community attempting to understand the practical value of AOP in terms of how aspects are used, the types of designs and patterns that may emerge, and how effective crosscutting modularity actually is [60]. The most popular language-based system is currently AspectJ.

3.5.1 AspectJ

AspectJ is an extension to the Java programming language that was developed by Kiczales et al. [59] at Xerox's Palo Alto Research Centre. The AspectJ language is designed to be a simple and practical aspect-oriented extension to the Java language that can be used to code crosscutting concerns that would otherwise lead to tangled code [60].

AspectJ is designed as a compatible extension to Java where compatibility is defined by Kiczales et al. [60] as:

- *Upward compatibility* – all legal Java programs are legal AspectJ programs.
- *Platform compatibility* – all legal AspectJ programs must run on standard Java virtual machines.
- *Tool Compatibility* – all existing tools, including IDEs, documentation tools, and design tools should be able to be extended to support AspectJ.
- *Programmer compatibility* – programming in AspectJ must feel like a natural extension to programming in Java.

While early versions of AspectJ operated on source code, later versions alter the bytecode generated by the Java compiler, thereby allowing aspects to be used in situations where the source code is not available.

As discussed in Section 3.3 join points are well-defined points in the execution flow of a program. AspectJ supports a number of join points as listed in Table 3.1 [60].

A pointcut in AspectJ is a set of join points that may be matched at runtime. For example, the pointcut[2]:

```
call(void Point.setX(int)) ||
call(void Point.setY(int))
```

[2]These examples are reproduced from Kiczales et al. [60, 61].

TABLE 3.1. The dynamic join points of AspectJ.

Kind of join point	*Points in the program execution at which ...*
method call constructor call	A method (or a constructor of a class) is called. Call join points are in the calling object, or in no object if the call is from a static method.
method call reception constructor call reception	An object receives a method or constructor call. Reception join points are before method or constructor dispatch, i. e. they happen inside the called object, at a point in the control flow after control has been transferred to the called object, but before any particular method or constructor has been called.
method execution constructor execution	An individual method or constructor is invoked.
field get	A field of an object, class or interface is read.
field set	A field of an object or class is set.
exception handler execution	An exception handler is invoked.
class initialisation	Static initialisers for a class are run.
object initialisation	When the dynamic initialisers for a class are run during object creation.

matches any call to either the `setX` or `setY` methods defined by `Point` that return `void` and have a parameter of `int`. Pointcuts may also be declared by name, for example:

```
pointcut weAreMoving():
    call(void Point.setX(int)) ||
    call(void Point.setY(int));
```

As well as pointcuts that match an explicit method call, as described above, pointcuts may also contain *wildcard* characters that can match a number of different methods. Consider the following:

```
call (public String Figure.get*(..))
call (public * Figure.*(..))
```

The first matches any call to public methods defined in `Figure` that start with `get`, take any number of parameters, and return a `String`. The second matches any call to a public method defined in `Figure`.

AspectJ defines the *advice* declaration to stipulate the code that is run at a join point. Three types of advice are supported:

```
aspect SimpleTracing {
  pointcut traced() :
      call(void Display.update()) ||
      call(void Display.repaint(..));

      before(): traced() {
        println("Entering: " + thisJoinPoint);
      }

      void println(String s) {
         // write message
      }
}
```

FIGURE 3.2. AspectJ `SimpleTracing` Example.

- *Before* advice – runs at the moment a join point is reached.
- *After* advice – runs after the join point has been reached.
- *Around* advice – runs when the join point is reached and has explicit control over whether the method is run or not.

An advice is declared using one of the advice keywords. For example, the following advice prints a message after the `weAreMoving` method is called using the `after` keyword:

```
after(): weAreMoving() {
  System.out.println("We have moved");
}
```

Aspects wrap up pointcuts, advice, and inter-type declarations in a modular unit of crosscutting implementation and is defined similar to a class. Inter-type declarations are members of an aspect (fields, members and constructors) that are owned by other types, and aspects can also declare that other types implement new interfaces or extend a new class [6].

Aspects can contain methods, fields, and initialisers in addition to the crosscutting members. The following is an example of a simple aspect that is used to print messages before certain display operations:

As can be seen from the example in Figure 3.2, aspects in AspectJ are not reusable because the context on which an aspect needs to be deployed is specified directly in the aspect definition – the pointcut is part of the aspect [106]. Although AspectJ allows aspects to be inherited from other aspects, it is only allowed if the inherited aspect has been declared as *abstract*. Concrete aspects are therefore not reusable. In addition, any change to a class may result in the necessity to alter the aspect as the pointcut definition may no longer be valid.

Recognising this limitation, a number of researchers have been focusing on removing the join point interception model from the aspect implementation. Lieberherr et al. [64] have developed the concept of Aspectual Components, which attempts to separate the pointcut from the advice, thereby making the advice reusable. This method has subsequently been adopted by Suvée et al. [106] in the JAsCo language. The Caesar system [71], uses an *aspect collaboration interface*, a higher-level module concept that decouples the aspect implementation from the aspect bindings, to enable reuse and componentisation of aspects.

As well as providing aspects, AspectJ also provides *introductions*, a mechanism for adding fields, methods and interfaces to existing classes. Introductions are motivated by the observation that concerns have an impact on the type structure of programs which compromises modularisation, because different fields and methods in the type structure may come from different concerns. With introductions, these fields and methods can be removed from the various concerns, modularised, and applied to the various classes at runtime [46].

3.6 Framework-Based Implementations

One approach to the implementation of AOP is by providing an object-oriented framework [80], a reusable semi-complete application that can be specialised to produce custom applications [55]. A framework dictates the architecture of an application by [41]:

- Defining its overall structure.
- Partitioning the application into classes and objects.
- Defining the key responsibilities of the classes and objects.
- Dictating how the classes and objects collaborate.
- Defining the application's thread of control.

These design parameters are predefined so that an application programmer can concentrate on the specifics of the application and not on the architecture [41].

The important classes in a framework are usually abstract. An abstract class is a class with no instances and is used only as a superclass [36]. As well as providing an interface, an abstract class provides part of the implementation of its subclasses by using either a template method or a hook method. A template method defines part of the implementation in an abstract class and defers other parts to subclasses by calling methods that are defined as abstract [41]. A hook method defines a default implementation that can be overridden by subclasses [86]. Abstract classes that are intended to be subclassed by the framework user are known as hot spots as they encapsulate possible variations [85].

Frameworks usually come with a component library containing concrete subclasses of the abstract classes in the framework [36]. According to Fayad et al. [36], frameworks provide the following benefits:

Modularity. Implementation details are hidden behind stable interfaces. This helps to improve quality by localising the impact of design and implementation changes.

```
 1  public class MyAspect {
 2    public Object trace(MethodInvocation invocation)
 3          throws Throwable {
 4      try {
 5         System.out.println("Entering method");
 6         // proceed to next advice or actual call
 7         return invocation.invokeNext();
 8      } finally {
 9         System.out.println("Leaving method");
10      }
11    }
12  }
```

FIGURE 3.3. JBoss aspect example.

Reusability. The stable interfaces provided by frameworks define generic components that can be reused in new applications.

Extensibility. A framework provides hook methods that can be re-implemented by subclasses.

Inversion of control. This allows the framework, as opposed to the application, to decide which application specific methods to invoke in response to external events.

Frameworks are designed to either be used as a general purpose framework usable in any environment, such as AspectWerkz [16], or to be used in a specific environment, such as the JEE framework. Components that have been developed to a specific framework environment cannot be reused outside that environment, severely limiting reuse.

3.6.1 The JBoss AOP Framework

The JBoss AOP framework provides a framework that can be used to develop aspect-oriented applications that are either tightly coupled to the JBoss JEE application server or are standalone. To use the framework, the programmer defines AOP constructs as Java classes and binds them to application code using XML or Java 1.5 annotations [54].

Aspect classes are defined as normal Java class that define zero or more advices, pointcuts and/or mixins (a mixin class is a class that is used to implement multiple unrelated interfaces and is often used as an alternative to multiple inheritance [17]).

Figure 3.3 contains an example of an aspect called `MyAspect`[3], which contains an advice, `trace` (line two), that traces calls to any method. The return statement, `invocation.invokeNext()` (line seven) is **required** in order to ensure that either

[3]All examples presented in this section are reproduced from: *JBoss AOP – Aspect-Oriented Framework for Java* [54].

Table 3.2. Pointcuts supported by JBoss AOP.

Pointcut Type	*Description*
execution(method or constructor)	Specifies that an interception occurs whenever a specified method or constructor is called.
get(field expression)	Specifies that an interception occurs when a specified field is accessed to be read.
set(field expression) field	Specifies that an interception occurs when a specified field is accessed to be written to.
field(field expression)	Specifies that an interception occurs when a specified field is accessed to be read from or written to.
all(type expression)	Specifies that calls to a specified constructor, method or field of a particular class will be intercepted.
call(method or constructor)	Specifies that calls to a specified constructor or method will be intercepted.
within(type expression)	Matches any join point (method or constructor call) within any code within a particular call.
withincode(method or constructor)	Matches any join point (method or constructor call) within a particular method or constructor.
has(method or constructor)	Used as an additional requirement for matching. If a join point is matched, its class must also have a constructor or method that matches the `has` expression.
hasfield(field expression)	Used as an additional requirement for matching. If a join point is matched, its class must also have a constructor or method that matches the `hasfield` expression.

the next advice in the chain (if there is more than one) or the actual method or constructor invocation is called. Failure to adhere to this protocol results in the failure to call other advice and/or the method or constructor.

The framework also supports other invocation types such as all invocations, `public Object trace(Invocation invocation)`, and constructor invocations, `public Object trace(ConstructorInvocation invocation)`.

XML files are used by the framework to describe pointcuts and the attachment of pointcuts to aspects. Table 3.2 lists the types of pointcuts supported by the JBoss AOP framework [54].

For example, the XML constructs used to trace all calls to the `withdraw` method on any object that has a parameter of `double` using the `MyAspect` aspect example presented in Figure 3.3 is:

```
<aop>
  <aspect class="MyAspect"/>
  <bind pointcut=
     "execution(* void *->withdraw(double amount))">
    <advice name="trace" aspect="MyAspect"/>
  </bind>
</aop>
```

One of the more interesting features of the JBoss AOP framework is that it allows for the use of *introductions* and *mixins*. An introduction is used to alter an existing class so that it implements one or more additional interfaces. For example the class:

```
public class POJO {
  private String field;
}
```

can be made to implement the `java.io.Externalizable` interface by using the following XML:

```
<introduction class="POJO">
  <mixin>
    <interfaces>
      java.io.Externalizable
    </interfaces>
    <class>ExternalizableMixin</class>
    <construction>
      new ExternalizableMixin(this)
    </construction>
  </mixin>
</introduction>
```

The class element above defines the mixin class that will implement the externalizable interface, while the construction element specifies the Java code that will be used to initialise the mixin class when it is created.

The JBoss AOP framework also has the ability to allow the deployment and undeployment of aspects at runtime. This is achieved by using the `aopc` compiler to 'prepare' join points in the application to accept advices at runtime. Preparing alters the bytecode by inserting dummy placeholders where advice can later be applied [54].

The JBoss AOP framework is an extensive elegant framework that supports many complex AOP constructs as well as many useful features, such as the hot deployment/undeployment of aspects at runtime. As with many object-oriented frameworks it requires the programmer to adhere to a specified protocol, such as ensuring that the `invocation.invokeNext()` method is called in an aspect. If the programmer fails to adhere to this protocol, the application will not behave as expected. Unfortunately these types of issues can only be picked up at runtime, not compile time.

TABLE 3.3. Four aspects of self-management with autonomic computing.

Concept	*Current Computing*	*Autonomic Computing*
Self-configuration	Corporate data centres have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimisation	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmers weeks.	System automatically detects, diagnoses, and repairs localised software and hardware problems.
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures.

3.7 AOP and Autonomics

Autonomic computing is an initiative proposed by the IBM corporation to overcome an impending complexity crisis in the development of applications [52]. According to IBM, this complexity is growing beyond human ability to manage it.

To overcome this, IBM proposes that systems are developed to manage themselves, given high-level objectives by administrators, so that they may adjust their operation, workloads, demands and external conditions in the face of hardware or software failures. IBM cites four aspects of self-management, which are detailed in Table 3.3 [56].

To meet the autonomic computing vision of self-managing, self-healing and self-optimising systems requires a system to be able to dynamically adapt to its environment. However, a key challenge limiting the use of autonomic features in applications is the lack of tools and frameworks that can alleviate the complexities stemming from the use of manual development methods [56].

McKinley et al. [69] define two general approaches to implement adaptive software:

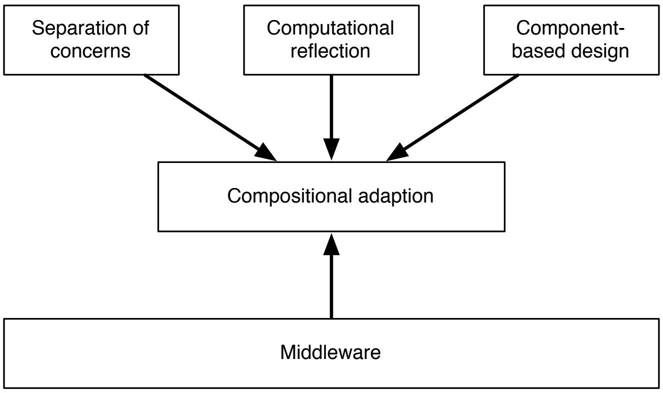


FIGURE 3.4. Main technologies supporting compositional adaption.

Parameter adaption modifies program variables that are used to determine behaviour, for example by adjusting a retry count depending on current network conditions. This type of adaption is severely limited as it does not allow new algorithms and software components to be added to an application after initial design and development.

Compositional adaption allows an application to replace parts of a program's components with another to improve the program's fit into the current operating environment, for example by adding new behaviour to a deployed system. While this is much more flexible than parameter adaption, incorrect use may result in a program that is difficult to test and debug.

McKinley et al. [69] define three main technologies, illustrated in Figure 3.4[4], that can be used to support compositional adaption; *separation of concerns*, *computational reflection*, and *component-based design*.

There are two main techniques used to implement compositional adaption in application code. The first is to use a language, such as CLOS or Python, that directly supports dynamic recomposition, and the second is to weave the adaptive code into the functional code using aspect-oriented techniques [69].

For the purposes of this discussion we concentrate on using the AOP technique to develop adaptive software and present relevant implementations of this approach.

[4]This diagram is reproduced from McKinley et al. [69].

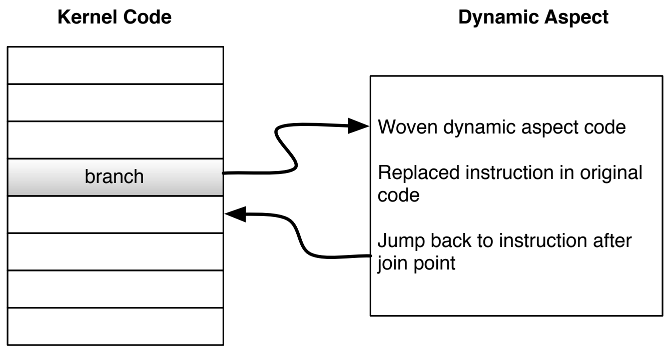

Kernel Code
Dynamic Aspect
branch
Woven dynamic aspect code
Replaced instruction in original code
Jump back to instruction after join point

FIGURE 3.5. TOSKANA Code Splicing.

3.7.1 The TOSKANA Toolkit

Engel and Freisleben [33] have developed the TOSKANA toolkit to dynamically apply aspects to the NetBSD operating system. This toolkit provides a set of tools, macros and libraries for developing and deploying dynamic aspects in the kernel space of NetBSD.

Aspects are developed using standard C macros and are inserted into the kernel as loadable kernel modules. A runtime library, called *aspectlib.ko*, dynamically applies join points using a technique known as *code splicing.* Code splicing replaces the bit patterns of instructions in native code with a branch to a location outside the predefined code flow, where additional instructions followed by the originally replaced instruction and a jump back to the instruction after the splicing location are inserted. As the execution of kernel functions may usually be interrupted at any time, the splicing operation is performed atomically [33]. This process is illustrated in Figure 3.5[5].

TOSKANA supports `before`, `after` and `around` advice variants and these are implemented using the BEFORE, AFTER and AROUND macros in standard C code as illustrated below:

```
#include <sys/aspects.h>
...
void aspect_init(void) { /* deploy three aspects */
   BEFORE(sys_open, open_aspect);
   AFTER(sys_open, close_aspect);
   AROUND(func, some_aspect);
}
ASPECT open_aspect(void) {
   ...
}
```

[5]This diagram is reproduced from Engel and Freisleben [33].

```
ASPECT close_aspect(void) {
   ...
}
ASPECT some_aspect(void) {
   ...
   PROCEED();
   ...
}
```

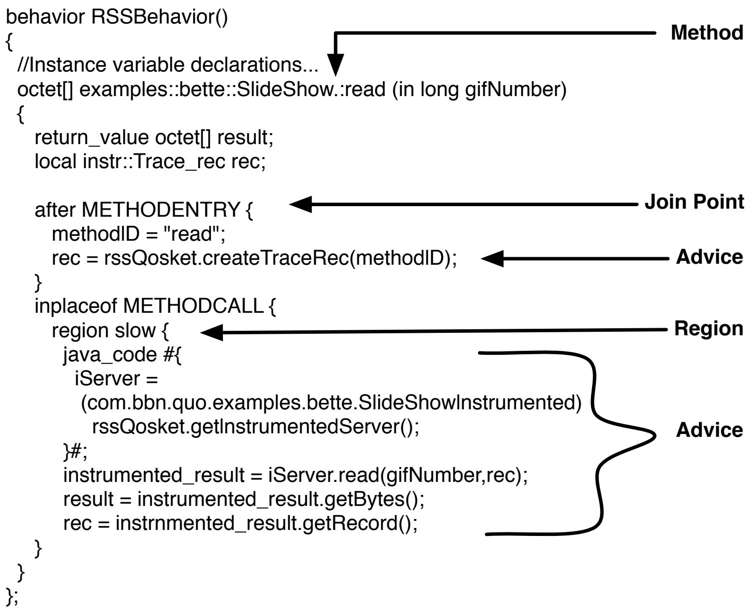
The code is then compiled as a standard NetBSD kernel module. At runtime, the user mode dynamic code weaver *weave* is used to load the requested kernel module and execute its corresponding initialisation function (*aspect_init()*), which calls a support library to do the actual weaving using the code splicing technique. Subsequent functions calls are then routed through the user-provided aspects.

Engel and Freisleben [33] provide a number of use cases of adding adaption to NetBSD using this approach:

Self-Configuration. In common with most operating systems, new devices may be added or removed from the NetBSD operating system dynamically. While adding a new device does not affect running processes, removing one may. For example removing a USB memory key that contains a file system that is currently being accessed by a process. The cross-cutting functionality affected here is the call to the *VOP_OPEN* function, located in 43 areas in the architecture-specific, base kernel, file system and device driver code in the operating system. In this scenario an aspect may intercept the device removal and signal the different parts of the operating system affected that a device is no longer available if the operating system tries to open a file on the now non-existent device.

Self-Healing. The NetBSD system, in common with all operating systems, may run out of memory if a process requests more memory than that which is available in the virtual memory system. Using an aspect, the out-of-memory error condition can be intercepted and the aspect can add additional virtual memory dynamically by adding additional swap files to the system.

Self-Optimization. To calculate the number of free blocks in a file system, the operating system skims through the free block list and counts the number of bits that indicate a free block. However, if the operating system could detect that reading the free block count occurs more frequently than updating the free block bitmap, an optimisation could be achieved by dynamically switching the free block calculation so that the number of free blocks is calculated prior to every bitmap update instead of every call to a readout. This provides self optimisation because the system can dynamically shift the count of free blocks depending on changing conditions and this functionality can be provided in an aspect.



```
behavior RSSBehavior()
{
 //Instance variable declarations...
 octet[] examples::bette::SlideShow.:read (in long gifNumber)
 {
   return_value octet[] result;
   local instr::Trace_rec rec;

   after METHODENTRY {
     methodID = "read";
     rec = rssQosket.createTraceRec(methodID);
   }
   inplaceof METHODCALL {
     region slow {
      java_code #{
        iServer =
         (com.bbn.quo.examples.bette.SlideShowInstrumented)
          rssQosket.getInstrumentedServer();
      }#;
      instrumented_result = iServer.read(gifNumber,rec);
      result = instrumented_result.getBytes();
      rec = instrnmented_result.getRecord();
   }
 }
};
```

FIGURE 3.6. QuO ASL example.

This approach presents the possibility of operating systems being able to be modularised to a far greater extent than that which is currently possible and, as described above, provides the possibly to add autonomic computing functionality.

3.7.2 The QuO Toolkit

Middleware technologies, such as CORBA [77], allow the development of distributed applications without the developer needing to be aware of details of the distribution technology involved. As these applications may be distributed across a number of different physical machines connected via one or more networks, the applications concerned may need to adapt to the changing network and system conditions to maintain an acceptable quality of of service (QoS).

Duzan et al. [30] have developed the QuO toolkit which builds QoS as an aspect and weaves the aspect into the boundary between the application and the middleware. QoU defines an aspect model, which includes join points specific to distribution and adaption, and an adaption model which defines the adaption strategy to be used. The QuO toolkit consists of four main entities [30]:

1. Contracts, which are used to define an adaption policy in QuO. Contracts are defined using QUO's Contract Definition Language (CDL).
2. System Condition Objects, which are used to monitor the environment.

3. Callbacks, which are used for system, middleware and out-of-band application adaption.
4. Delegates, which define the aspect-oriented weaving of the adaptive behaviour into wrappers around application interfaces.

QuO provides an Aspect Specification Language (ASL), which is used to define the monitoring or control behaviour and is compiled to produce a *delegate*, which acts as a proxy for calls to an object reference or a servent (the remote object).

Figure 3.6 illustrates an example of QuO's ASL used to define two advices that are to be applied for the method `examples::bette::SlideShow::read`.

Although delegates may be defined for other middleware environments, the current implementation of the QuO toolkit is designed for the CORBA environment and therefore the ASL advice is applied to a particular method in the CORBA IDL, e. g. `examples::bette::SlideShow::read` in Figure 3.6.

The region statement in the ASL Figure 3.6 refers to a region defined in QuO's Contract Definition Language (not shown), which defines the meaning of the term 'slow'. In this example, once the system reaches that state, the corresponding advice is executed.

3.7.3 Reflection and AOP

A reflective system is a system that incorporates structures about itself. The sum of these structures is called the self-representation of the system, which makes it possible for the system to answer questions about itself and support actions on its behalf [67].

In a language that supports reflection, each object is given a *meta-object*, which holds the reflective information available about the object [67]. Metaobject protocols (MOP's) are interfaces, to the language that give users the ability to incrementally modify the language's behaviour and implementation [58], typically by using a set of classes and methods that allow a program to inspect and alter the state of the application.

Reflection is a common way of developing adaptive software and Grace et al. [43] have proposed combining the use of AOP and reflective middleware to implement dynamic adaptive systems to provide the following benefits:

- The ability to support fine-grained introspection and dynamic adaption of aspects including the ability to adapt or re-order advice behaviour and reconfigure the joinpoint set thereby supporting self-adaption and system wide validation of crosscutting concerns.
- The provision of multiple system viewpoints to support complex adaptions. For example an MOP to manage component adaption, another to manage crosscutting module adaption and another to manage resource usage adaption.

- Increased performance (compared to a purely reflective implementation) by deploying reflection using aspects only where required.

Surajbali et al. [105] argue that the reflective middleware approach is limiting as the reflective APIs have been found to expose a steep learning curve and places too much expressive power in the hands of developers. Instead their approach is to build an AOP support layer on top of an underlying component-based reflective middleware substrate. Surajbali et al. [105] provide an implementation of their model using the OpenCOM component model and the GridKit middleware platform and claim their approach provides the following benefits [105]:

- The complexity of the reflection layer is hidden from programmers.
- The AOP support layer can be dynamically deployed and undeployed when required thereby avoiding overhead when not in use.
- As the AOP support layer is constructed from OpenCOM components like the rest of the system, the underlying middleware system and the AOP support layer can be the target of an advice.
- As distributed dynamic aspects are supported, aspects can be dynamically deployed across a distributed system on the basis of distributed pointcut expressions.

Greenwood and Blair [44] have proposed the use of *dynamic AOP* to implement autonomics. This approach allows adaptions to be encapsulated as aspects, thereby allowing adaptions to be contained and applied retrospectively at runtime.

Greenwood and Blair's implementation [45] uses the AspectWerkz [16] dynamic AOP framework combined with reflection and a policy framework to define adaptive behaviour based on Event-Condition-Action rules.

3.8 AOP and the Distribution Concern

Several attempts have been made to apply distribution aspects to existing Java code. These attempts typically target a single distribution protocol, RMI, and either generate code in the general purpose aspect language, AspectJ [18, 91, 114], use a domain-specific language [66, 73], extend Java [83], or extend the AspectJ language to provide distribution [74].

While RMI is the most widely used distribution protocol in Java systems and is used as the protocol for Enterprise JavaBeans (EJB), Jini and JavaSpaces, there are a number of other distributed systems, such as CORBA, JMS, SOAP, HTTP, Java sockets etc. that Java programmers may choose to use and indeed may have to use to solve a particular integration problem.

Programmers therefore have a large choice of protocols, each with its own framework and possibly different programming convention. This significantly complicates distributed systems development.

3.8.1 Domain-Specific Aspect Language Implementations

We define a domain-specific aspect language as a language designed for a specific domain e. g. distribution, that is used by an aspect weaver to insert code based on instructions and possibly code contained in the language into existing code either statically, at compile time, or dynamically at runtime.

A number of domain-specific aspect languages have been proposed including KALA for the transactional domain [35], ERTSAL for the real-time domain [94] and ALPH for the healthcare domain [72].

Many domain-specific languages (e. g. Orca [8]) have been proposed to aid distributed programming, and seminal work on aspect-orientation proposes a number of domain-specific aspect languages, such as the D language framework consisting of the domain-specific aspect languages COOL, for concurrency management, and RIDL for distribution [66], and the RG language [70], amongst others.

Since the RIDL language was conceived, little work has been done on domain-specific aspect languages for the distribution concern and the closest work to ours thus far are RIDL and AWED, which we describe in this section.

3.8.1.1 The D Language Framework

The D language framework consists of three sub-languages:

- JCore, an object-oriented language used to express the basic functionality and the activity of the system. JCore is a subset of Java 1.0.
- COOL, an aspect language used to express the co-ordination of threads.
- RIDL, an aspect language used to express distribution and remote access strategies.

A tool that implements an aspect weaver takes the programs written in the different sub-languages and combines them to produce an executable program with the specified distributed behaviour. The D framework consists of three types of modules [65]:

- Classes – Used to implement functional components.
- Coordinators – Used to implement the thread coordination aspect.
- Portals – Used to define code for dealing with application-level data transfers over remote method invocations.

For the purposes of this discussion, we concentrate on RIDL, the aspect language used to express remote access strategies.

RIDL is used to define remote RMI objects, the parameter passing mode for each distributed method in those objects, and the parts of the object graph that should be copied if the call uses the copy-by-value semantics.

In order for an object to become a remote object, RIDL requires that a remote interface and portal be defined (for that object) that stipulates the subset of

```
public class BoundedBufferV1 {
  private Book array[];
  private int takePtr = 0, putPtr = 0;
  protected int usedSlots = 0, size;
  BoundedBufferV1(int capacity) throws IllegalArgumentException {
    if (capacity <= 0) throw new IllegalArgumentException();
    array = new Object[capacity];
    size = capacity ;
  }
  public int count() { return usedSlots; }
  public int capacity() { return size; }
  public void put(Book x) throws Full {
    if (usedSlots == array.length) throw new Full();
    array[putPtr] = x;
    putPtr = (putPtr + 1) % size;
    usedSlots++;
    System.out.println("BB got book:");
    b.print();
  }
  public Book take() throws Empty {
    if (usedSlots == 0) throw new Empty();
    Book old = array[takePtr];
    takePtr = (takePtr + 1) % size;
    usedSlots--;
    return old;
  }
}
public class Book {
  private int isbn = 0;
  private String title = null;
  private Postscript ps;
  Book(int n, String t) {isbn = n; title = t;}
  public void print() {
     System.out.println ("Book: " + isbn + title);
   }
}
```

FIGURE 3.7. BoundedBuffer example (reproduced from Lopes [65]).

methods of the class that can be invoked remotely, and the parameters and return values of those methods. For each parameter and return value the programmer may optionally define the mode that describes how the data transfers are to be made, copy-by-value or copy-by-reference.

For example, given the class[6] in Figure 3.7 the portal:

```
portal BoundedBufferV1 {
  int capacity();
  void put(Book x);
  Book take();
}
```

states that:

- The methods `capacity`, `put` and `take` are remote methods.
- The method `count`, as it has not been defined in the portal, is a local method.
- The `Book` argument to the `put` method and the `Book` return value from the `take` method are passed and returned respectively using the default transfer strategy, copy-by-value.

If, instead, the programmer wishes to use the copy-by-reference semantics, the following portal can be defined using the `gref` (for global reference) keyword:

```
portal BoundedBufferV1 {
  int capacity();
  void put(Book x) {
    x: gref;
  }
  Book take() {
    return: gref;
  }
}
```

However by doing so, the `Book` class must now be defined as a remote object as well:

```
portal Book {
  void print();
}
```

The application that instantiates the bounded buffer class, defined in Figure 3.7, must export a reference to that instance in the name server as illustrated below:

```
public class StartBuffer {
  public static void main(String args[]) {
    BoundedBufferV1 bb = new BoundedBufferV1(100);
    try {
      DJNaming.bind("rmi://goblin/BB", bb);
    } catch (Exception e) {
      System.out.println("StartBuffer err: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

[6]The examples in this section are reproduced from Lopes [65].

RIDL's DJNaming class is a wrapper class that interacts with Java's RMI Naming class and is therefore tied to a single protocol, RMI. Objects that export these remote object references are explicitly tied to the RIDL specific naming framework, thereby tying them to the framework.

Client calls to remote methods are exactly the same as calls to local methods with the following exceptions:

- The run-time exception `DJInvalidRemoteOp` may be thrown.
- RIDL framework elements must be used to locate the remote method.

For example, to bind to the remote object, `BoundedBuffer`, the RIDL specific naming framework must be used:

```
BoundedBuffer1 bb = new BoundedBuffer(100);
String url = "rmi://parc.xerox.com/BoundedBuffer";
// bind url to remote object
DJNaming.bind(url, bb);
        ...
// lookup bounded buffer
bb = (BoundedBuffer)DJNaming.lookup(url);
```

One of the compelling features of RIDL is the ability to pass or return partial copies of objects in the remote call. For example, the portal:

```
portal BoundedBufferV1 {
  int capacity();
  void put(Book x) {
    x: gref;
  }
  Book take() {
    return: copy { Book bypass title, ps; }
  }
}
```

declares that the returned `Book` object does not contain the `title` or `ps` fields. This feature can dramatically reduce the overhead of a remote call. If the programmer inadvertently refers to the `title` or `ps` fields, an error is generated.

RIDL programmers are required to adhere to RIDL's naming framework and the use of the RIDL specific exception, `DJInvalidRemoteOp`. RIDL is therefore not entirely transparent to the programmer and by being so is intrusive in nature although this intrusiveness is fairly limited.

3.8.1.2 AWED

AWED [73] is a comprehensive aspect language for distribution which provides remote pointcuts, distributed advice, and distributed aspects and is implemented by extensions to the JAsCo [106] AOP framework called DJAsCo.

The main characteristics of the AWED model are:

Remote pointcuts which can be used to match events on remote hosts, including remote sequences. Sequences define a list of methods that have been executed in order and may be referred to in pointcuts and advice. Remote pointcuts enable the matching of join points on remote hosts and includes remote calls and remote `cflow` constructs (matching of nested calls over different machines).

Distributed advice execution. Advice can be executed either synchronously or asynchronously.

Distributed aspects which may be configured using different deployment and instantiation options.

The motivation behind the development of the AWED language is transactional cache replication in the JBoss application server and consequently the language supports the notion of distributed hosts with the keyword `host` and includes the ability to define groups of hosts.

On the occurrence of a join point, AWED evaluates all pointcuts on all hosts where the corresponding aspects are deployed. Pointcuts may contain conditions about hosts where the join point originated and may also be defined in terms where advice is executed [73].

For example[7] the pointcut:

```
call(void initCache()) && host("adr1:port")
```

matches calls to the initCache method on the host with the specified address and the advice may be executed on any host where the aspect is deployed [73].

The AWED language is a fairly low-level language and borrows much of its syntax from AspectJ, including the keywords *pointcut*, *call*, *after*, *around* and *before*. In common with RIDL, the AWED language has no support for either multiple protocols (its current implementation uses the RMI protocol exclusively) or the recovery concern.

3.8.2 AspectJ Implementations

Soares et al. [91] illustrate how the AspectJ language can be used to introduce the distribution concern in the form of the RMI protocol into existing non-distributed applications. Soares et al.'s solution utilises two aspects, a client aspect and a server aspect. For the server aspect, a remote interface is generated for each object that is to be distributed and each object is altered to implement the interface. The client aspect redirects local method calls to the now remote object and alters the local methods to declare that they throw the `RemoteException` exception. Due to AspectJ's inability to allow changes to the target object in its *proceed* statement, a dedicated redirection advice is used to redirect calls to the remote object from the client object.

[7]These examples are reproduced from Navarro et al. [73].

Ceccato and Tonella [18] use a static code analyser and code generator to analyse a non-distributed application and generate AspectJ code to apply the distribution concern using the RMI protocol. All public methods in a class are automatically altered to be remote methods and the various RMI specific conventions are applied. In addition, parameters and return values are declared to be remote objects so that they may be passed by reference, instead of the RMI default pass-by-value, to avoid issues that may occur if the object cannot be serialized. However, as identified by Tilevich and Smaragdakis [111], this approach is extremely inefficient as each method call generates network traffic.

3.8.3 J-Orchestra

J-Orchestra [110, 113] is an automatic partitioning system for Java, which uses bytecode rewriting to apply distribution and claims to be able to partition any Java application and allow any application object to be placed on any machine, regardless of how the application objects interact.

Although J-Orchestra's focus is on the automatic partitioning of Java applications and does not employ a domain-specific aspect language or specifically define aspect-oriented concepts, such as join points or pointcuts, its use of bytecode rewriting is essentially an aspect-oriented approach.

J-Orchestra replaces Java's RMI with NRMI [111], a modified version that implements *call-by-copy-restore* semantics for object types for remote calls. In RMI, copy-by-value is used to pass parameters from the client to the server. That is, parameter objects are serialized and copied to the server. However, any changes to parameter objects on the server are lost when the call returns. Call-by-copy-restore overcomes this by copying changes made on the server back to the client which is, to the user, more natural as it emulates a local procedure call. However, inevitably, additional network traffic is generated.

J-Orchestra implements distributed thread management [112] so that multi-threaded applications can behave in the same manner once they are automatically partitioned. Distributed thread management is implemented by altering only the thread specific bytecode with calls to operations of the J-Orchestra distribution-aware synchronisation library. Again, this has an overhead on network traffic.

Users interact with the J-Orchestra system using XML files, which simply detail a list of classes to be distributed. J-Orchestra has no concept of a domain-specific language, multiple protocols or user-defined definition and manipulation of the recovery concern. Nevertheless it does, for the RMI protocol, provide sophisticated automatic partitioning.

3.8.4 Other Systems

Both JavaParty [83] and DJcutter [74] use a language-based approach and supply Java language extensions to provide explicit support for distribution. Again these systems target the RMI protocol exclusively.

JAC [82] is a dynamic AOP framework that has been extended to support a distributed pointcut definition, which extends the regular pointcut definition with the ability to specify a named host where the join point should be detected. To support the distributed deployment of aspects, JAC replicates its Aspect-Component manager, which is used to keep track of registered aspects on the named hosts. A consistency protocol is used to ensure that the weaving/unweaving of an aspect on one site triggers the weaving or unweaving of the same aspect on other sites [82].

A number of multi-protocol systems, such as RMIX [62] and ACT [29], have been proposed. However, these systems use the high-level framework approach discussed in Section 2.7 and therefore have the same issue as all framework-based approaches, namely tying the application code to the framework or API.

3.8.5 Our Approach

The D language framework concentrated on distributed thread co-ordination and remote access strategies but did not address error handling mechanisms. Indeed Lopes [65], the author of the D language framework, states that error handling was omitted from D, not because it was not a problem, but because it was *too big of a design problem that needed much more research.* Our research addresses this issue by introducing a domain-specific aspect language that provides modularisation not only for the distribution concern but also for the *distribution recovery concern.* In addition we provide support for multiple protocols while other approaches only support a single protocol.

Our approach introduces the concept of a Distribution Definition Language (DDL), a simple high-level domain-specific aspect language, which generalises distributed systems development by describing the classes and methods to be made remote, the protocol to use to make them remote and the method used to recover from a remote error. The DDL is used by the RemoteJ compiler/generator, which uses bytecode manipulation and generation techniques to provide a distributed version of the application while retaining existing classes for reuse in other distributed or non-distributed applications.

By generalising and modularising the distribution and recovery concerns, the use of a DDL provides a method of developing distributed applications that is significantly simplified, allows multiple protocols to be supported for the same code base, allows explicit definition of the recovery concern and enables the same code to be used in both a distributed and non-distributed application thereby improving software reuse.

3.9 Chapter Summary

This chapter has provided the lineage towards the domain-specific aspect language (DSAL) approach to distributed systems development by examining aspect-orientation, aspect-orientation as applied to autonomic systems, aspect-oriented languages and aspect-oriented frameworks and their features and facilities.

Related work in using aspects for distribution, including frameworks, domain-specific languages, Java language extensions and AspectJ approaches, have been discussed.

We have discussed three common implementations of aspect-oriented systems:

1. The language-based approach, which is designed as an extension to an existing language.
2. The object-oriented framework-based approach, which provides a framework that is used by developers to apply aspects to existing code.
3. The domain-specific aspect language approach, which uses a domain-specific language to apply aspects to existing code.

AspectJ [6], an example of the first approach, extends the Java programming language and provides a low level generalised approach to aspect-oriented programming. Because AspectJ, and other language-based systems, are at a low level they are relatively complicated to use. In addition, they introduce additional concepts and constructs, such as the notion of introductions, join points and pointcuts, which further complicates their understanding.

The JBoss AOP [54] framework is an example of the framework-based approach as it allows programmers to define AOP constructs as Java classes using an object-oriented framework. These constructs are used to alter the bytecode of the target application using information contained in an XML file. Object-oriented framework approaches are relatively easy to use compared to the language-based approach as they use the same language as the application. However, this approach requires developers to adhere to the framework's protocol, such as ensuring that `invocation.invokeNext()` is called in an aspect, as is the case with the JBoss AOP framework. Therefore, although framework-based approaches are easier to use than language-based approaches, they require the programmer to have a good understanding of the framework.

KALA [35], a domain-specific aspect language for the transactional domain, is an example of the third approach as it uses a high-level domain-specific aspect language to apply aspects to existing code. Domain-specific aspect languages require the developer to use a different language alongside the application language and therefore require the programmer to learn a new language, although the language is generally relatively simple. Nevertheless domain-specific aspect languages, because they are at a higher level of abstraction, are generally much simpler to use than the other two approaches.

A number of systems and proposals that use AOP to provide autonomics have been discussed. However, none of these approaches are specifically targeted at the distribution concern and therefore do not provide a means of implementing recovery, nor do they generalise the distribution concern. Rather they are either targeted towards a specific domain, such as QoS or NetBSD, or are layered on top of an existing middleware system, which hides the distribution and recovery concerns.

We have introduced our approach consisting of a high-level domain-specific aspect language for the distribution and recovery concerns we call a Distribution Definition Language and the RemoteJ compiler/generator, which is used to apply the distribution and recovery concerns described in the Distribution Definition Language to existing applications. The Distribution Definition Language generalises distributed systems development by describing the classes and methods to be made remote, the protocol to use to make them remote and the method used to recover from a remote error.

The closest work to ours thus far are RIDL and AWED. Both RIDL and AWED use a lower level approach than RemoteJ, which, by introducing the concept of a Distribution Definition Language, is at a higher level of abstraction. In addition, the Distribution Definition Language supports error handling, which is not supported by either RIDL or AWED or indeed any other system to our knowledge.