Shakeins: Non-Intrusive Aspects for Middleware Frameworks[⋆]

Tal Cohen[⋆⋆] and Joseph (Yossi) Gil[⋆ ⋆ ⋆]

{ctal, yogi}@cs.technion.ac.il
Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel

Shaken, not stirred.

— Bond, James Bond (Goldfinger, 1964).

Abstract. *Shakeins* are a novel programming construct which, like mixins and generic classes, generates new classes from existing ones in a universal, uniform, and automatic manner: From a given class, a shakein generates a new class which has the same type as the original, but with different data and code implementation. We argue that shakeins are restricted, yet less chaotic, aspects. We further claim that shakeins are well suited for the introduction of *aspect-oriented programming* (AOP) into existing middleware applications. We introduce the ASPECTJ2EE language which, with the help of shakeins and a new *deploy-time* weaving mechanism, brings the blessings of AOP to the J2EE framework. A unique advantage of ASPECTJ2EE, which is less general (and hence less complicated) than ASPECTJ, is that it can be smoothly integrated into J2EE implementations without breaking their architecture.

1 Introduction

1.1 Enterprise Applications

Enterprise applications are large-scale software programs used to operate and manage large organizations. These make the world's largest, and arguably most important, software systems, including the programs that run government organizations, banks, insurance companies, financial institutes, hospitals, etc. Enterprise applications make the world go around!

It is often the case that enterprise applications run on heterogeneous platforms connecting various independent systems (called *tiers*) into a coherent whole. The various tiers of an enterprise application can include, e.g., legacy mainframe servers, dedicated database servers, personal computers, departmental servers, and more.

[⋆] Part of the contents of this paper was presented in the European Conference on Object Oriented Programming [1] and in a recently submitted paper [2].

[⋆⋆] Contact author

[⋆ ⋆ ⋆] Research supported in part by the IBM faculty award

The core functionality served by enterprise applications is often quite simple. It does not involve overly elaborate computation or pose complex algorithmic demands. However, developing enterprise applications is considered a daunting task, due to orthogonal requirements presented by most of these applications: uncompromising reliability demands, unyielding security requirements, and complete trustworthiness that the applications must exhibit.

The staggering demand for rapid development of enterprise applications initiated a series of component-based *middleware architectures*, such as CORBAand DCOM. A prime example of these, and emphasizing client/server and multi-tier structures, is *Java 2, Enterprise Edition* (J2EE)[1] which uses Enterprise JavaBeans (EJB)[2] as its elaborate component model.

The immense complexity of middleware frameworks and enterprise applications makes these an appealing target for mass application of aspect-oriented programming (AOP) [3]. AOP is a methodology which can be used for encapsulating the code relevant to any distinct non-functional concern in *aspect* modules. (We assume basic familiarity with standard AOP terminology, including *join point*—a well-defined point in the program's execution, *pointcut*—a specification of a set of join points, *advice*—code that is added at specified join points, *weaving*—the process of applying advices to join points, and *aspect*—a language construct containing advices.)

AOP can also be thought of as answering the same demand [4, 5]. Indeed, the functionality of J2EE application servers can be *conceptually* decomposed into distinct aspects such as persistence, transaction management, security, and load balancing. The effectiveness of this decomposition is evident from Kim and Clarke's case study [6], which indicates that the EJB framework drastically reduces the need for generic AOP language extensions and tools.

Yet, as we shall see below, the EJB support for functional decomposition is limited and inflexible. In cases where the canned EJB solution is insufficient, applications resort again to a tangled and highly scattered implementation of cross-cutting concerns. Part of the reason is that current J2EE servers do not employ AOP in their implementation, and do not enable developers to decompose new non-functional concerns that show up during the development process.

1.2 J2EE Services as Managers of Non-Functional Concerns

A natural quest is for a harmonious integration of middleware architectures and AOP. Indeed, there were several works on an AOP-based implementation of J2EE servers and services (see e.g., the work of Choi [7]).

Ideally, with the J2EE framework (and to a lesser extent in other such frameworks), the developer only has to implement the domain-specific *business logic*. This "business logic" is none other than what the AOP community calls *functional concerns*. The framework takes charge of issues such as security, persistence, transaction management, and load balancing which are handled by *services* provided by the *EJB container*. Again, these issues are none other than *non-functional concerns* in AOP jargon.

[1] http://java.sun.com/j2ee
[2] http://java.sun.com/products/ejb/

Suppose for example that the programmer needs data objects whose state is mirrored in persistent storage. This storage must then be constantly updated as the object is changed during its lifetime, and vice versa. Automatic updates can be carried out by the *Container-Managed Persistence* (CMP) service of the EJB container. To make this happen, the objects should be defined as *entity beans*. Entity bean types are mapped to tables in a relational database with an appropriate XML configuration file. This *deployment descriptor* file also maps each bean attribute (persistent instance variable) to a field of the corresponding table.

Another standard J2EE service is security, using an approach known as *role-based security*. Consider, for example, a financial software system with two types of users: clients and tellers. A client can perform operations on his own account; a teller can perform operations on any account, and create new accounts. By setting the relevant values in the program's deployment descriptor, we can limit the account-creation method so that only users that were authenticated as tellers will be able to invoke it.

Other services provided by the EJB container handle issues such as transaction management and load balancing. The developer specifies which services are applied to which EJB. Deployment descriptors are used for setup and customization of these services. Thus, J2EE reduces the implementation of many non-functional concerns into mere configuration decisions; in many ways, they turn into *non-concerns*.

The prime example in this work are EJBs. But, the J2EE design guideline, according to which the developer configures various services via deployment descriptors, is not limited to EJBs; it is also used in other parts of the J2EE platform. For example, servlets (server-side programs for web servers) also receive services such as security from their container, and access to specific servlets can be limited using role-based security. This is also true for Java Server Pages (JSPs), another key part of the J2EE architecture. Hence, in our financial software example, certain privileged web pages can be configured so that they will be only accessible to tellers and not to clients.

The various issues handled by EJB container services were always a favored target for being implemented as aspects in AOP-based systems [8, pp. 13–14]. For example, Soares *et al.* [4] implement distribution, persistence and transaction aspects for software components using ASPECTJ [9]. Security was implemented as an aspect by Hao *et al.* [5]. The use of aspects reduces the risk of scattered or tangled code when any of these non-functional concerns is added to a software project.

Conversely, we find that J2EE developers, having the benefit of container services, do not require as much AOP! Kim and Clarke's case study [6] comprised of an e-voting system which included five non-functional concerns: (1) persistent storage of votes, (2) transactional vote updates, (3) secure database access, (4) user authentication, and (5) secure communications using a public key infrastructure [6, Table 1]. Of these five non-functional concerns, *not one* remained cross-cutting or introduced tangled code. The first three were handled by standard J2EE services, configured by setting the proper values in the deployment descriptors. The last two were properly modularized into a small number of classes (two classes in each case) with no code replication and no tangled code.

This is hardly surprising. The J2EE platform designers have provided what, in their experience, an enterprise application developer *needs* for his non-functional concerns.

This was done without relying on aspect-oriented research concepts; it was done in order to scratch an itch. J2EE provides a robust solution to real-world problems encountered by real-world programmers when developing real-world multi-tier enterprise applications.

1.3 Shakeins and Middleware

In summary, despite the complexity of middleware systems, and despite the similarity between AOP and the services that frameworks such as J2EE provide, it is expected that integrating this modern programming technique into the frameworks that need it, will face reluctance. Some J2EE application server vendors, most noticeably JBoss,have recently integrated support for aspects into their products. However, the forthcoming version of the EJB standard (EJB 3.0[3]) chose not to adopt a complete AOP solution, but rather a rudimentary mechanism that can only be used to apply advice of a limited kind to a limited set of methods in an inflexible manner. The basic services are not implemented using such advise, but rather remain on a different plane, unreachable and non-modifiable by the developer.

Plain aspects should make it easier to *add* services to J2EE, if re-engineered to use AOP. They should also make it simpler to *replace* existing services with alternative implementations. This paper makes the case that much more can be offered to the client community by *shakeins*, a novel programming construct, combining features of aspects with selected properties of generics and mixins [10]. Shakeins should make it possible not only to *add* services, but also address two issues that current J2EE users face: *configuration* of existing services and *greater flexibility* in their application. Shakeins should also enjoy a smoother entrance into the domain because they *reuse* the existing J2EE architecture, and because they simplify some of the daunting tasks (e.g., lifecycle methods, see Sec. 6) incurred in the process of developing EJBs.

In a nutshell, a shakein receives a class parameter and optional configuration parameters, and generates a new class which has the same type as the original, but with different data and code implementation. In a sense, shakeins are restricted, yet less chaotic, aspects. Like aspects, they can be used to modularize non-functional concerns without tangled and scattered code. Unlike traditional aspects, shakeins preserve the object model, present better management of aspect scope, and exhibit a more understandable and maintainable semantic model.

Shakeins are more restricted than aspects in that there are limits to the change they can apply to code. However, unlike aspects, shakeins take configuration parameters, which make it possible to tailor the change to the modified class.

We argue that the explicit, intentional, configurable and parameterized application of shakeins to classes makes them very suitable to middleware frameworks and enterprise applications, based of the following reasons:

– The architecture of *existing* middleware frameworks is such that in generating enterprise application from them, the system architect may selectively use any of the

[3] http://www.jcp.org/en/jsr/detail?id=220

services that the framework provides. We have that *a chief functionality of middleware frameworks is in the precise management of non-functional concerns*, and that this objective is well served by the shakeins construct,
– The services that a middleware framework offers are applicable only to certain and very distinguishable components of the application, e.g., EJBs in J2EE. The need to isolate the application of aspects is aggravated by the fact enterprise applications tend to use other software libraries, which should not be subjected to the framework modifications. For example, a business mathematics library that an application uses for amortizing loan payments, should not be affected by the underlying framework security management. Shakeins are suited to this domain because of its fundamental property by which *cross cutting concerns should not be allowed to cut through an entire system*.
– Further, for modularity's sake, it is important that these parts of the enterprise application, which are external to the middleware, are not allowed to interact with the services offered by the middleware. The fact shakeins are forbidden from creating new types helps in *isolation of the cross cutting concerns.*
– To an extent, shakeins generalize existing J2EE technology, presenting it in a more systematic, programming language-theoretical fashion. While doing so, shakeins make it possible to enhance J2EE services in directions that plain aspects fail to penetrate: The configurability of shakeins makes it possible to generalize existing J2EE services. Explicit order of application of shakeins to components adds another dimension of expressive power to the framework. And by using *factories* (a new language construct, detailed in the Appendix), such enhancements can be applied with a minimal disturbance of existing code.

1.4 AspectJ2EE

Shakeins draw from the lessons of J2EE and its implementation. To demonstrate their applicability to this domain, we introduce the AspectJ2EE [1] language, which shows how shakeins can be used to bring the blessings of AOP to the J2EE framework. AspectJ2EE is geared towards the generalized implementation of J2EE application servers and applications within this framework.

As the name suggests, AspectJ2EE borrows much of the syntax of AspectJ. The semantics of AspectJ2EE is adopted from shakeins, while adapting these to AspectJ. The main syntactical differences are due to the fact that "aspects" in AspectJ2EE can be parameterized, as will be revealed by the forthcoming exposition. In the initial implementation we have of AspectJ2EE, parameter passing and the application of shakeins are not strictly part of the language. They are governed mostly by external XML configuration files, a-la J2EE deployment descriptors.

A distinguishing advantage of this new language is that it can be smoothly integrated into J2EE implementations without breaking their architecture. This is achieved by generalizing the existing process of binding services to user applications in the J2EE application server into a novel *deploy-time* mechanism of weaving aspects. Deploy-time weaving is superior to traditional weaving mechanisms in that it preserves the object model, has a better management of aspect scope, and presents a more understandable and maintainable semantic model. Also, deploy time weaving stays away from specialized JVMs and bytecode manipulation for aspect-weaving.

Standing on the shoulders of the J2EE experience, we can argue that shakeins in general, and ASPECTJ2EE in particular, are suited to systematic development of enterprise applications. Unlike existing attempts to add AOP functionality to J2EE application servers, the ASPECTJ2EE approach is methodical. Rather than add aspects as an additional layer, unrelated to the existing services, our approach is that even the standard services should be implemented on top of the AOP groundwork. Using ASPECTJ2EE, the fixed set of standard J2EE services is replaced by a library of core aspects. These services can be augmented with new ones, such as logging and performance monitoring.

It should be noted that ASPECTJ2EE has its limitations compared to more traditional implementations of aspects; in particular, it is not at all suited to low-level debugging or nit-picking logging. (For example, access to *non-private* fields by classes other than the defining class is not a valid join point.) However, it is not for these tasks that ASPECTJ2EE was designed, and it is highly suited for dealing with large systems and global aspect oriented programming. (The field access restriction example does not mature into a major hurdle in the kind systems we are interested in; field management can be always be decomposed into getter and setter methods, and in fact *must* be decomposed this way in J2EE applications, where fields are realized as *attributes* with proper join points at their retrieval and setting.)

Moreover, the ASPECTJ2EE language has specific support for the composition of aspects that are scattered across program tiers (*tier-cutting concerns*), such as encryption, data compression, and memoization. We stress that unlike previous implementations of aspects within the standard object model, ASPECTJ2EE does not merely support "before" and "after" advices and "method execution" join points. ASPECTJ2EE supports "around" advices, and a rich set of join points, including control-flow based, conditional, exception handling, and object- and class-initialization.

Historically, the developers of enterprise applications are slow to adopt new technologies; a technology has to prove itself again and again, over a long period of time, before the maintainers of such large-scale applications will even consider adopting it for their needs. It is not a coincidence that many large organizations still use and maintain software developed using some technologies, such as COBOL , that other sectors of the software industry view as thoroughly outdated. The huge investment in legacy code slows the adoption of new technology.

We believe that the fact that shakeins preserve the standard object model, and rely on existing J2EE technologies, should contribute to widespread adoption of this new technology in the middleware software domain.

Outline. Sec. 2 explains in greater detail the shakeins programming construct while comparing it to other such constructs. Sec. 3 makes the case for representing J2EE services as aspects in general, and as shakeins in particular. In Sec. 4 we compare the shakeins mechanism with two important products introducing aspects into J2EE. Deploy time weaving is discussed in Sec. 5, which also explains why this mechanism is suited to shakeins. We then show in Sec. 6 how shakeins can be married with middleware frameworks, by means the ASPECTJ2EE language. Sec. 7 lists several possible innovative uses for ASPECTJ2EE, some of which can lead to substantial performance benefits. Sec. 8 concludes.

2 The Case for Shakeins

As explained above, a shakein S takes an existing class c as input, along with optional configuration parameters, and generates from it a new class $S\langle c\rangle$, such that the *type* of $S\langle c\rangle$ is the *same* as that of c. The *realization* of this type in $S\langle c\rangle$ may, and usually will, be different than in c. Further, the process of generating $S\langle c\rangle$ from c is *automatic*, *universal* and *uniform*.

This section makes the case for shakeins, and explains this process in greater detail: Sec. 2.1 explains what we mean by making the distinction between the type defined by a class and its realization by the class definition. With this distinction, we explain in Sec. 2.2 why the mechanisms of aspects[4] and inheritance serve similar purposes but from different, foreign and hard-to-reconcile perspectives. Sec. 2.3 presents the shakeins mechanism to resolve this schism.

Sec. 2.4 explains that shakeins, just like mixins, have an explicit parameter-based application semantics, and discusses some of the benefits of this semantics. The theoretical discussion of this new language mechanism is concluded in Sec. 2.5 which shows how they simplify some of the subtle points of aspects.

2.1 The Five Facets of a Class

As early as 1971, Parnas [11] made the classical distinction between the *interface* and the *materialization* perspectives of a software component. It is only natural to apply this distinction to class definitions in JAVA and other mainstream object oriented languages. In this section we do so, while taking into notice that, unlike the software components of the seventies, classes are instantiable.

Accordingly, the interface of a class presents two facets: the *forge* of the class, being the set of non-**`private`** constructor signatures; and its *type*, the set of externally-accessible fields and methods as well as available cast operations. The materialization of a class has three facets: the *implementation*, i.e., the method bodies; the *mill*, being the implementation of the forge; and the *mold*, which is the memory layout used for creating objects.

Fig. 2.1 shows a sample class[5] (a) alongside its forge (b), type (c) and mold (d).

2.2 The Aspects-Inheritance Schism

A key feature of OOP is the ability to define new classes based on existing ones. The five facets described above can be used for explaining inheritance in JAVA and similar

[4] We refer here to ASPECTJ-style aspects. Most other aspect-oriented languages are variants of this same style.

[5] A similarly structured class served as the running example in Kiczales and Mezini's [12] work on modularity and aspects.

Fig. 2.1 The `Point` class definition and the ensuing forge, type and mold.

```
class Point implements Shape {
   int x, y;
   public Point(int x, int y){
      this.x = x;
      this.y = y;
   }
   public Point(){
      this(0,0);
   }
   public Point(Point other){
      this.x = other.x;
      this.y = other.y;
   }
   public int getX() {
      return x;
   }
   public int getY() {
      return y;
   }
   public void setX(int x) {
      this.x = x;
   }
   public void setY(int y) {
      this.y = y;
   }
   public void moveBy(int dx, int dy) {
      x += dx; y += dy;
   }
}
```

(a) The class definition.

```
public Point(int x, int y);
public Point();
public Point(Point other);
```

(b) The forge.

```
int x, y;
public int getX(); public int getY();
public void setX(int);
public void setY(int);
// From Shape:
public void moveBy(int, int);
// From java.lang.Object:
public boolean equals(Object);
public int hashCode();
// ... etc.
// Upcast operations:
public (Shape)(); public (Object)();
```

(c) The resulting type.

int x	32 bits
int y	32 bits
Fields inherited from `Object`	
Hidden fields added by the JVM	

(d) The class's mold.

languages: Inheritance *extends* the type, mold and implementation of a class. By means of overriding, inheritance also makes it possible to *replace* or *refine* parts or all of the implementation. The mold, however, cannot be refined.

Interestingly, inheritance in mainstream programming languages *recreates* the forge from scratch, since constructor signatures are not inherited. Therefore, inheritance is allowed to make arbitrary and incompatible changes to the forge facet of a class. The mill however can only be refined, since constructors must invoke inherited versions.

Examining the notion of aspects from the perspective of modifications to the five facets, we see that they offer a similar repertoire of modifications. Aspects are advertised as means for breaking the implementation into its orthogonal concerns; accordingly, an aspect may replace or refine the implementation. In many languages, aspects can also *extend* the type and the mold by introducing new members into a class. (This is known as *member introduction* in ASPECTJ. Conceptually, fields in an aspect that has per-object instantiation may also be viewed as fields added to the base object's mold.)

Still, unlike inheritance, the application of an aspect does not define a new type, nor does it introduce a new mold or a new implementation; it changes the mold and the implementation of existing classes *in situ*.[6] And, while member introduction makes it possible to modify the type of classes, it does not introduce a *new* type—because classes touched by aspects cease to exist in their original form; only the modified form exists. In particular, there cannot be instances of the original class in memory.

The similarity of aspects and inheritance raises intriguing questions pertaining to the interaction between the two mechanisms: Does the aspectualized class inherit from its

[6] Also, aspects can only *extend* the forge (using member introduction) but not replace it.

base version? May aspects delete fields from the mold, or modify it by changing the type of fields? Likewise, can methods be removed from the type? Can their signature be changed? How does the application of an aspect to a class affect its subclasses?

The interaction of join points with inheritance raises still more questions.[7] Suppose that class `B` inherits from `A`, with, or without overriding method `m()`. Then, does the pointcut **`call`**`(A.m())` apply to an invocation of method `m()` on an object whose dynamic type is `B`? Conversely, does the pointcut **`call`**`(B.m())` ever apply if class `B` does not define a method `m()`, but inherits it from `A`?

The reason that all these questions pop up is that aspects were not designed with inheritance in mind. The original description of aspects [3] did not dedicate this new construct to OOP languages. Similarly, the founding fathers of OOP did not foresee the advent of AOP, and in many occasions inheritance was used for some of the purposes AOP tries to serve. We witness what may be called *The Aspects-Inheritance Schism*.

Gradecki and Lesiecki [13, p. 220] speculate that *"mainstream aspect-oriented languages . . . will possess simpler hierarchies with more type-based behavior defined in terms of shallow, crosscutting interfaces."* In other words, these two authors expect that this schism is resolved by programmers abandoning much of the benefits of inheritance. Our suggestion is that the resolution will be a re-definition of aspects cognizant of the class facets and of inheritance.

2.3 Shakeins as Class Re-implementors

The shakeins construct is a proposal to resolve the aspect-inheritance schism by making aspects which are a restricted form of inheritance. Like inheritance, shakeins generate a new class from an existing one. Yet unlike inheritance, they cannot extend the class type. In fact, they cannot change the base class type at all.

Thus, we can say that *shakeins make a* ***re-implementation*** *of a class*.

Shakeins allow only specific changes to the class facets. Programming constructs which restrict one or more facets are not strange to the community: an *abstract class* is a type, a partial (possibly empty) implementation, and an incomplete mold. *Interfaces* are pure types, with no implementation, forge, mill or mold. Also, *traits* [14] have a fragment of a type and its implementation, but no forge, mill or mold.

Another familiar notion on which shakein rely is that of multiple implementations of the same (abstract) type, as manifested e.g., in the distinction between signatures and structures in ML. The major difference is, however, that a shakein assumes an existing implementation of a class, which is then modified by the shakein.

For concreteness, Fig. 2.3(a) shows a shakein `DisplayUpdating` that can be used to modify class `Point`, so that the display is refreshed after each coordinate change. The shakein works on any class that defines methods `setX(`**`int`**`)`, `setX(`**`int`**`)`, and/or `moveBy(`**`int,int`**`)`. The desired effect is obtained by applying this shakein to `Point`.

[7] In ASPECTJ, two different join point types can be used to apply an advice to method execution: The **`execution`** join point places an advice at the method itself, whereas **`call`** places the advice in at the method's client, i.e., at the point of invocation rather than at its target.

Fig. 2.2 Two shakeins that can be applied to the `Point` class.

```
shakein DisplayUpdating {
  pointcut change() :
       execution(setX(int)) ||
       execution(setY(int)) ||
       execution(moveBy(int,int));

  after() returning: change() {
    Display.update();
  }
}
```

(a) A shakein adding display refresh operations.

```
shakein Confined {
  pointcut update(int v) :
       (set(int x) || set(int y))
       && args(int v);

  before(int v): update(v) {
    if (v < 0)
      throw new IllegalArgumentException();
  }
}
```

(b) A shakein for limiting the valid range of `Point`'s coordinates.

Fig. 2.3(b) shows the `Confined` shakein. This shakein confines the range of valid values for `x` and `y` to positive integers only. It is applicable to class `Point`, or any class with **`int`** fields `x` and `y`.[8]

In the process of re-implementing a class, a shakein may introduce methods and fields to the class. Such members must be **`private`**, accessible to the shakein but not to the external world (including inheriting classes, and classes in the same package). The reason is that such accessibility would have implied a change to the class type, which is not allowed to shakeins. Shakeins are allowed to introduce **`concealed`** members, which are accessible to further re-implementations of this re-implementation.

We will write `Confined<Point>` to denote the application of shakein `Confined` to `Point`, and `DisplayUpdating<Confined<Point>>` for the application of shakein `DisplayUpdating` to the result, etc.

The re-implementation property implies that although a class $S\left\langle c\right\rangle$ can (and usually will) be instantiated, it is not possible to define *variables* of this class. Instances of $S\left\langle c\right\rangle$ can be freely stored in variables of type c, as in the following JAVA like pseudo-code:

c `var =` **`new`** $S\left\langle c\right\rangle$`();`

In the `Point` example, we may then write

```
Point p = new DisplayUpdating<Point>();
```

The type preservation property of shakeins sets a clear semantics for the interaction of these with inheritance. As it turns out, the application of shakeins does not modify or break the inheritance structure. More formally, let c_1 and c_2 be two classes, and suppose that c_2 inherits from c_1. Then, we can write $c_2 \prec c_1$ to denote the fact that the type of c_2 is a subtype of c_1. Let S be a shakein. Then, we can also write $c_1 \simeq S\left\langle c_1\right\rangle$ to denote the fact that the type of $S\left\langle c_1\right\rangle$ is the same as c_1. Similarly, $c_2 \simeq S\left\langle c_2\right\rangle$. By substitution, we can obtain $S\left\langle c_2\right\rangle \prec S\left\langle c_1\right\rangle$, $c_2 \prec S\left\langle c_1\right\rangle$, and $S\left\langle c_2\right\rangle \prec c_1$. In fact, we have

Proposition 1. *For all classes c_1 and c_2 such that $c_2 \prec c_1$, and arbitrary shakeins S and S', $S'\left\langle c_2\right\rangle \prec S\left\langle c_1\right\rangle$.*

In our running example, shakein `DisplayUpdating` can be applied to any subclass of `Point`. If class `ColorPoint` extends `Point`, then the type of `DisplayUpdating <ColorPoint >` is a subtype of `Point`.

Fig. 2.3 makes a graphical illustration of Prop. 1. It depicts a simple base class hierarchy consisting of classes c_1, c_2, c_3, and c_4, where $c_4 \prec c_2 \prec c_1$, and $c_3 \prec c_1$. There are also three shakeins, S_a, S_b and S_c, where shakeins S_b and S_c are implemented using S_a, and each of the shakeins is applied to each of the classes.

[8] While the update presented by this shakein can cause methods that update `x` and `y` to throw an exception, it is an unchecked exception, and therefore it does not alter the methods' signature.

We see in the figure that for all $i = 1, \ldots, 4$, the type of class c_i is the same as its three re-implementations $S_a \langle c_i \rangle$, $S_b \langle c_i \rangle$ and $S_c \langle c_i \rangle$. This common type is denoted by a round-cornered box labeled "Type c_i". As shown in the figure, the subtyping relationship is not changed by re-implementations; e.g., the type of class $S_a \langle c_4 \rangle$ is a subtype of $S_b \langle c_2 \rangle$'s type.

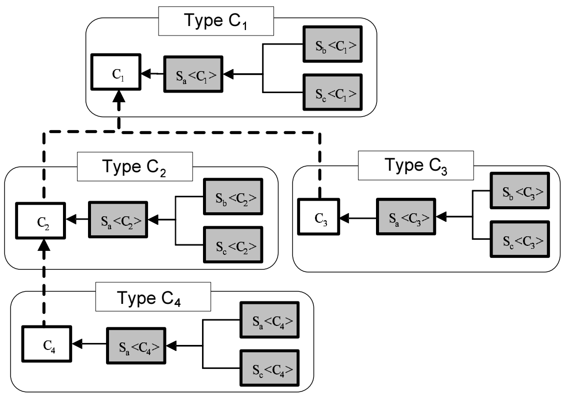

Figure 2.3. A class hierarchy subjected to shakeins.

The figure should also make it clear how the type system of shakeins can be embedded in a standard JVM. Each shakein application results in generation of a JAVA class, which compiles into a distinct `.class` file. Both vertical dashed arrows, representing type inheritance, and horizontal arrows, representing shakein application, are translated to class inheritance, i.e., an **`extends`** relationship between classes.

To see why Prop. 1 holds in this embedding, recall that the program does not have any variables (including parameters and fields) of the shakein classes: All instances of class $S \langle c \rangle$ are stored in variables of type c. In the figure, instances of type $S_a \langle c_4 \rangle$ are stored in variables of type c_4, which is upcastable to type c_2.

2.4 Parameterized Class Modification

Aspects are distinguished from inheritance in that they can be automatically applied to multiple classes. In contrast, a subclass is defined with respect to a specific superclass; the nature of the extension (both of the interface and the materialization) is specific to every such instance. To apply the same kind of change to multiple base classes, the details of the change must be explicitly spelled out each time. Thus, although inheritance is a kind of what is known in the literature as *universal polymorphism* [15], it is not a *uniform* mechanism; each inheriting class modifies the base class in an *ad-hoc* manner.

It is therefore instructive to compare aspects to the parameterized version of inheritance, i.e., *mixins* [10]. A mixin, just like an aspect, makes it possible to apply the same kind of change (expressed in terms of inheritance) to multiple base classes.

Mixins were invented with the observation that there is a recurring need to extend several different classes in the exact same manner. They allow programmers to carry out such a change without rewriting it in each inheriting class. In languages such as MODULA-π [16] the repeating change to the base class can be captured in a mixin M, which, given a class c, generates a class $M \langle c \rangle$ such that $M \langle c \rangle$ inherits from c. In languages with first-class genericity, mixins can be emulated by writing e.g.,

```
template<typename c> class M: public c { ... }
```

in C++, or `class M<c> extends c` in NEXTGEN [17].

Generic structures are also a kind of a universal polymorphism, but their application is *uniform*. The above emulation of mixins by generics makes it clear that mixins are both *universal* and *uniform* kind of polymorphism.

Here, we enrich and simplify the aspects approach with ideas drawn from the work on mixins. We argue that the seemingly pedestrian notation of mixins, $M\left\langle c\right\rangle$, hides much expressive power. The reason is that parameter passing, a familiar mechanism of programming languages, exhibits several useful features which are missing in the current implicit and declarative mechanism of aspect application. These features are:

1. ***Selective Application.*** After a mixin was defined, it can be applied to selected classes without affecting others.
2. ***Non-destructive Application.*** The application of a mixin M to a class c does not destroy c, and both classes $M\left\langle c\right\rangle$ and c can be used. Instances of both may reside in memory simultaneously and even interact with each other.
3. ***Explicit and Flexible Ordering.*** Mixins M_1 and M_2 can be applied in any order, to generate either $M_1\left\langle M_2\left\langle c\right\rangle\right\rangle$, $M_2\left\langle M_1\left\langle c\right\rangle\right\rangle$, or both. Further, it is possible to apply the same mixins in a different orders to different class.
4. ***Composition.*** Mixins can be conveniently thought of as functions, and as such it makes sense to compose them. In some languages supporting mixins one can define a new mixin by writing e.g., $M := M_1 \circ M_2$, with the obvious and natural semantics.
5. ***Configuration Parameters.*** It is straightforward to generalize mixins so that they take additional parameters which control and configure the way they extend the base class. In the templates notation, one can write for example:
```
template<typename c, char *log_file_name>
    class Log: public c {
        // Log into file log_file_name
    }
```
6. ***Repeated Application.*** One can write $M\left\langle M\left\langle c\right\rangle\right\rangle$, but it is not possible to apply the same aspect twice to a class. This works well with parameterizes mixins; e.g.,
   `Log["post.log"]<Security<Log["pre.log"]<c>>>`
   will generate two log files, one before and one after any security checks imposed by the `Security` mixin.
7. ***Parameter Checking.*** It is useful to declare constraints to the parameters of mixins, and apply (meta-) type checking in passing actuals to these. Languages such as JAM [18] offer this feature in mixins.

None of these features is available in ASPECTJ-style aspects, where aspect application is global and destructive, its order globally set and not customizable per target class, aspects cannot be composed and take no class-specific configuration parameters, repeated application is impossible, and there is no parameter checking mechanism (e.g., it is not possible to ensure that aspects are applied only to specific kinds of classes, except in a very narrow sense).

Evidently, these differences are not a coincidence. AOP languages were designed with the belief that the composition of a system from its aspects is an implicit process, carried out by some clever engine which should address the nitty gritty details. One of the chief claims of this paper is that this presumption does not carry to middleware applications, especially when we wish to preserve the investment in existing architecture. Complex systems are composed of many different components in many different ways. An automatic composition engine tends to make arbitrary decisions, whose combined effect is more likely to break larger systems. Our support for this claim is by the detailed description of the aspect oriented re-structuring of J2EE.

Shakeins are similar to mixins in that they take parameters. As such they are a *universal* and *uniform* polymorphic programming construct. Shakeins are similar to generic structures in that they may take multiple parameters. However, whereas both mixins and generics suffer from their *oblivious* and *inflexible* mode of operation, i.e., they are unable to inspect the details of the definition of the base-, or argument- class, and generate specific code accordingly. As a result, mixins fail in tasks such as re-generating constructors with the same signature as the base, or applying the same change to multiple methods in the base class. Similar restrictions apply to generics. In contrast, shakeins use the same pointcut mechanism as aspects, and are therefore highly adaptable[9].

> The component-system Jiazzi [19] uses a mixin-like mechanism to implement *open classes*, which in turn can be used for implementing cross-cutting concerns [20]. Jiazzi components are also parameterized, taking packages (sets of classes) as arguments. However, Jiazzi's open class approach differs from shakeins in that the base class is modified, both in its implementation and in its type; and such changes are propagated to all existing subclasses.

The chief parameter of a shakein is the class that this shakein re-implements. The definition of a shakein does not need to name or even mention this parameter, since it is implicit to all shakeins, in the same way that JAVA methods have a **`this`** parameter. Additional parameters, if they exist, are used to configure or direct the re-implementation[10].

More formally, a shakein S takes an existing class c as input along with other configuration parameters, $\mathcal{P}_1, \ldots, \mathcal{P}_n$, $n \geq 0$, and generates from it a new class $S[\mathcal{P}_1, \ldots, \mathcal{P}_n]\, \langle c \rangle$, such that the *type* of $S[\mathcal{P}_1, \ldots, \mathcal{P}_n]\, \langle c \rangle$ is the *same* as that of c.

Fig. 2.4(a) shows how configuration parameters can enhance the functionality of shakein `Confined` (first shown in Fig. 2.3(b)). As shown in the figure, the updated version of `Confined` is configured by four **`int`** parameters, specifying the minimal and maximal values for the `x` and `y` coordinates of its target class. To obtain an instance of `Point` restricted to the $[0, 1023] \times [0, 767]$ rectangle, one can write

```
Point p = Confined[0,1023,0,767]<Point>(511,383); //Initially at center.
```

Another kind of configuration parameter is a pointcut expression. Fig. 2.4(b) shows a revised version of `DisplayUpdating`, which uses a pointcut parameter. The parameter `change` denotes the join points whose execution necessitates a display update. An actual value of `change` specifies a concrete such set. For example, the following:

```
DisplayUpdating[
   execution(setX(int)) || execution(setY(int)) || execution(moveBy(int,int)
]<Point>
```

is an application of `DisplayUpdating` to class `Point`.

Consider now Fig. 2.5, showing class `Line` implemented using two `Points`. An application of `DisplayUpdating` to `Line` is by writing

```
  DisplayUpdating[execution(moveBy(int,int))]<Line> .
```

[9] Another issue that plagues mixins (and generics that inherit from their argument) is that of *accidental overloading* [18]. Shakeins overcome this problem using an automatic renaming mechanism, which is possible since no shakein-introduced member is publicly accessible.

[10] Note that aspects can also be thought of as taking implicit parameters. However, shakeins are distinguished from aspects in that their invocation is *explicit*, and that aspects take no configuration parameters.

Fig. 2.4 Parameterized shakeins.

```
shakein Confined[int minX, int maxX,
                 int minY, int maxY] {
  pointcut updateX(int v) :
             set(int x) && args(int v);
  pointcut updateY(int v) :
             set(int y) && args(int v);

  before(int v): updateX(v) {
    if ((v < minX) || (v > maxX))
      throw new IllegalArgumentException();
  }

  before(int v): updateY(v) {
    if ((v < minY) || (v > maxY))
      throw new IllegalArgumentException();
  }
}
```

(a) A parameterized version of `Confined`.

```
shakein DisplayUpdating
       [pointcut change()] {

   after() returning: change() {
      Display.update();
   }

}
```

(b) A parameterized version of `DisplayUpdating`.

Fig. 2.5 Class `Line`.

```
class Line {
   private Point a, b;

   public Line(Point from, Point to) {
      a = new Point(from);
      b = new Point(to);
   }
      // cont.

      public moveBy(int x, int y) {
         a.moveBy(x,y);
         b.moveBy(x,y);
      }
   }
```

This re-implementation of `Line` does not suffer from the redundant display updates problem [12], which would have occurred in traditional AOP, i.e., display updates occurring both in the implementation `Line` and its encapsulated `Points`. Thanks to the non-destructive semantics of shakeins, these two `Points` can be of the non-updating variant. This does not prohibit other `Points` in the system (which are not part of `Lines`) to be display-updating.

In contrast, an aspect based solution should check that no `change` advice is in effect before executing this advice. This checking must be carried out *at runtime*, by examining the runtime stack with what is known in AspectJ as a **`cflowbelow`** condition[11].

Fig. 2.6 is a re-definition of the `Confined` shakein using pointcut parameters. Comparing the figure with Fig. 2.4(a), we see that this implementation uses *shakein*

Fig. 2.6 A third version of `Confined`, using a composition of `ConfinedUpdate`.

```
// Auxiliary shakein, used to confine updates to one axis:
shakein ConfinedUpdate[pointcut setValue(int v), int min, int max] {
  before(int v): setValue(v) {
    if ((v < min) || (v > max)) throw new IllegalArgumentException();
  }
}

// Compose the auxiliary shakein twice, once per axis:
shakein Confined[int minX, int maxX, int minY, int maxY] :=
  ConfinedUpdate[set(int x) && args(int v), minX, maxX] ∘
      ConfinedUpdate[set(int y) && args(int v), minY, maxY];
```

[11] Still, **`cflowbelow`**-based pointcuts can be used in shakeins where needed—for example, to prevent a call to `Point.moveBy()` from causing multiple display updates as it changes both point coordinate; see [12] for a discussion of this use of **`cflowbelow`**.

composition. In fact, we have here a *repeated application* of the auxiliary shakein `ConfinedUpdate`, first for the Y-axis, and then, on the result for the Y-axis.

In one sentence summary of this section, we may refine the description of shakeins from Sec. 2.3: *Shakeins make a* ***configurable re-implementation*** *of a* ***class parameter***.

2.5 A New Light on Aspect Terminology

By viewing shakeins are operators, taking classes as arguments and producing classes, we can clarify some of the illusive notions and terms used in traditional AOP jargon:

1. ***Aspect Instantiation.*** The semantics of instantiation of aspects, i.e., the reification of aspects at runtime, can be quite confusing; AspectJ has are as many as five different modes of such instantiations.

   In contrast, shakeins, just like mixins and generics, operate on code, and as such, they no longer exist at runtime. (Of course, the result of a shakein is a class which may have runtime instances.)
2. ***Aspect Precedence.*** AOP languages make it possible to define global precedence rules for aspects. However, this declaration is never complete unless all participating aspects are known; and there is no possibility of applying a set of aspects to different classes in a different order.

   As operators, shakeins can easily be applied in a specific order as the need arises.
3. ***Abstract Pointcut.*** An *abstract pointcut* is a named pointcut with no concrete specification. An *abstract aspect* may include an abstract pointcut and apply advice to it. A *concrete* version of this aspect may be generated by defining a “sub-aspect” for it, which must provide a concrete value to the abstract pointcut. This mechanism provides a measure of flexibility; yet the terms are confusing when used in conjunction with OOP. An abstract pointcut does not offer dynamic binding to the concrete version, nor does it offer a signature that must be obeyed by all its implementors.

   Standing at the shakein point of view, abstract pointcuts are nothing more than a poor man’s replacement for a pointcut parameters.
4. ***Abstract Aspects.*** An aspect is abstract not only when it contains abstract pointcuts, but also when it contains abstract methods. These methods can then be invoked from advice, effectively using the Template Method [21] design pattern. Using concrete sub-aspects that implement these methods, the programmer may define a different aspect each time.

   A shakein may also define a method as abstract. Such a definition overrides the method definition in the shakein parameter. An application of such a shakein yields an abstract class, which can then be extended by other shakeins and provided with a concrete implementation for the missing methods. However, unlike in the case of abstract aspects, the implementation of the abstract methods in a shakein can optionally be different for each application of the shakein.
5. ***Aspect Inheritance.*** Except in the case of abstract super-aspects, the notion of aspect inheritance is ill-defined, and most aspect languages (including AspectJ) prohibit this—because a sub-aspect will clearly share the same pointcut definitions as its parent, and the same advice; does this imply that each advice should therefore be applied *twice* to each matching join point?

As shakeins are operators, shakein inheritance is nothing more than operator composition, as in **`shakein`** `S3<c> = S2<S1<c>>`.

It is therefore evident that while sharing the flexibility and expressive power of aspects, shakeins, by virtue of being parameterized operators on code, reconcile naturally with the concept of inheritance. They do not exhibit the confusing notions that accompany aspects, and their behavior is easy to understand within the domain of aspect-oriented programming.

3 The Case for AOP in J2EE

Having presented the advantages of shakeins, the time has come for evaluating their usefulness. To do so, we would like to go beyond the small examples presented above. The more thorough examination is in the context of a real life application, and in particular, with respect to the J2EE implementation.

In this section, we examine first some of the existing limitations of the service based architecture of J2EE (Sec. 3.1), and then proceed (Sec. 3.2) to explain how the marriage of AOP with J2EE is better served with shakeins, which also help in addressing these limitations.

3.1 Limitations of the Services-Based Solution

Even though the J2EE framework reduces the developer's need for AOP tools, there are limits to such benefits. The reason is that although the EJB container is configurable, it is neither extensible nor programmable. Pichler, Ostermann, and Mezini [22] refer to the combination of these two problems as *lack of tailorability*.

The container is *not extensible* in the sense that the set of services it offers is fixed. Kim and Clarke [6] explain why supporting logging in the framework would require scattered and tangled code. In general, J2EE lacks support for introducing new services for non-functional concerns which are not part of its specification. Among these concerns, we mention memoization, precondition testing, and profiling.

The container is *not programmable* in the sense that the implementation of each of its services cannot be easily modified by the application developer. For example, current implementations of CMP rely on a rigid model for mapping data objects to a relational database. The service is then useless in the case that data attributes of an object are drawn from several tables. Nor can it be used to define read-only beans that are mapped to a database view, rather than a table. The CMP service is also of no use when the persistent data is not stored in a relational database (e.g., when flat XML files are used).

Any variation on the functionality of CMP is therefore by *re-implementation* of object persistence, using what is called *Bean-Managed Persistence* (BMP). BMP support requires introducing callback methods (called *lifecycle methods* in EJB parlance) in each bean. Method `ejbLoad()` (`ejbStore()`), for example, is invoked whenever memory (store) should be updated.

The implication is that the pure business logic of EJB classes is contaminated with unrelated I/O code. For example, the tutorial code of Bodoff *et al.* [23, Chap. 5] demonstrates a mixup in the same bean of SQL queries and a JAVA implementation of functional concern. Conversely, we find that the code in charge of persistence is *scattered* across all entity bean classes, rather than being encapsulated in one cohesive module.

Worse, BMP may lead to code *tangling*. Suppose for example that persistence is optimized by introducing a “dirty” flag for the object’s state. Then, each business logic method which modifies state is tangled with code to update this flag.

Similar scattering and tangling issues rise with modifications to any other J2EE service. In our financial software example, a security policy may restrict a client to transfer funds only out of his own accounts. The funds-transfer method, which is accessible for both clients and tellers, acts differently depending on user authentication. Such a policy cannot be done by setting configuration options, and the method code must explicitly refer to the non-functional concern of security.

To summarize, whenever the canned solutions provided by the J2EE platform are insufficient for our particular purpose, we find ourselves facing again the problems of scattered, tangled and cross-cutting implementation of non-functional concerns. As Duclos, Estublier and Morat [24] state: “*clearly, the ‘component’ technology introduced successfully by EJB for managing non-functional aspects reaches its limits*”.

3.2 Marrying J2EE with AOP

Despite the limitations, the J2EE framework enjoys extensive market penetration, commanding a multi-billion dollar market[12]. In contrast, AOP is only making its first steps into industry acceptance. It is natural to seek a reconciliation of the two approaches, in producing an aspect based, programmable and extensible middleware framework.

Release 4.0 of JBoss (discussed below in Sec. 4.1) is an open-source application server which implements the J2EE standard, and supports aspects with no language extensions[13]. Aspects are implemented in JBoss as JAVA classes which implement a designated interface, while pointcuts are defined in an XML syntax. However, JBoss does not implement J2EE services aspects.

We argue that a proper marriage of AOP and J2EE requires that each of J2EE’s core services is expressed as an aspect. The collection of these services then forms the *core aspect library*, which relying on J2EE success, would not only be provably useful, but also highly customizable. Developers should be able to add their own services (e.g., logging) or modify existing ones, possibly using inheritance in order to re-use proven aspect code. The resulting aspects could then be viewed as stand-alone modules that can be re-used across projects.

Another implication is that not all aspects must come from a single vendor; in the current J2EE market, all J2EE-standard services are provided by the J2EE application server vendor. If developers can choose which aspects to apply, regardless of the application server used, then aspects implemented by different vendors (or by the developers themselves) can all be used in the same project.

Choi [7] was the first to demonstrate that an EJB container can be built from the ground up using AOP methodologies, while replacing services with aspects which exist independently of the container.

[12] http://www.serverwatch.com/news/article.php/1399361
[13] http://www.onjava.com/lpt/a/3878

Focal to all this prior work was the attempt to make an existing widespread framework more robust using AOP techniques. Our work here adopts a new approach to the successful marriage of J2EE and AOP, by using the notion of shakeins. More concretely, we propose a new AOP language ASPECTJ2EE, which in using shakeins, draws from the lessons of J2EE and its programming techniques.

The main issues in which the ASPECTJ2EE language differs from ASPECTJ are:

1. *Aspect targets.* ASPECTJ can apply aspects to any class, whereas in ASPECTJ2EE aspects can be applied to *enterprise beans* only, i.e., those modules to which J2EE services are applied. (This selective application is made possible by the shakein semantics, which always have a designated target.)
   As demonstrated by the vast experience accumulated in J2EE, aspects have great efficacy precisely with these classes. We believe that the acceptance of aspects by the community may be improved by narrowing their domain of applicability, which should also benefit understandability and maintainability.
   It should be stressed, however, that this is not a limitation of the shakein concept but rather an ASPECTJ2EE design decision.
2. *Weaving method.* Weaving the base class together with its aspects in ASPECTJ2EE relies on the same mechanisms employed by J2EE application servers to combine services with the business logic of beans. This is carried out entirely within the dominion of object oriented programming, using the standard JAVA language, and an unmodified JVM. Again, this is made possible by the shakein semantics.
   In contrast, different versions of ASPECTJ used different weaving methods relying on preprocessing, specialized JVMs, and dedicated byte code generators, all of which deviate from the standard object model.
3. *Aspect parametrization.* Since the aspects in ASPECTJ2EE are shakeins, they take three kinds of parameters: pointcut definitions, types and literal values. Parameterized aspects can be applied to EJBs by providing (in the EJBs deployment descriptor) a concrete value for each parameter, including concrete pointcut definitions. Pointcut parameters provide significant flexibility by removing undesired cohesion between aspects and their target beans, and enables the development of highly reusable aspects. It creates, in ASPECTJ2EE, the equivalent of Caesar's [25] much-touted separation between aspect implementation and aspect binding.
   Other aspect parameter types also greatly increase aspect reusability and broaden each aspect's applicability.
4. *Support for tier-cutting concerns.* ASPECTJ2EE is uniquely positioned to enable the localization of concerns that cross not only program modules, but program tiers as well. Such concerns include, for example, encrypting or compressing the flow of information between the client and the server. Even with AOP, the handling of tier-cutting concerns requires scattering code across at least two distinct program modules. We show that using ASPECTJ2EE, many tier-cutting concerns can be localized into a single, coherent program module.

J2EE application servers offer the developer only minimal control over the generation of support classes. ASPECTJ2EE however, gives a full AOP semantics to the deployment process. With deploy-time weaving, described next, the main code is unmodified, both at the source and the binary level. Further, the execution of this code is unchanged, and can be carried out on any standard JVM.

ASPECTJ2EE does not impose constraints on the base code, other than some of the dictations of the J2EE specification on what programmers must, and must not, do while defining EJBs. These dictations are that instances must be obtained via the Home interface, rather than by directly invoking a constructor or any other user-defined method; business methods must not be **`final`** or **`static`**; and so forth.

4 Comparison with JBoss and Spring

Obviously, we were not the first to observe the case for using aspects in middleware applications in general, and in J2EE in particular. Indeed, there is a large body of previous work in which aspects are applied in the middleware domain: JAC [26], Lasagna [27], PROSE [28], JAsCo [29], and others.

Shakeins were designed with J2EE in mind—so in this section we compare shakeins with the two prime applications of AOP technology to J2EE. Sec. 4.1 compares shakeins with the dynamic aspects of *JBoss Application Server*[14]. A comparison with the AOP features of the *Spring Application Framework*[15] is the subject of Sec. 4.2. Unlike shakeins and other similar research, JBoss and Spring are industrial-strength software artifacts employed in production code. The lessons that these two teach are therefore valuable in appreciating the merits of shakeins.

4.1 JBoss AOP

Version 4.0 of JBoss was the first implementation of J2EE to integrate AOP support. For technical reasons, the *JBoss AOP* approach features advice- rather than aspect-level granularity, where each advice is encapsulated in what is called an *interceptor* class.

It is telling that JBoss AOP enhancements of the aspect notion are similar to these of shakeins, including: advice composition (*stacks* in the JBoss jargon), parameterized advice, selective and repeated application of advice, explicit and flexible (rather than global) ordering, and configuration parameters. We interpret this as a supporting empirical support to the claim that flat-oblivious aspects should be extended in certain ways.

Still, since the application of (standard) advice in JBoss is carried out in situ, destroying the original class, the JBoss approach suffers from the aspect/inheritance schism, instantiation complexity, etc. Perhaps in recognition of these difficulties, JBoss AOP also supports *dynamic AOP*—the ability to apply advice per instance rather than per class. A class must be "prepared" for such a dynamic application, by injecting into its code (at load time) hooks for the potential join points. The class loader consults an XML configuration file for the list of classes to prepare, and the hook locations. It is then possible, at run time, to add or remove interceptors.

In this extent, JBoss's flexibility and repertoire of features is greater than that of shakeins. JBoss offers the ability to *un-apply* an advice at runtime. This feature is missing in shakeins, but can be emulated by testing an on/off flag at the beginning of every advice. Conversely, note that JBoss's flexibility has its inherent limits; e.g., since interceptors are applied to existing objects, advice cannot be applied to constructors.

[14] http://www.jboss.org
[15] http://www.springframework.org

The most major difference between JBoss AOP and shakeins is the approach taken for integration with the base technology. As explained in Sec. 2, shakeins are a language extension, which draws from principles of OO and genericity. In contrast, the variety of features in JBoss AOP is realized by a sophisticated combination of loaders, runtime libraries, and XML configuration files, and without any changes to the compiler (or the JVM). Thus, (probably in answer to Sun's J2EE certification requirements) JBoss AOP is an implementation of aspects through a software framework built over vanilla JAVA.[16]

Since JBoss aspects are not part of the language, a programmer who wishes to exploit aspect features is asked to master a variety of tools, while following the strict discipline dictated by the framework, with little if any compiler checking. For example, the runtime application of advice is achieved using a JAVA API; the compiler is unaware of the involved AOP semantics. As a result, many mistakes (e.g., an attempt to access an invalid method argument), can only be detected at runtime, possibly leading to runtime errors. Other problems, such as a mistyped pointcut (which matches no join points) will not be detected at all. Thus, in a sense, the offers of JBoss can be compared to assembly programming: immense flexibility but with greater risks. However, unlike assembly code, performance in JBoss is not at its peak.

The clockwork driving the JBoss framework is visible to the programmer. The programmer *must* understand this mechanism in order to be able use it. This visibility has its advantages: the illusive issue of aspect instantiation in ASPECTJ is clarified by JBoss: since interceptors must be instantiated explicitly prior to their application, the semantics of aspect instance management is left up to the programmer.

To illustrate some of the issues of a framework based implementation of aspects, consider Fig. 4.1, which demonstrates how the `Confined` shakein (Fig. 2.6) is implemented in JBoss. Fig. 4.1(a) depicts a JAVA class which, by obeying the framework rules, can be used as an interceptor. The runtime content of argument `inv` to method

Fig. 4.1 **(a)** `Confined` as a JBoss interceptor, and **(b)** the supporting configuration file.

(a)
```
public class Confined implements Interceptor {
  private int min, max;
  private String fieldName;

  public Confined(String fieldName, int min, int max) {
     this.min = min; this.max = max;
     this.fieldName = fieldName;
  }

  public String getName() { return "Confined"; }

  public Object invoke(Invocation inv) throws Throwable {
     FieldWriteInvocation fwi = (FieldWriteInvocation)inv;
     int v = (Integer)(fwi.getValue());
     if (fwi.getField().getName().equals(fieldName))
        if (v < min || v > max) throw new IllegalArgumentException();
     return inv.invokeNext(); // proceed to subsequent interceptors/base code
  }
}
```

(b)
```
<aop>
   <prepare expr="set(int Point->x) OR set(int Point->y)" />
</aop>
```

[16] Compliance with these requirements also explains why standard J2EE services are not implemented as aspects in JBoss, and are therefore not as flexible as they might be.

`invoke` (lines 12–18) is the only information that the method has on the interception. The method assumes (line 13) that the join point kind is a field-write. The newly assigned value is obtained (line 14) by downcasting an `Object` to the proper type. Both downcasts will fail if the interceptor is applied to an incorrect join point.

Fig. 4.1(b) is the XML code that directs the injection of the hooks intended for this interceptor into class `Point`. The interceptor is intimately coupled with the XML, in making tacit assumptions on the join point kind and the argument type; violations of these assumptions are only detected at runtime.
To create a shakein-typed instance of `Point`, one may write

```
Point p = Confined[0,1023,0,767]<Point>(); //Shakein version.
```

The JBoss equivalent is a bit longer:

```
Point p = new Point(); // JBoss dynamic AOP version.
((Advised)p)._getInstanceAdvisor().appendInterceptor(new Confined("x", 0, 1023));
((Advised)p)._getInstanceAdvisor().appendInterceptor(new Confined("y", 0, 767));
```

This demonstrates more intricacies of implementing aspects by a software framework: First, we see that advices are applied only after the object was constructed (no refinement of the constructors is possible). Second, since there is no explicit composition operator[17], two interceptors must be manually applied, one per axis. Third, we see that `p` must be casted to the interface type `Advised`. This interface type is implemented by the modified (prepared) version of `Point`; yet the compiler is not aware of this change. If the class was not prepared (e.g., an inconsistency in the XML file), then this cast attempt will fail. Finally, again due to compiler obliviousness, field names are represented as string literals (here, as arguments to the interceptor's constructor). Any mistake in the field name (e.g., writing "`X`" instead of "`x`") will go undetected and result in silent failure. (By comparison, an empty pointcut argument to the auxiliary shakein from Fig. 2.6 triggers a compile-time warning.)

Fig. 4.2 compares the runtime performance of the JBoss and the shakein implementation of aspect `Confined`. The figure depicts the operations throughput of class `Point` in the base implementation and in the different aspectualized versions.[18] We see that the original class suffers no performance penalty in the shakein version. The shakein-advised instance, generated by two subclassing operations, is about 8% slower. In contrast, while using JBoss AOP, instances of the original class suffer from a performance impact of about 28% before any advice is applied; this is the overhead introduced by the join point hooks. Once applied, the reflection-based interceptors slow the JBoss version to 1.5% of the original throughput.

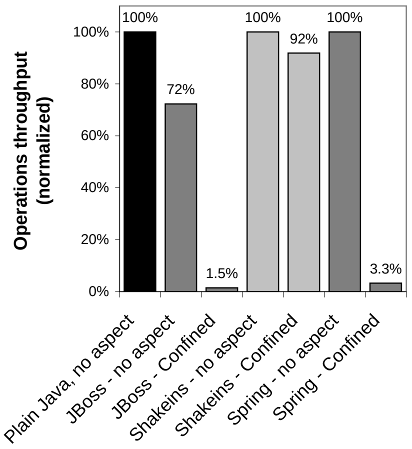

Figure 4.2. Performance degradation of class `Point` with different strategies of applying a "confinement" aspect.

There are two main sources of performance degradation in the JBoss implementation. *Time wise,* hooks slow down code, and this slowdown occurs even if no advices are applied to the receiver instance. (In

[17] Stacks cannot be used here.

[18] Specifically we used the number of times the sequence of calls ⟨`setX`,`setY`,`moveBy`⟩ can be completed in a time unit.

class `Point`, this slowdown was by 28%.) Moreover, even a non-prepared class may be slowed down, if its code matches, e.g., a join point of a **`call`** to a prepared class.

Additional slowdown is caused by the advice having to use reflection-like objects in order to learn about the join point. The invocation object must be downcast to the specific invocation type (`FieldWriteInvocation` in Fig. 4.1(a)), and arguments must be downcast from `Object` references to their specific types. (As we have seen, this slowdown was by more than an order of magnitude in `Point`. We expect a more modest relative performance degradation in classes that do more substantial computation.)

Space wise, the advice (whenever a join point is reached) is reified in a runtime object. The memory consumed by this object must be managed, thereby leading to additional slowdown. The invocation itself is reified in a number of objects (metadata, argument wrappers, arguments array, etc.) which add to the space and time overheads of the implementation. (A quick inspection of Fig. 4.1(a) reveals that there are at least four objects generated in reifying the join point.)

4.2 A Comparison with Spring AOP

The Spring Application Framework is an “inversion of control”[19] container, used for applying services to standard JAVA objects. It is often used in conjunction with a J2EE server for developing enterprise applications; however, Spring provides alternatives to many of the J2EE services, in a more flexible and developer-friendly manner.

Objects in Spring are “beans”, all obtained via a centralized factory which is configured using an XML file. This XML file specifies what properties should be set and what services applied to each bean type. Developers can choose from a wide range of predefined services (e.g., Hibernate[20]-based persistence) or define their own. New services (as well as existing ones) are defined using Spring’s AOP facilities[21].

Much like shakeins, AOP in Spring is based on the generation of new classes. When advice is applied to a class, a new class is generated, which either implements the same interfaces as the base class or else extends it as a subclass. Thus, Spring enjoys several of the benefits of shakeins; most notably, there is no performance penalty to instances of the original class, which remains unmodified (see Fig. 4.2). However, beyond this similarity, there are several differences of note between the Spring and shakein approaches.

Advice in Spring is manifested as an interceptor class, which is invoked whenever an advised method is executed. Pointcuts are also manifested as classes, and interrogated at runtime to find out which methods should be advised. Much as in JBoss, the mechanism relies on a sophisticated combination of libraries and configuration files, with no changes to the language itself. Therefore, Spring AOP shares much of the tolls noted for JBoss AOP, including similar space and time complexities (with the exception of hooks-induced slowdowns). Additional performance penalties are caused by the need to evaluate pointcuts at runtime, as well as the runtime generation of subclasses.

As a design decision, Spring AOP only support method invocation join points (and that, only for non-**`private`** methods). In our tests, the lack of support for field access

[19] http://www.martinfowler.com/articles/injection.html
[20] http://www.hibernate.org
[21] Spring also supports the integration of standard ASPECTJ aspects.

join points implied that the `Confined` aspect had to be made explicitly aware of each of the `Point` methods that can update the point's coordinates; in particular, the advice had to re-create the logic for the `moveBy` method. The Spring point of view contends that this would not have been needed, had `moveBy` relied on `setX` and `setY` to update the fields, rather than using direct access (recall Fig. 2.2(a)). But from this very claim we must conclude that the Spring aspect is fragile with respect to changes in the implementation of `Point`; should `moveBy` be updated to rely on the setter methods, the advice must be accordingly updated. A non-fragile aspect implementation must rely on examining the control-flow at runtime, which a noticeable performance hit.

In our benchmarks (Fig. 4.2), the Spring-based `Confined` aspect (which was not created using composition, due to its asymmetry with regard to `moveBy`) was over twice as fast as the JBoss version, but still much slower than the shakeins-based version.

5 Weaving, Deployment and Deploy-Time Weaving

Now that the theoretical foundation of the shakeins construct was established, and that we understand how and why it may be useful in the context of middleware frameworks, the time has come to combine the two. The first step is in describing how deployment, a basic technique of J2EE, can be generalized for the process of weaving aspects (specifically, shakeins) into an application.

Sec. 5.1 explains weaving. Deployment is the subject of Sec. 5.2. Weaving of shakeins onto EJBs is discussed in Sec. 5.3. Sec. 5.4 generalizes this process to arbitrary classes.

5.1 Weaving

Weaving is the process of inserting the relevant code from various aspects into designated locations, known as *join points*, in the main program. In their original presentation of ASPECTJ [9], Kiczales *et al.* enumerate a number of weaving strategies: "*aspect weaving can be done by a special pre-processor, during compilation, by a post-compile processor, at load time, as part of the virtual machine, using residual runtime instructions, or using some combination of these approaches.*" Each of these weaving mechanisms was employed in at least one AOP language implementation. As mentioned before, our implementation of shakeins use its own peculiar *deploy-time weaving* strategy. In this section we motivate this strategy and explain it in greater detail.

We first note that the weaving strategies mentioned in the above quote transgress the boundaries of the standard object model. Patching binaries, pre-processing, dedicated loaders or virtual machines, will confuse tools such as debuggers, profilers and static analyzers, and may have other adverse effects on generality and portability.

Further, weaving introduces a major *conceptual* bottleneck. As early as 1998, Walker *et. al* [30] noted the disconcert of programmers when realizing that merely reading a unit's source code is not sufficient for understanding its runtime behavior[22].

[22] Further, Laddad [8, p. 441] notes that in ASPECTJ the runtime behavior cannot be deduced even by reading *all* aspects, since their application to the main code is governed by the command by which the compiler was invoked.

The remedy suggested by Constantinides *et. al* in the *Aspect Moderator* framework [31] was restricting weaving to the dominion of the OOP model. In their suggested framework, aspects and their weaving are realized using pure object oriented constructs. Thus, every aspect oriented program can be presented in terms of the familiar notions of inheritance, polymorphism and dynamic binding. Indeed, as Walker *et al.* conclude: "*programmers may be better able to understand an aspect-oriented program when the effect of aspect code has a well-defined scope*".

Aspect Moderator relies on the PROXY design pattern [21] to create components that can be enriched by aspects. Each core class has a proxy which manages a list of operations to be taken before and after every method invocation. As a result, join points are limited to method execution only, and only **before**`()` and **after**`()` advices can be offered. Another notable drawback of this weaving strategy is that it is *explicit*, in the sense that every advice has to be manually registered with the proxy. Registration is carried out by issuing a plain JAVA instruction—there are no external or non-JAVA elements that modify the program's behavior. Therefore, long, tiresome and error-prone sequences of registration instructions are typical to Aspect Moderator programs.

A better strategy of implementing explicit weaving is that this code is generated by an automatic tool from a concise specification. The shakein weaving mechanism gives in essence this tool. However, rather than generate explicit weaving code for a proxy, it generates a woven version of the code in a *newly generated subclass*. By replacing the proxy pattern with the notion of subclassing, it also able to handle advice types other than **before**`()` and **after**`()`, and handle a richer gamut of join point types, as detailed in Sec. 6.3.

Thus, shakeins do not use any of the obtrusive weaving strategies listed above. Instead, the mechanism employs a weaving strategy that *does not break* the object model. Instead of modifying binaries (directly, or by pre-processing the source code), the application of a shakein to a class results in an "under the hood" generation of a new class that inherits from, rather than replaces, the original. The new class provides an alternative realization, a re-implementation, of the same type; it does not introduce a new type, since there are no visible changes to the interface. This re-implementation is generated by advising the original one with the advice contained in the shakein.

Clearly, there are limitations to the approach of implementing shakeins as subclasses. The main such limitation is that Prop. 1 does not hold in the general case. Below we will show that in the particular case of EJBs in J2EE, this restriction does not arise, because access to EJBs is through interfaces.

5.2 Deployment

J2EE offers a unique opportunity for generating the subclasses required for the weaving of shakeins. Fig. 5.1 compares the development cycle of traditional and J2EE application. We see that *deployment* is a new stage in the program development process, which occurs after compilation but prior to execution. It is unique in that although new code is generated, it is not part of the development, but rather of user installation.

Deployment is the magic by which J2EE *services*, such as security and transaction management, are welded to applications. The generation of sub- and support classes is governed by *deployment descriptors*, which are XML configuration files.

The idea behind deploy-time weaving is to extend this magic, by placing the shakein semantics in government of this process. As shakeins are based on straightforward inheritance, this extension also simplifies the structure of and inter-relationships between the generated support classes.

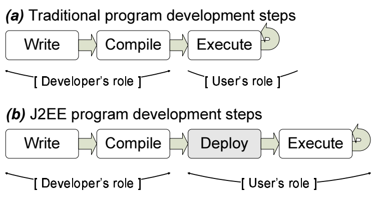

Figure 5.1. Program development steps: **(a)** traditional, **(b)** J2EE.

Technically, *deployment* is the process by which an application is installed on a J2EE application server. Having received the application binaries, deployment involves generating, compiling and adding additional support classes to the application. For example, the server generates *stub* and *tie* (skeleton) classes for all classes that can be remotely accessed, in a manner similar to, or even based on, the RMI compiler[23].Even though some J2EE application servers generate support class binaries directly (without going through the source), these always conform to the standard object model.

We must study some of the rather mundane details of deployment in order to understand how it can be generalized to do weaving. To do so, consider first Fig. 5.2, which shows the initial hierarchy associated with an ACCOUNT CMP bean (see Sec. 1.2).

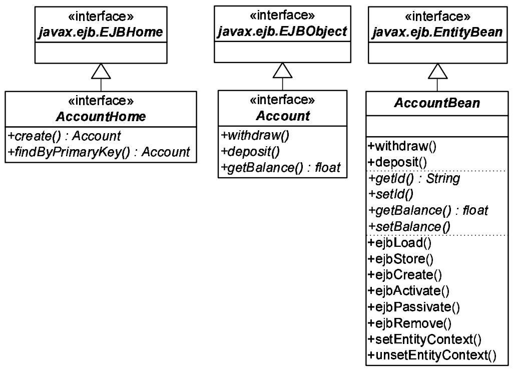

Figure 5.2. Programmer-created classes for the ACCOUNT EJB.

At the right hand side of the figure, we see interface `Account`, which inherits from `javax.ejb.EJBObject`. This interface is written by the developer in support of the remote interface to the bean[24]. This is where all client-accessible methods are declared. In the example, there are three such methods: `withdraw()`, `deposit()`, and `getBalance()`. Class `Account` resides at the *client side*.

On the left hand side of the figure, we see abstract class `AccountBean`, inheriting from `javax.ejb.EntityBean`. The J2EE developer's main effort is in coding this class, which will reside at the *server side*. There are three groups of methods in the bean class:

1. *Business Logic.* The first group of methods in this class consists the implementation of business logic methods. These are `deposit()` and `withdraw()` in the example.
2. *Accessors of Attributes.* EJB *attributes* are those fields of the class that will be governed by the persistence service in the J2EE server. Each attribute `attr` is represented by abstract setter and getter methods, called `setAttr()` and `getAttr()` respectively. Attributes are not necessarily client-accessible.
   In the example, there are four such accessors, indicating that the bean ACCOUNT has two attributes: `id` (the primary key) and `balance`. Examining the `Account` interface we learn that `id` is invisible to the client, while `balance` is read-only accessible.
3. *Lifecycle.* The third and last method group comprises a long list of mundane lifecycle methods, such as `ejbLoad()` and `ejbStore()`, most of which are normally empty.

[23] http://java.sun.com/j2se/1.5.0/docs/guide/rmi/

[24] For the sake of simplicity, we assume that ACCOUNT has a remote interface only, even though beans can have either a local interface, a remote interface, or both.

Even though sophisticated IDEs can produce a template implementation of these, they remain a developer's responsibility, contaminating the functional concern code. Later we shall see how deploy-time weaving can be used to remove this burden.

Finally, at the center of Fig. 5.2, we see interface `AccountHome`, which declares a FACTORY [21] of this bean. Clients can only generate or obtain instances of the bean by using this interface.

Concrete classes to implement `AccountHome`, `Acount` and `AccountBean` are generated at deployment time. The specifics of these classes vary with the J2EE implementation. Fig. 5.3 shows some of the classes generated by IBM's WebSphere Application Server (WAS)[25] version 5.0 when deploying this bean.

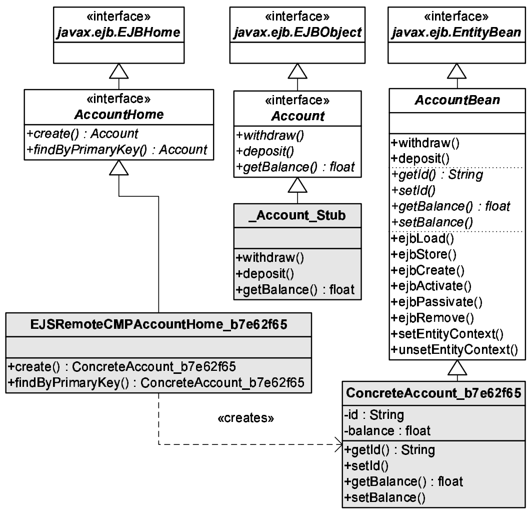

Figure 5.3. ACCOUNT classes defined by the programmer, and support classes (in gray) generated by WAS 5.0 during deployment.

Examining the figure, we see that it is similar in structure to Fig. 5.2, except for the classes, depicted in gray, that the deployment process created: `Concrete-Account_b7e62f65` is the *concrete bean class*, implementing the abstract methods defined in `AccountBean` as setters and getters for the EJB attributes. Instances of this class are handed out by class `EJSRemoteCMP-AccountHome_b7e62f65`, which implements the factory interface `AccountHome`.

Finally, `_Account_Stub`, residing at the client side, intercommunicates with `Concrete-Account_b7e62f65` which resides at the server side.

In support of the ACCOUNT bean, WAS deployment generates several additional classes which are not depicted in the figure: a stub for the home interface, ties for both stubs, and more. Together, the deployment classes realize services that the container provides to the bean: persistence, security, transaction management and so forth. However, as evident from the figure, all this support is provided within the standard object oriented programming model.

5.3 Deployment as a Weaving Process for EJBs

Having understood the process of deployment and the generation of classes in it, we can now explain how deployment can be used as a weaving process. Consider first the ordered application of four standard ASPECTJ2EE shakeins: `Lifecycle`, `Persistence`, `Security` and `Transactions` to the bean ACCOUNT. (Such a case is easier than the more general case, in which the target class is not an EJB. We will discuss this issue below.)

[25] http://www.ibm.com/software/websphere/

Weaving by deployment generates, for each application of an aspect (or a shakein) to a class, a subclass of the target. This subclass is called an *advised class*, since its generation is governed by the advices given in the aspect. Accordingly, the sequence of applications under consideration will generate four advised classes.

Fig. 5.4 shows the class hierarchy after the deployment tool generated these four class in support of the shakein application expression

`Transactions<Security<Persistence<Lifecycle<Account>>>>`.

Comparing this figure to Fig. 5.3, we see first that the class `AccountBean` was shortened by moving the lifecycle methods to a newly defined class, `AdvAccount_Lifecycle`. The shakein `Lifecycle` made it possible to eliminate the tiring writing of token (and not always empty) implementations of the lifecycle methods in each bean. All these are packaged together in a standard `Lifecycle` aspect[26].

`AdvAccount_Lifecycle` is the advised class realizing the application of `Lifecycle` to ACCOUNT. There are three other advised classes in the figure, which correspond to the application of aspects `Persistence`, `Security` and `Transactions` to ACCOUNT.

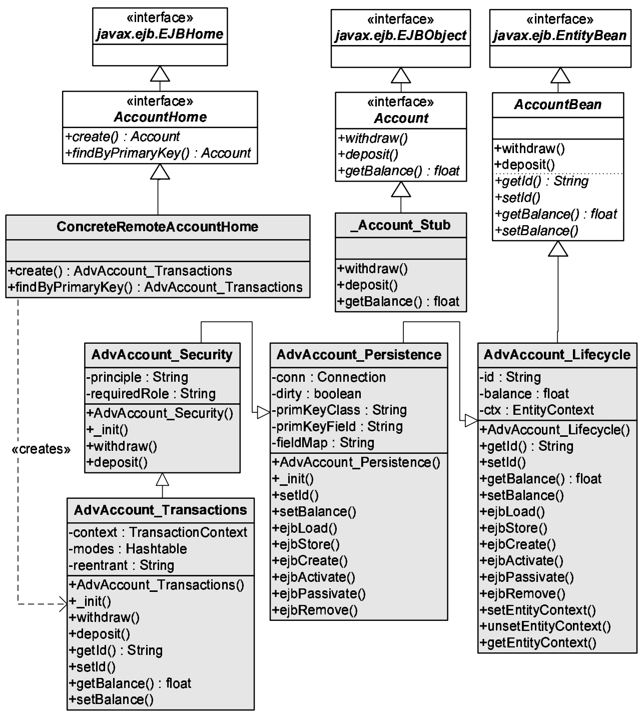

Figure 5.4. ACCOUNT classes defined by the programmer, and support classes (in gray) generated by ASPECTJ2EE deployment.

The sequence of aspect applications is translated into a chain of inheritance of advised classes, starting at the main bean class. The *root advised class* is the first class in this chain (`AdvAccount_Lifecycle` in the example), while the *terminal advised class* is the last (`AdvAccount_Transactions` in the example). Fields, methods and inner classes defined in an aspect are copied to its advised class. *Advised methods* in this class are generated automatically based on the advices in the aspect.

We note that although all the advised classes are concrete, only instances of the terminal advised class are created by the bean factory (the generated EJB home). In the figure for example, class `ConcreteRemoteAccountHome` creates all ACCOUNTs, which are always instances of `AdvAccount_Transactions`.

[26] The lifecycle methods are declared in the interface `javax.ejb.EntityBean`. Hence, implementing them in a shakein does not change the type of class `AccountBean`.

It may be technically possible to construct instances of this bean in which fewer aspects are applied; there are, however, deep theoretical reasons for preventing this from happening. Suppose that a certain aspect applies to a software module such as a class or a routine, etc., in all but some exceptional incarnations of this module. Placing the tests for these exceptions at the point of incarnation (routine invocation or class instantiation) leads to scattered and tangled code, and defeats the very purpose of AOP. The bold statement that some accounts are exempt from security restrictions should be made right where it belongs—as part of the definition of the security aspect! Indeed, J2EE and other middleware frameworks do not support conditional application of services to the same business logic. A simple organization of classes in packages, together with JAVA accessibility rules, enforce this restriction and prevents clients from obtaining instances of non-terminal advised classes.

5.4 Deploy Time Weaving for General Classes

We just saw that deploy time weaving generates, at deployment time, an advised class for each application of an aspect of ASPECTJ2EE. Let us now consider the more general case, in which the target class is not an EJB.

It is instructive to compare the advising of EJBs (Fig. 5.4) with the general structure of shakein classes, as depicted in Fig. 2.3. We see that the diagrams are similar in making each aspect application into a JAVA class. However, Fig. 5.4 adds two factors to the picture: First, the generation of instances of ACCOUNT is controlled by an external *factory class*. Second is the fact that the class `Account` is abstract.

Together these two make one of the key properties of EJBs, namely the fact that an EJB does not have a *forge facet*. Instead, the framework imposes a requirement that all instances of the class are obtained from an external class, which follows the ABSTRACT FACTORY design pattern.

This property makes it possible to apply an aspect, a service, or a shakein to *all* instances of a certain class. When applying the deploy time weaving technique to non-EJB classes, one may chose to degenerate the forge facet of the target class, as in EJBs, or in the case that this is not possible, make sure that the correct constructors are invoked in the code.

6 The ASPECTJ2EE Programming Language

Having described the shakeins construct and deploy-time weaving, we are ready to describe the ASPECTJ2EE language.

The syntax of ASPECTJ2EE is a variant of ASPECTJ. The semantics of ASPECTJ2EE-aspects is based on a (limited) implementation of the shakein concept. Hence, aspects in ASPECTJ2EE (unlike in ASPECTJ) do not have a global effect, and are woven into the application at deployment time (rather than compile time).

When compared to shakeins, the main limitation of ASPECTJ2EE-*aspects* (henceforth just "aspects", unless noted otherwise), is that their application to classes is governed by an external *deployment descriptor file*, written in XML. Accordingly, ASPECTJ2EE does not provide a syntax for explicitly applying aspects to classes. Consequently, the integration of aspects into ASPECTJ2EE is not complete. Indeed, ASPECTJ2EE suffers from two XML-JAVA coupling issues: (i) JAVA code using a class whose generation is governed by XML is coupled with this XML code. (ii) XML file applying an aspect to a ASPECTJ2EE class is coupled with the ASPECTJ2EE names. However, in contrast with JBoss aspects, the detection of errors due to such coupling, i.e., using wrong class names or illegal or empty pointcut expressions, is not at run time, but rather at deployment time.

Comparing the ASPECTJ2EE version of shakeins with the theoretical description of the concept, we find that some of the benefits (see Sec. 2.4) are preserved, while others are not:

1. *Selective application* is available; aspects are applied only to classes specified in the deployment descriptor.
2. *Non-destructive application* is preserved. However, instances are obtained using Home objects (factories) only. Therefore, the programmer, wearing the hat of an *application assembler*,can dictate which combinations of aspect application are available. For example, it is possible to ensure that all instances of ACCOUNT are subjected to a security aspect.
3. *Explicit and Flexible Ordering* is provided by the XML binding language.
4. *Composition* is not supported; there is no syntax for composing two or more aspects.
5. *Configuration parameters* are available; the deployment descriptor is used for argument passing.
6. *Repeated application* is fully supported.
7. *Parameter checking* is absent.

Sec. 6.1 presents the language syntax. The application of aspects through deployment descriptors is the subject of Sec. 6.2. Sec. 6.3 explains how deploy-time weaving can implement the various kinds of join points. Finally, Sec. 6.4 gives a broad overview of the standard aspect library.

6.1 Language Syntax

The major difference between ASPECTJ2EE and ASPECTJ is that ASPECTJ2EE supports parameterized aspects. For example, Fig. 6.1 shows the definition of a role-based security shakein that accepts two parameters. The first parameter is a pointcut definition specifying which methods are subjected to a security check. The second is the user role-name that the check requires.

Fig. 6.1 The definition of a simple `Security` aspect in ASPECTJ2EE.

```
aspect Security[pointcut secured(), String requiredRole] {
   before(): secured() {
      if (!userInRole(requiredRole)) {
         throw new RuntimeException("Security Violation");
      }
   }

   private boolean userInRole(String roleName) {
      // Check if the currently active user has the given role...
   }
}
```

Parameter values must be known at deploy time. Accordingly, there are four kinds of parameters for aspects:

– *Type parameters*, preceded by the keyword **`class`**. The type can be restricted (like type parameters in JAVA generics) using the **`implements`** and **`extends`** keywords.
– *Pointcut parameters*, preceded by the **`pointcut`** keyword.
– *String parameters* and *primitive type parameters*, preceded by the type name (`String`, **`int`**, **`boolean`**, etc.).

In contrast with ASPECTJ, the scope of a specific aspect application in ASPECTJ2EE is limited to its target class. Therefore, any pointcut that refers to join points in other classes is meaningless. Accordingly, ASPECTJ2EE does not have a **`call`** join point, since it refers to the calling point, rather than the execution point, of a method. (To apply advice to method execution, an **`execution`** join point can be used.) This restriction is a direct result of the shakein semantic model, and it eliminates the confusion associated with the **`call`** join point in relation to inheritance (see Sec. 2.2).

All other join point kinds are supported, but with the understanding that their scope is limited to the target class; for example, a field-set join point for a **`public`** field will not capture access to the field from outside its defining class. ASPECTJ2EE also introduces a new kind of join point for handling remote invocation of methods.

Since the application of aspects in ASPECTJ2EE is explicit, it does not recognize the ASPECTJ statement **`declare precedence`**.

Finally, there is a subtle syntactical difference due to the "individual target class" semantics of ASPECTJ2EE aspects: The definition of a pointcut should not include the target class name as part of method, field or constructor signatures. Only the member's name, type, access level, list of parameter types, etc. can be specified. For example, the signature matching any **`public void`** method accepting a single `String` argument is written ASPECTJ as **`public void`** `*.*(String)`. The same signature should be written as **`public void`** `*(String)` in ASPECTJ2EE. The ASPECTJ form applies to the methods with this signature in *all* classes, whereas the ASPECTJ2EE form applies only to such methods in the class to which the containing aspect is applied.

6.2 The Deployment Descriptor

In ASPECTJ, the application of aspects to classes is specified declaratively. Yet the process is not completely transparent: the application assembler must take explicit actions to make sure that the specified aspect application actually takes place. In particular, he must remember to compile each core module with all the aspects that may apply to it. (Or else, an aspect with global applicability may not apply to certain classes if these classes were not compiled with it.)

The order of application of aspects in ASPECTJ is governed by **`declare precedence`** statements; without explicit declarations, the precedence of aspects in ASPECTJ is undefined. Also, ASPECTJ does not provide any means for passing parameters to the application of aspects to modules.

In contrast, the shakeins semantics in general, and ASPECTJ2EE in particular, require an explicit specification of each application of an aspect to a class, along with any configuration parameters. This specification could have been done as part of the programming language. But, following the conventions of J2EE, and in the sake of minimizing the syntactical differences between AspectJ2EE and AspectJ, we chose to place this specification in an *external* XML deployment descriptor.

In fact, we shall see that the XML specification is in essence the abstract syntax tree, which would have been generated from parsing the same specification if written inside the programming langauge. Fig. 6.2 gives an example, showing the sequence of application of aspects to Account which generated the classes in Fig. 5.4. Overall, four aspects are applied to the bean: `Lifecycle`, `Persistence`, `Security`, and `Transactions`. All of these are drawn from the `aspectj2ee.core` aspect library.

Fig. 6.2 A fragment of an EJB's deployment descriptor specifying the application of aspects to the Account bean.

```
<entity id="Account">
   <ejb-name>Account</ejb-name>
   <home>aspectj2ee.demo.AccountHome</home>
   <remote>aspectj2ee.demo.Account</remote>
   <ejb-class>aspectj2ee.demo.AccountBean</ejb-class>
   <apply>
      <aspect>aspectj2ee.core.Lifecycle</aspect>
   </apply>
   <apply>
      <aspect>aspectj2ee.core.Persistence</aspect>
      <parameter name="primKeyClass">java.lang.String</parameter>
      <parameter name="primKeyField">serialNumber</parameter>
      <parameter name="table">ACCOUNTS</parameter>
      <parameter name="fieldMap">serialNumber:SERIAL, balance:BALANCE</parameter>
   </apply>
   <apply>
      <aspect>aspectj2ee.core.Security</aspect>
      <parameter name="secured">execution(*(..))</parameter>
      <parameter name="requiredRole">User</parameter>
   </apply>
   <apply>
      <aspect>aspectj2ee.core.Transactions</aspect>
      <parameter name="reentrant">false</parameter>
      <parameter name="requiresnew">execution(deposit(..)) ||
                                 execution(withdraw(..))</parameter>
      <parameter name="required">execution(*(..)) && !requiresnew()</parameter>
   </apply>
</entity>
```

The figure shows the XML element describing bean Account. (In general, the deployment descriptor contains such entities for each of the beans, along with other information.) We follow the J2EE convention, in that the bean is defined by the `<entity>` XML element.

Element `<entity>` has several internal elements. The first four: `<ejb-name>`, `<home>`, `<remote>`, and `<ejb-class>`, specify the Java names that make this bean. These are part of J2EE and will not concern us here.

Following are elements of type `<apply>`, which are an AspectJ2EE extension. Each of these specifies an application of an aspect to the bean.

An `<apply>` element has two kinds of internal elements:

1. *`<aspect>`*, naming the aspect to apply to the bean.
   For example, element `<aspect>aspectj2ee.core.Lifecycle</aspect>` in the figure specifies that `aspectj2ee.core.Lifecycle` is applied to Account.

2. *<parameter>*, specifying the configuration parameters passed to the aspect.
   Consider for example the `Security` aspect (Fig. 6.1). In Fig. 6.2, we see that the actual value for the `secured` pointcut formal parameter is **`execution`**`(*(..))` (i.e., the execution of any method). Similarly, formal string parameter `requiredRole` was actualized with value *"User"*.
   Thus, the third `<apply>` element is tantamount to configuring the aspect with the following pseudo-syntax: `Security[`**`execution`**`(*(..)),` *"User"*`]`.

In support of *explicit and flexible ordering*, the order of `<apply>` elements specifies the order by which aspects are applied to the bean. Intra-aspect precedence (where several advices from the same aspect apply to a single join point) is handled as in ASPECTJ, i.e., by order of appearance of advices.

We can generalize the example above to write the entire sequence of application of aspects to the bean, along with their parameters. In total, there are nine such parameters. These, together with the aspect names, would have made the programming language equivalent of the application sequence in Fig. 6.2 cumbersome and error-prone. We found that the XML notation is a convenient replacement to developing syntax for dealing with this unwieldiness.

Note that ACCOUNT can be viewed as an entity bean with container-managed persistence (CMP EJB) simply because it relies on the core persistence aspect, which parallels the standard J2EE persistence service. Should the developer decide to use a different persistence technique, that persistence system would itself be defined as an ASPECTJ2EE aspect, and applied to ACCOUNT in the same manner. This is parallel to bean-managed persistence beans (BMP EJBs) in the sense that the persistence logic is provided by the application programmer, independent of the services offered by the application server. However, it is completely unlike BMP EJBs in that the persistence code would not be tangled with the business logic and scattered across several bean and utility classes. In this respect, ASPECTJ2EE completely dissolves the distinction between BMP and CMP entity beans.

6.3 Implementing Advice by Sub-Classing

ASPECTJ2EE supports each of the join point kinds defined in ASPECTJ, except for **`call`**, since **`call`** advice is applied at the *client* (caller) site and not to the main class. We next describe advice are woven into the entity bean code in each supported kind of join point.

Execution Join Points. The **`execution`**`(`*`methodSignature`*`)` join point is defined when a method is invoked and control transfers to the target method. ASPECTJ2EE captures **`execution`** join points by generating advised methods in the advised class, overriding the inherited methods that match the execution join point. Consider for example the advice in Fig. 6.3(a), whose pointcut refers to the execution of the `deposit()` method. This is a **`before`**`()` advice which prepends a printout line to matched join points. When applied to ACCOUNT, only one join point, the execution of `deposit()`, will match the specified pointcut. Hence, in the advised class, the `deposit()` method will be overridden, and the advice code will be inserted prior to invoking the original code. The resulting implementation of `deposit()` in the advised class appears in Fig. 6.3(b).

Fig. 6.3 **(a)** Sample advice for `deposit()` execution, and **(b)** the resulting advised method.

(a)
```
before(float amount): execution(deposit(float)) && args(amount) {
       System.out.println("Depositing " + amount);
   }
```

(b)
```
void deposit(float amount) {
       System.out.println("Depositing " + amount);
       super.deposit(amount);
   }
```

Recall that only instances of the terminal advised class exist in the system, so every call to the advised method (`deposit()` in this example) would be intercepted by means of regular polymorphism. Overriding and refinement can be used to implement **`before()`**, **`after()`** (including **`after() returning`** and **`after() throwing`**), and **`around()`** advice. With **`around()`** advice, the **`proceed`** keyword would indicate the location of the call to the inherited implementation.

The example in Fig. 6.4 demonstrates the support for **`after() throwing`** advice. The advice, listed in part (a) of the figure, would generate a printout if the `withdraw()` method resulted in an `InsufficientFundsException`. The exception itself is re-thrown, i.e., the advice does not swallow it. The resulting advised method appears in part (b) of the figure. It shows how **`after() throwing`** advice are implemented by encapsulating the original implementation in a **`try`**/**`catch`** block.

Fig. 6.4 **(a)** Sample **`after() throwing`** advice, applied to a method execution join point, and **(b)** the resulting advised method.

(a)
```
after() throwing (InsufficientFundsException ex)
   throws InsufficientFundsException:
   execution(withdraw(..)) {
       System.out.println("Withdrawal failed: " + ex.getMessage());
       throw ex;
   }
```

(b)
```
void withdraw(float amount) throws InsufficientFundsException {
       try { super.withdraw(amount);
       } catch (InsufficientFundsException ex) {
          System.out.println("Withdrawal failed: " + ex.getMessage());
          throw ex;
       }
   }
```

An **`execution`** join point may refer to **`private`** methods. Since such methods cannot be overridden in subclasses, the ASPECTJ2EE weaver generates a new, advised version of the method—and then overrides any method that *invokes* the private method, so that the callers will use the newly-generated version of the private callee rather than the original. The overriding version of the callers includes a complete re-implementation of each caller's code, rather than using refinement, so that only the new version of the callee will be used. The only exception is where a private method is invoked by a constructor, which cannot be replaced by an overriding version. ASPECTJ2EE will issue a warning in such cases.

This technique is used not only with **`execution`** join points, but whenever an advice apples to code inside a **`private`** method (e.g., when a field access join point is matched by code inside one).

A similar problem occurs with **`final`** and **`static`** methods. However, such methods are disallowed by the J2EE specification and may not be included in EJB classes.

Constructor Execution Join Points. The constructor execution join point in ASPECTJ is defined using the same keyword as regular method execution. The difference lies in the method signature, which uses the keyword **`new`** to indicate the class's constructor. For example, the pointcut `execution(new(..))` would match the execution of any constructor in the target class.

Unlike regular methods, constructors are limited with regard to the location in the code where the inherited implementation (`super()`) must be invoked. The invocation must be the first statement of the constructor, and in particular it must occur before any field access or virtual method invocation. Hence, join points that refer to constructor signatures can be advised, but any code that executes before the inherited constructor (`before()` advice, or parts of `around()` advice that appear prior to the invocation of `proceed()`) is invalid.

An `around()` advice for constructor execution that does not contain an invocation of `proceed()` would be the equivalent of a JAVA constructor that does not invoke `super()` (the inherited constructor). This is tantamount to having an implicit call to `super()`.

Field Read and Write Access Join Points. Field access join points match references to and assignments of fields. ASPECTJ2EE presents no limitations on advice that can be applied to these join points. However, if a field is visible *outside* of the class (e.g., a **`public`** field), then any *external* access will bypass the advice. It is therefore recommended that field access will be restricted to **`private`** fields and EJB *attributes* only. Recall that attributes are not declared as fields; rather, they are indicated by the programmer using **`abstract`** getter and setter methods. These methods are then implemented in the concrete bean class (in J2EE) or in the root advised class (in ASPECTJ2EE).

If no advice is provided for a given attribute's read or write access, the respective method implementation in the root advised class would simply read or update the class field. The field itself is also defined in the root advised class. However, an attribute can be advised using `before()`, `around()` and `after()` advice, which would affect the way the getter and setter method are implemented.

If an advice is applied to a field (which is not an attribute), all references to this field by method in the class itself are advised by generating overriding versions of these methods. However, since a **`private`** field is not visible even to subclasses, this might require generating a new version of the field, which *hides* [32, Sect. 8.3.3] the original declaration. In such cases, any method that accesses the field must be regenerated, even where the advice does not cause any code alteration, so that the overriding version will access the new field.

Exception Handler Join Points. The **`handler`** join point can be used to introduce advice into **`catch`** blocks for specific exception types. Since the **`catch`** block per-se cannot be overridden, advising such a join point results in a new, overriding version of the entire advised method. Most of the code remains unchanged from the original, but the code inside the **`catch`** block is altered in accordance with the advice.

Remote Call Join Points. The **`remotecall`** join point designator is a new keyword introduced in ASPECTJ2EE. Semantically, it is similar to ASPECTJ's **`call`** join point designator, defining a join point at a method invocation site. However, it only applies to remote calls to various methods; local calls are unaffected.

Remote call join points are unique, in that their applied advice does not appear in the advised sub-class. Rather, they are implemented by affecting the stub generated at deploy time for use by EJB clients (such as `_Account_Stub` in Fig. 5.4). For example, the **`around`**`()` advice from Fig. 6.5(a) adds printout code both before and after the remote invocation of `Account.deposit()`. The generated stub class would include a `deposit()` method like the one shown in part (b) of that figure. Since the advised code appears in the stub, rather than in a server-side class, the output in this example will be generated by the client program.

Fig. 6.5 **(a)** Sample advice for a method's **`remotecall`** join point, and **(b)** the resulting `deposit()` method generated in the RMI stub class.

```
(a) around(): remotecall(deposit(..)) {
        System.out.println("About to perform transaction.");
        proceed();
        System.out.println("Transaction completed.");
    }

(b) public void deposit(float arg0) {
        System.out.println("About to perform transaction.");
        // ... normal RMI/IIOP method invocation code ...
        System.out.println("Transaction completed.");
    }
```

Remote call join points can only refer to methods that are defined in the bean's remote interface. Advice using **`remotecall`** can be used to localize tier-cutting concerns, as detailed in Sec. 7.

Control-Flow Based Pointcuts. ASPECTJ includes two special keywords, **`cflow`** and **`cflowbelow`**, for specifying control-flow based limitations on pointcuts. Such limitations are used, for example, to prevent recursive application of advice. Both keywords are supported by ASPECTJ2EE.

The manner in which control-flow limitations are enforced relies on the fact that deployment can be done in a completely platform-specific manner, since at deploy time, the exact target platform (JVM implementation) is known. Different JVMs use different schemes for storing a stack snapshot in instances of the `java.lang.Throwable` class[27] (this information is used, for example, by the method `java.lang.Exception.printStackTrace()`). Such a stack snapshot (obtained via an instance

[27] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Throwable.html

of `Throwable`, or any other JVM-specific means) can be examined in order to test for **`cflow`**/**`cflowbelow`** conditions at runtime.

An alternative implementation scheme relies on `ThreadLocal`[28] objects. The advice application would result in a `ThreadLocal` flag that will be turned on or off as various methods are entered or exited. At the target join point, the flag's value will be examined to determine if the **`cflowbelow`** condition holds, and the advice should be executed. The one-instance-per-thread nature of `ThreadLocal` objects ensures that this flag-based scheme will function properly in multi-threaded application.

6.4 The Core Aspects Library

ASPECTJ2EE's definition includes a standard library of core aspects. Four of these aspects were used in the ACCOUNT example, as shown in Fig. 5.4. Here is a brief overview of these four, and their effect on the advised classes:

1. The `aspectj2ee.core.Lifecycle` aspect (used to generated the root advised class) provides a default implementation to the J2EE lifecycle methods. The implementations of `setEntityContext()`, `unsetEntityContext`, and `getEntityContext()` maintain the entity context object; all other methods have an empty implementation. These easily-available common defaults make the development of EJBs somewhat easier (compared to standard J2EE development); the user-provided `AccountBean` class is now shorter, and contains strictly business logic methods[29].
2. The `aspectj2ee.core.Persistence` aspect provides a CMP-like persistence service. The attribute-to-database mapping properties are detailed in the parameters passed to this aspect in the deployment descriptor. This aspect advises some of the lifecycle methods, as well as the attribute setters (for maintaining a "dirty" flag), hence these methods are all overridden in the advised class.
3. The `aspectj2ee.core.Security` aspect can be used to limit the access to various methods based on user authentication. This is a generic security solution, on par with the standard J2EE security service. More detailed security decisions, such as role-based variations on method behavior, can be defined using project-specific aspects without tangling security-related code with the functional concern code.
4. Finally, the `aspectj2ee.core.Transactions` aspect is used to provide transaction management capabilities to all business-logic methods. The parameters passed to it dictate what transactional behavior will be applied to each method. Transactional behaviors supported by the J2EE platform include methods that must execute within a transaction context, and will create a new transaction if none exists; methods that must execute within an existing transaction context; methods that are neutral to the existence of a transaction context; and methods that will fail to run within a transaction context. The list of methods that belong to each group is specified with a pointcut parameter passed to this aspect.

[28] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ThreadLocal.html

[29] The fact that the fields used to implement the attributes, and the concrete getter and setter method for these attributes, appear in `AdvAccount_Lifecycle` (in Fig. 5.4) stems from the fact that this is the root advised class, and is not related to the `Lifecycle` aspect per se.

7 Innovative Uses for AOP in Multi-Tier Applications

The use of aspects in multi-tier enterprise applications can reduce the amount of cross-cutting concerns and tangled code. As discussed in Sec. 3, the core J2EE aspects were shown to be highly effective to this end, and the ability to define additional aspects (as well as alternative implementations to existing ones) increases this effectiveness and enables better program modularization.

But AspectJ2EE also allows developers to confront a different kind of cross-cutting non-functional concerns: aspects of the software that are implemented in part on the client and in part on the server. Here, the cross-cutting is extremely acute as the concern is implemented not just across several classes and modules, but literally across programs. We call these *tier-cutting concerns*. In the context of AspectJ2EE, tier-cutting concerns are applied to the business methods of EJBs.

The notion of remote pointcuts was independently discovered by Nishizawa, Chiba, and Tatsubori [33].

The remainder of this section shows that a number of several key tier-cutting concerns can be represented as single aspect by using the **`remotecall`** join point designator. In each of these examples, the client code is unaffected; it is the RMI stub, which acts as a proxy for the remote object, which is being modified.

7.1 Client-Side Checking of Preconditions

Method preconditions [34] are commonly presented as a natural candidate for non-functional concerns being expressed cleanly and neatly in aspects. This allows preconditions to be specified without littering the core program, and further allows precondition testing to be easily disabled.

Preconditions should normally be checked at the method execution point, i.e., in the case of multi-tier applications, on the server. However, a precondition defines a contract that binds whoever invokes the method. Hence, by definition, precondition violations can be detected and flagged at the invocation point, i.e., on the client. In a normal program, this matters very little; but in a multi-tier application, trapping failed preconditions on the client can prevent the round-trip of a remote method invocation, which incurs a heavy overhead (including communications, parameter marshaling and un-marshaling, etc.).

Fig. 7.1 presents a simple precondition that can be applied to the Account EJB: neither `withdraw()` nor `deposit()` are ever supposed to be called with a non-positive amount as a parameter. If such an occurrence is detected, a `PreconditionFailed-Exception` is thrown. Using two named pointcut definitions, the test is applied both at the client and at the server.

In addition to providing a degree of safety, such aspects decrease the server load by blocking futile invocation attempts. In a trusted computing environment, if the preconditioned methods are invoked only by clients (and never by other server-side methods), the server load can be further reduced by completely disabling server-side tests.

When using aspects to implement preconditions, always bear in mind that preconditions test for logically flawed states, rather than states that are unacceptable from a

Fig. 7.1 An aspect that can be used to apply precondition testing (both client- and server-side) to the ACCOUNT EJB.

```
public aspect EnsurePositiveAmounts {
   pointcut clientSide(float amount):
      (remotecall(public void deposit(float)) ||
       remotecall(public void withdraw(float))) && args(amount);

   pointcut serverSide(float amount):
      (execution(public void deposit(float)) ||
       execution(public void withdraw(float))) && args(amount);

   before(float amount): clientSide(amount) || serverSide(amount) {
      if (amount <= 0.0)
         throw new PreconditionFailedException("Non-positive amount: "+amount);
   }
}
```

business process point of view. Thus, preventing the withdrawal of excessive amounts should be part of `withdraw()`'s implementation rather than a precondition.

7.2 Symmetrical Data Processing

By adding code both at the sending and receiving ends of remotely-invoked methods, we are able to create what can be viewed as an additional layer in the communication stack. For example, we can add encryption at the stub and decryption at the remote tie; or we can apply a compression scheme (compressing information at the sender, decompressing it at the receiver); and so forth.

Consider an EJB representing a university course, with the method `register()` accepting a `Vector` of names of students (`String`s) to be registered to that course. The aspect in Fig. 7.2 shows how the remote invocation of this method can be made more effective by applying compression. Assume that the class `CompressedVector` represents a `Vector` in a compressed (space-efficient) manner. Applying this aspect to the COURSE EJB would result in a new method, `registerCompressed()`, added to the advised class. Unlike most non-public methods, this one would be represented in the class's RMI stub, since it is invoked by code that is included in the stub itself (that code would reside in the advised stub for the `register()` method).

Fig. 7.2 An aspect that can be used for sending a compressed version of an argument over the communications line, when applied to the COURSE EJB.

```
public aspect CompressRegistrationList {
   around(Vector v): remotecall(public void register(Vector)) && args(v) {
      CompressedVector cv = new CompressedVector(v);
      registerCompressed(cv);
   }

   private void registerCompressed(CompressedVector cv) {
      Vector v = cv.decompress();
      register(v);
   }
}
```

Compression and encryption can be applied not only for arguments, but also for return values. In this case, the aspect should use **`after() returning`** advice for both the **`remotecall`** and **`execution`** join points. Advice for **`after() throwing`** can be used for processing exceptions (which are often information-laden, due to the embedded call stack, and would hence benefit greatly from compression).

7.3 Memoization

Memoization (the practice of caching method results) is another classic use for aspects. When applied to a multi-tier application, this should be done with care, since in many cases the client tier has no way to know when the cached data becomes stale and should be replaced. Still, it is often both possible and practical, and using ASPECTJ2EE it can be done without changing any part of the client program.

For example, consider a session EJB that reports international currency exchange rates. These rates are changed on a daily basis; for the sake of simplicity, assume that they are changed every midnight. The aspect presented in Fig. 7.3 can be used to enable client-side caching of rates.

Fig. 7.3 An aspect for caching results from a currency exchange-rates EJB.

```
public aspect CacheExchangeRates {
   private static class CacheData { int year; int dayOfYear; float value; }

   private Hashtable cache = new Hashtable();

   pointcut clientSide(String currencyName):
      remotecall(public float getExchangeRate(String)) && args(currencyName);

   around(String currencyName): clientSide(currencyName) {
      Calendar now = Calendar.getInstance();
      int currentYear = now.get(Calendar.YEAR);
      int currentDayOfYear = now.get(Calendar.DAY_OF_YEAR);

      // First, try and find the value in the cache
      CacheData cacheData = (CacheData) cache.get(currencyName);
      if (cacheData != null) && currentYear = cacheData.year &&
         currentDayOfYear == cacheData.dayOfYear)
            return cacheData.value; // Value is valid; no remote invocation

      float result = proceed(currencyName); // Normally obtain the value

      // Cache the value for future reference
      cacheData = new CacheData();   cacheData.year = currentYear;
      cacheData.dayOfYear = currentDayOfYear; cacheData.value = result;
      cache.put(currencyName, cacheData);
   }
}
```

8 Conclusions, Open Questions, and Future Work

Shakeins are a novel, aspect-like, programming construct, with three distinguishing characterization: explicit application semantics, configuration parameters, and restriction on the changes to a class to be re-implementation only. We argued that the shakein construct integrates well with the object model, by distinguishing the five facets of a class: type, forge, implementation, mold and mill, and explaining how these can be modified by shakeins. It was shown that if we adher to the principle that no variables of shaked classes are allowed, then the construct can be implemented with current JVMs, and using the existing inheritance model of JAVA.

Shakeins enjoy the advantages of the parameterized notation of mixins, while offering a simple answer to the accidental overriding problem. Thanks to the pointcuts semantics of aspects, shakeins become more expressive than mixins in the sense that they can "examine" the internals of their target classes. Conversely, thanks to the parameterized semantics and the object model integration, shakeins simplify some of the more subtle issues of aspects, including aspect inheritance, instantiation, and abstract aspects.

As an important application and prime motivation for actual use of this construct, we presented ASPECTJ2EE, an AOP programming language, similar to ASPECTJ, whose aspects have a shakein semantics. ASPECTJ2EE shows that the shakeins semantics integrates well with the current architecture of J2EE servers. The langauge makes it possible to think of existing services as aspects, while unifying the deployment process of J2EE with aspect weaving as in AOP. Also, the langauge shows that the shakeins semantics allows existing services to be configured, and even applied multiple times. Such benefits are not possible with plain aspects.

By using deploy-time weaving, ASPECTJ2EE allows the programmer's code to be advised without being tampered with. Programmers can define methods that will provide business functionality while being oblivious to the various services (transaction management, security, etc.) applied to these methods. (It is the *code*, not the programmers, that is oblivious to the non-functional concerns—an important distinction, no doubt [35].) With the exception of the **`call`**, we showed that all join point kinds can be implemented using deploy-time weaving.

To the existing repertoire of join points ASPECTJ2EE adds a new join point kind, **`remotecall`**. This join point makes it possible to add to the familiar services provided by EJB containers, ASPECTJ2EE aspects can be used to unscatter and untangle tier-cutting concerns, which in many cases can improve an application server's performance. We discussed in particular interesting such services, including client side checking of pre-conditions, symmetrical data processing, and memoization.

In using the shakeins semantics, aspects in ASPECTJ2EE are less general, and have a more defined target, than their ASPECTJ counterparts. Also, even though the same aspect can be applied (possibly with different parameters) to several EJBs, each such application can only affect its specific EJB target. Therefore, we expect ASPECTJ2EE aspects should be more understandable, and the woven programs more maintainable.

We believe that ASPECTJ2EE opens a new world of possibilities to developers of J2EE applications, allowing them to extend, enhance and replace the standard services provided by EJB containers with services of their own. EJB services can be distributed and used across several projects; libraries of services can be defined and reused.

ASPECTJ2EE does not encompass the shakeins semantics in full. In particular, aspect application is external to the language and is specified by an XML deployment descriptor file. The file format is such that shakeins can be applied to EJBs only, and that un-shaked versions of such a bean are not available to clients.

This restriction demonstrates one of the open problems that this new construct opens, namely, *granularity of applicability*. We argued that explicit application of shakeins to classes contributes to the expressive power that programmers may need. We explained why a global, system-wide application of aspects may lead to undesired results.

Still, as evident by the ASPECTJ2EE experience, it is necessary at times to apply shakeins to a large number of classes. We need a mechanism that supports this need. The answer may lie with a syntax and semantics for applying a certain shakein, or even a family of shakeins, to an entire package, or to a class hierarchy.

Shakein also highlight the issue of using a class as a factory of its instances. It should be possible to change the class implementation, as shakeins do, without modifying client code. However, clients that use the class constructors directly to generate instances, resist such changes.

Although there are design patterns (e.g., ABSTRACT FACTORY and FACTORY METHOD), and framework rules (e.g., as in J2EE) for dealing with this predicament, we believe that a more systematic solution is possible. The appendix overviews *factories*, a mechanism which helps client code access only certain re-implementations of a class. Factories provide similar functionality to ASPECTJ2EE, but are implemented at the langauge level, rather than by using external XML files.

Another challenging topic is that of a type system for shakeins, including a type system for making and enforcing constraints on shakein parameters, and typing of shakein composition. Such a type system should deal with the case that some of the configuration parameters are specified at the time of composition.

References

1. Cohen, T., Gil, J.: AspectJ2EE = AOP + J2EE: Towards an aspect based, programmable and extensible middleware framework. In Odersky, M., ed.: Proc. of the Eighteenth European Conference on Object-Oriented Programming (ECOOP'04) 219–243
2. Cohen, T., Gil, J.: Distinguishing class from type with factories and shakeins. Submitted for publication (2006)
3. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proc. of the Eleventh European Conference on Object-Oriented Programming (ECOOP'97) 220–242
4. Soares, S., Laureano, E., Borba, P.: Implementing distribution and persistence aspects with AspectJ. In: Proc. of the Seventeenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02) 174–190
5. Hao, R., Boloni, L., Jun, K., Marinescu, D.C.: An aspect-oriented approach to distributed object security. In: Proc. of the Fourth IEEE Symp. on Computers and Communications
6. Kim, H., Clarke, S.: The relevance of AOP to an applications programmer in an EJB environment. Proc. of the First International Conference on Aspect-Oriented Software Development (AOSD) Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) (2002)
7. Choi, J.P.: Aspect-oriented programming with Enterprise JavaBeans. In: Proc. of the Fourth International Enterprise Distributed Object Computing Conference (EDOC 2000) 252–261
8. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning, Greenwich (2003)
9. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: Proc. of the Fifteenth European Conference on Object-Oriented Programming (ECOOP'01) 327–355
10. Bracha, G., Cook, W.R.: Mixin-based inheritance. In Meyrowitz, N.K., ed.: Proc. of the Fifth Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming OOPSLA/ECOOP'90 303–311

11. Parnas, D.L.: Information distribution aspects of design methodology. In Freiman, C.V., Griffith, J.E., Rosenfeld, J.L., eds.: Proc. of the IFIP Congress 339–44
12. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In Gruia-Catalin Roman, William G. Griswold, B.N., ed.: Proc. of the Twenty Seventh International Conference on Software Engineering (ICSE'05) 49–58
13. Gradecki, J.D., Lesiecki, N.: Mastering AspectJ: Aspect-Oriented Programming in Java. Wiley (2003)
14. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behavior. In Cardelli, L., ed.: Proc. of the Seventeenth European Conference on Object-Oriented Programming (ECOOP'03) 248–274
15. Cardelli, L., Wegner, P.: On understanding types, data abstractions, and polymorphism. ACM Computing Surveys **17** (1985) 471–522
16. Bracha, G.: The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. PhD thesis, Department of Computer Science, University of Utah (1992)
17. Allen, E., Bannet, J., Cartwright, R.: A first-class approach to genericity. [36] 96–114
18. Ancona, D., Lagorio, G., Zucca, E.: Jam — a smooth extension of Java with mixins. In Bertino, E., ed.: Proc. of the Fourteenth European Conference on Object-Oriented Programming (ECOOP'00) 154–178
19. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New-age components for old-fashioned Java. In: Proc. of the Sixteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01) 211–222
20. McDirmid, S., Hsieh, W.C.: Aspect-oriented programming with Jiazzi. [37] 70–79
21. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing series. Addison-Wesley Publishing Company, Reading, Massachusetts (1995)
22. Pichler, R., Ostermann, K., Mezini, M.: On aspectualizing component models. Software - Practice and Experience **33** (2003) 957–974
23. Bodoff, S., Green, D., Haase, K., Jendrock, E., Pawlan, M., Stearns, B.: The J2EE Tutorial. Addison-Wesley Publishing Company, Reading, Massachusetts (2002)
24. Duclos, F., Estublier, J., Morat, P.: Describing and using non functional aspects in component based applications. In: Proc. of the First International Conference on Aspect-Oriented Software Development (AOSD 2002) 22–26
25. Mezini, M., Ostermann, K.: Conquering aspects with Caesar. [37] 90–100
26. Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., Martelli, L.: JAC: an aspect-based distributed dynamic framework. Soft. - Pract. and Exper. **34** (2004) 1119–1148
27. Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Jørgensen, B.N.: Dynamic and selective combination of extensions in component-based applications. In: Proc. of the Twenty Third International Conference on Software Engineering (ICSE'01) 233–242
28. Popovici, A., Gross, T.R., Alonso, G.: Dynamic weaving for aspect-oriented programming. In: Proc. of the First International Conference on Aspect-Oriented Software Development (AOSD'02) 141–147
29. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: an aspect-oriented approach tailored for component based software development. [37] 21–29
30. Walker, R.J., Baniassad, E.L.A., Murphy, G.C.: An initial assessment of aspect-oriented programming. In: Proc. of the Twenty First International Conference on Software Engineering (ICSE'99) 120–130
31. Constantinides, C.A., Elrad, T., Fayad, M.E.: Extending the object model to provide explicit support for crosscutting concerns. Software - Practice and Experience **32** (2002) 703–734
32. Gosling, J., Joy, B., Steele, G.L.J., Bracha, G.: The Java Language Specification. Third edn. Addison-Wesley Publishing Company, Reading, Massachusetts (2005)

33. Nishizawa, M., Chiba, S., Tatsubori, M.: Remote pointcut: a language construct for distributed AOP. In: Proc. of the Third international conference on Aspect-Oriented Software Development (AOSD'04) 7–15
34. Meyer, B.: Object-Oriented Software Construction. Second edn. Prentice-Hall, Englewood Cliffs, New Jersy 07632, Englewood Cliffs, New Jersy (1997)
35. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: OOPSLA 2000 Workshop on Advanced Separation of Concerns. ACM, Minneapolis (2000)
36. Crocker, R., Jr., G.L.S., eds.: Proc. of the Eighteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03), Anaheim, California, USA, ACM SIGPLAN Notices 38 (11) (2003)
37. Proc. of the Second International Conference on Aspect-Oriented Software Development (AOSD'03), Boston, Massachusetts, USA, ACM Press, New York, NY, USA (2003)
38. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. [36] 302–312
39. Grand, M.: Patterns in Java, Volume 1. John Wiley & Sons, USA (1998)

A Beyond ASPECTJ2EE: Factories

The ASPECTJ2EE experiment indicates that the shakeins semantics integrates well with current J2EE server architectures. However, the same experiment also led us to the conclusion that program modules should be provided with a better mechanism for controlling object instantiation. We therefore present *factories* as a new language construct, providing each class with complete control over its own instance-creation process.

Factories are methods which return new class instances. Syntactically, a factory is a method which overloads the **new** operator with respect to a certain class. The language extension requires no changes to the JVM.

The need for factories stems directly from anomalies that can be found in constructors (in JAVA, C++, etc.). First, constructors are simultaneously static and dynamic: static, since they are invocable without an instance; and dynamic, since they work on a specific object. Second, it is mundane to see that constructors obey a static binding scheme, and it takes just a bit of pondering to understand the difficulties that this scheme brings about. If a class C' inherits from C, then C' should be always substitutable for C. An annoying exception is made by constructor invocation sites in client code; these have to be manually fixed in switching from C to C'.[30]

The confusion between static and dynamic binding penetrates even into the constructor code itself, i.e., into the mill. Method invocation from the mill follows a static binding scheme in C++[31]; in JAVA and C$^{\#}$, however, dynamic binding is used. Neither approach is without fault. Static binding can lead to illegal invocation of pure virtual methods. Dynamic binding prevents methods, invoked from within the mill, from assuming that all fields were properly initialized. This leads, among other things, to difficulties in implementing non-nullable types, as described by Fähndrich and Leino [38]: during construction, fields of non-null types may contain null values.

[30] Interestingly, the Gang of Four [21, p.24] place this predicament first in their list of causes for redesign, saying: "*Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface*".

[31] Even for **virtual** methods.

Unlike constructors, factories can be defined as either static or dynamic.

In studying constructors further, we can identify three steps in instances birth process: (a) *creation*, in which the object's actual type is selected, memory is allocated and structured by the mold; (b) *initialization*, in which fields are set to their initial values; and (c) *setup*, in which the mill is executed.

JAVA (as well as C++, C$^\#$ and other languages) does not provide the programmer with control over the creation step[32].Yet, as the classical creational design patterns (e.g., ABSTRACT FACTORY, FACTORY METHOD, SINGLETON [21], and OBJECT POOL [39]) demonstrate, elaborate software systems require intervention in this step. The use of *home interfaces* in J2EE, or a centralized Factory object in Spring, further underlines this point: since the middleware environment requires control over the object instantiation process, it forces a particular mechanism for obtaining instances, which makes bean client code cumbersome. Creational patterns and home interfaces grant the programmer control over the creation step, by replacing constructor signatures from the forge facet with a different, statically-bound, common method (e.g., `getInstance`).

However, in contrast to most other patterns, the creational patterns cannot be implemented in OO languages without revealing implementation details to the client. If class `T` is implemented as a SINGLETON, then clients of this class cannot write **`new`** `T()` and expect the correct instance to be returned; rather, they must be explicitly aware of the nonstandard creation mechanism. Further, to generalize object instantiation, middleware frameworks often place arguments to the creation mechanism in a configuration file. This extra-lingual resource is a source for potential errors that will only be detected at runtime. Factories enable a clear-cut separation between creation and initialization and setup, and make the use of external configuration files entirely optional.

A.1 Supplier-Side Factories

Class `Demo` in Fig. A.1 realizes the SINGLETON design pattern, by overloading **`new`** with the factory defined in lines 4–8.

Fig. A.1 A Singleton defined using a factory.

```
 1class Demo {
 2  private static Demo instance = null;

 4  public static new() {
 5    if (instance == null)
 6      instance = this();
 7    return instance;
 8  }

10  Demo() {
11    // ... setup code
12  }
13}
```

This factory is invoked whenever the expression **`new`** `Demo()` is evaluated, in class `Demo` or any of its clients. Note that the factory is declared **`static`**, which stresses that it binds statically, and that (unlike constructors) it has no implicit **`this`** parameter. Examining the factory body we see that it always returns the same instance of the class. Thus, the clients cannot know that `Demo` is a singleton, and will not be affected if this implementation decision is changed.

In general, a factory must either return a valid object of the class (instances of subclasses are ok), or throw an exception.[33] A constructor call is required to generate such

[32] Overloading **`new`** in C++ grants the programmer control over memory allocation, but not over the kind of object to be created, nor the decision if an object has to be created at all.

[33] Should the returned expression evaluate to **`null`**, a `NullPointerException` is thrown.

a new object; but simply writing **new** `Demo()` would recurse infinitely. Instead, the factory invokes the class constructor directly (line 6) with the expression **this**`()`.

Like constructors, factories are not inherited. Had class C' inherited a factory **new**`()` from its superclass C, then the expression **new** C'`()` might yield an instance of C, contrary to common sense. (In contrast, the expression **new** C`()` *can* yield an instance of the C', since a subclass is always substitutable for its superclass.)

Existing techniques for controlling the creation step, such as the `getInstance` method of a SINGLETON, are not protected from inheritance. Therefore, if C is an old-fashioned singleton, then the expression C'`.getInstance()` is valid—but returns an instance of C! This happens because `getInstance` is technically part of the type, while conceptually being part of the forge.

A.2 Client-Side Factories

All examples so far defined factories in the same class on which the overload takes place. Factories of this sort are called *supplier-side factories*. It is also possible to define *client-side factories*, as demonstrated in Fig. A.2.

Fig. A.2 A shakein-using factory.

```
1class Bank {
2 public static new Account(Client c) {
3  if (c.hasBadHistory())
4   return new Secure<LimitedAccount>(c);
5   // LimitedAccount is a subclass of Account

7  return new Secure<Account>(c);
8 }
9 // ... rest of the class
10}
```

Line 2 in the figure starts the definition of a factory. Unlike the previous examples, this definition specifies the returned type. The semantics is that the definition overloads **new** when used for creating `Account` objects from within class `Bank`. It is invoked in the evaluation of an expression of the form **new** `Account(c)` (where `c` is of type `Client` or any of its subclasses) in this context. This factory chooses (lines 3–7) an appropriate kind of `Account` depending on the particular business rules used by the enclosing class.

Unlike supplier-side factories, client-side factories *are* inherited by subclasses. Therefore, the factory from Fig. A.2 will also be used for evaluating expressions of the form **new** `Account(c)` in subclasses of `Bank`.

It is easy to see how middleware applications using shakeins can benefit from factories. In Fig. A.2, the client-side factory always applies the `Secure` shakein to the selected type before generating an instance. Thus, any `Account` created by a `Bank` (or a subclass thereof) will be a `Secure`d one. Should the developers desire that a given shakein S must be applied to all instances of some class c, then the supplier-side factory of c can be made to return only instances of $S\langle c\rangle$. The supplier-side factories can choose the particulars of which shakeins to apply based on configuration values set, e.g., in deployment descriptors. This completely alleviates the need for home interfaces for EJBs, while allowing EJB clients to obtain instances as simply as invoking **new**.