AspectJ2EE = AOP + J2EE
Towards an Aspect Based, Programmable and Extensible Middleware Framework

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

Abstract. J2EE is a middleware architecture augmented with supporting tools for developing large scale client/server and multi-tier applications. J2EE uses Enterprise JavaBeans as its component model. The realization of these components by a J2EE application server can be *conceptually* decomposed into distinct aspects such as persistence, transaction management, security, and load balancing. However, current servers do not employ aspect-oriented programming in their implementation. In this paper, we describe a new aspect language, AspectJ2EE, geared towards the generalized implementation of J2EE application servers, and applications within this framework. AspectJ2EE can be easily employed to extend the fixed set of services that these servers provide with new services such as logging and performance monitoring. Even *tier-cutting concerns* like encryption, data compression, and memoization can be added while avoiding the drags of cross-cutting and scattered code.

AspectJ2EE is less general (and hence less complicated) than AspectJ, yet demonstrably powerful enough for the systematic development of large scale (and distributed) applications. The introduction of *parameterized aspects* makes aspects in AspectJ2EE more flexible and reusable than aspects in AspectJ.

AspectJ2EE also generalizes the process of binding services to user applications in the application server into a novel *deploy-time* weaving of 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.

1 Introduction

The term *enterprise applications* is used to describe the large-scale software programs used to operate and manage large organizations. The world's largest and most important software systems are enterprise applications; this includes the programs used to run government organizations, banks, insurance companies, financial institutes, hospitals, and so forth. Enterprise applications make the world go around.

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

In many cases, enterprise applications are based on a heterogeneous platform configuration, 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.

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, unyielding security, and complete trustworthiness.

The staggering demand for rapid development of enterprise applications initiated a series of component-based *middleware architectures*. 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 component model.

Aspect-oriented programming (AOP) [3], the methodology which encapsulates the code relevant to any distinct non-functional concern in *aspect* modules, can also be thought of as answering the same demand [4, 5]. As it turns out, 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 here, 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.

A natural quest then 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]).

The new approach and main contribution of this paper is in drawing from the lessons of J2EE and its implementation to design a new AOP language, AspectJ2EE, geared towards the generalized implementation of J2EE application servers and applications within this framework. In particular, AspectJ2EE generalizes the process of binding services to user applications in the J2EE application server into a novel *deploy-time* weaving mechanism. 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.

As a consequence of its particular weaving method, and of staying away from specialized JVMs and bytecode manipulation for aspect-weaving, AspectJ2EE is similar to, yet (slightly) less general than, the *AspectJ* programming language [8]. Nevertheless, standing on the shoulders of the J2EE experience, we can argue that AspectJ2EE is highly suited to systematic development of enterprise applications. Perhaps the main limitation of AspectJ2EE when compared to ApsectJ is that it does not directly support *field* read and write join points, and hence cannot be employed for low-level debugging or nit-picking logging. If however the design of a software solution is such that the management of a certain field can be decomposed into several aspects, then this field can be realized as a J2EE *attribute*, with join points at its retrieval and setting.

The semantic model of applying an aspect to a class in AspectJ2EE is shown to be conceptually similar to the application of a generic type definition to a class, yielding a new type. This has both theoretical and practical implications, since maintaining the standard object model makes AspectJ2EE easier to understand and master, a crucial consideration for the widespread adoption of any new technology in the field of enterprise application development[1]. Despite the similarities, we show that AspectJ2EE aspects are more flexible and expressive than generics when used to extend existing types.

AspectJ2EE also introduces *parameterized aspects*. These constructs, combined with AspectJ2EE's aspect binding language, make aspects in AspectJ2EE more reusable than AspectJ aspects.

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, and object- and class-initialization.

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. 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.

Terminology. The article assumes basic familiarity with standard AOP terms, 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).

Outline. Section 2 makes the case for AspectJ2EE by explaining in greater detail how J2EE services can be thought of as aspects. Discussing the benefits of using AOP for these services, we present the main points in which the design of AspectJ2EE is different than standard AOP. The deploy time weaving strategy is discussed in Sect. 3. Section 4 shows how the AspectJ2EE approach introduces AOP into the OOP model without breaking it. Section 5 introduces some of the fine points and innovations in the language, and discusses implementation details. Section 6 lists several possible innovative uses for AspectJ2EE, some of which can lead to substantial performance benefits. Section 7 concludes.

[1] 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 [9], that other sectors of the software industry view as thoroughly outdated.

2 The Case for AOP in J2EE

2.1 J2EE Services as Managers of Non-Functional Concerns

Ideally, with the J2EE middleware 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* [10, Chap. 2]. Again, these issues are none other than *non-functional concerns* in AOP jargon.

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* [2, Chap. 10]. 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*.

And while this work focuses on EJBs, the J2EE design guideline, according to which the developer configures various services via deployment descriptors, is not limited to EJBs only. 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 prime target for being implemented as aspects in AOP-based systems [11, pp. 13–14]. For example, Soares *et. al.* [4] implement distribution, persistence and transaction aspects for software components using AspectJ. 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. Indeed, Kim and Clarke [6] present a case study where they investigate the relevance of AOP to J2EE developers. The case study 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.

The implementation of services in J2EE also includes sophisticated mechanisms for combining each of the services, as configured by the deployment descriptors, with the user code. We will discuss these mechanisms, which can be though of as the equivalent of aspect weaving, in detail below (Sect. 3). Suffice to say at this point that the combination in J2EE is carried out without resorting to drastic means such as byte code patching and code preprocessing—means which may break the object model, confuse debuggers and other language tools, and even obfuscate the semantics.

2.2 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 [12] 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.* [13, 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 a single 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 [14] state: “*clearly, the ‘component’ technology introduced successfully by EJB for managing non-functional aspects reaches its limits*”.

2.3 Marrying J2EE with AOP

Having exposed some of the limitations of J2EE, it is important to stress that the framework enjoys extensive market penetration, commanding a multi-billion dollar market [15].

In contrast, AOP, with its elegant syntax and robust semantics, did not find its place yet in mainstream industrial production. It is only natural then to seek a reconciliation of the two approaches, in producing an aspect based, programmable and extensible middleware framework. Indeed, Pichler *et. al.* call for “a marriage of aspects and components” [12, Sect. 4].

Obviously, each of the services that J2EE provides should be expressed as an aspect. The collection of these services will be the *core aspect library*, which relying on J2EE success, would not only be provably useful, but also highly customizable. Developers will be able to add their own aspects (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 reused 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. The resulting prototype server, called *AES*, allows developers to add and remove aspects from the container, changing the runtime behavior of the system.

Release 4.0 of JBoss [16], an open-source application server which implements the J2EE standard, supports aspects with no language extensions [17]. Aspects are implemented as Java classes which implement a designated interface, while pointcuts are defined in an XML syntax. These can be employed to apply new aspects to existing beans without introducing scattered code. Standard services however are not implemented with this aspect support.

Focal to all this prior work was the attempt to make an existing widespread framework more robust using AOP techniques. In this research, we propose a new approach to the successful marriage of J2EE and AOP in which the design of a new AOP language 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. In OOP terminology these beans are the core classes of the application, each of which represents one component of the underlying data model. 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.
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 enterprise beans. This is carried out entirely within the dominion of object oriented programming, using the standard Java language, and an unmodified Java virtual machine (JVM). 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.* Aspects in AspectJ2EE can contain two types of parameters that accept values at the time of aspect application: abstract pointcut definitions, and field values. Aspects that contain abstract pointcut definitions can be applied to EJBs, by providing (in the EJBs deployment descriptor) a concrete definition for each such pointcut. This provides 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 [18] much-touted separation between aspect implementation and aspect binding. Field values, the other type of aspect parameters, 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 (processing the data at one end and reversing the process at the other). 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.

3 Deployment and Deploy-Time 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 [8], 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 which was employed in at least one aspect-oriented programming language implementation.

As noted before, AspectJ2EE uses its own peculiar *deploy-time weaving* strategy. In this section we motivate this strategy and explain it in greater detail.

3.1 Unbounded Weaving Considered Harmful

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

However, beyond the intricacies of the implementation, weaving introduces a major conceptual bottleneck. As early as 1998, Walker, Baniassad and Murphy [19] noted the disconcert of programmers when realizing that merely reading the source of a code unit is not sufficient for understanding its runtime behavior[2].

3.2 Non-Intrusive Explicit Weaving

The remedy suggested by Constantinides, Bader, and Fayad in their *Aspect Moderator* framework [20] 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.

[2] Further, Laddad [11, 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 *Aspect Mediator* framework, due to Cohen and Hadad [22], ameliorates the problem by simplifying the registration process, and each of the registration instructions. Still, their conclusion is that the explicit weaving code should be generated by an automatic tool from a more concise specification. The AspectJ2EE language processor gives this tool, which generates the explicit registration sequence out of an AspectJ-like weaving specification.

We stress that AspectJ2EE does not use any of the obtrusive weaving strategies listed above. True to the spirit of Aspect Mediator, it employs a weaving strategy that *does not break* the object model. Instead of modifying binaries (directly, or by preprocessing the source code), AspectJ2EE generates new classes that inherit from, rather than replace, the core program classes. Aspect application is carried out by subclassing, during the deployment stage, the classes that contain the business logic.

3.3 J2EE Deployment as a Weaving Process

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 *remote method invocation* (RMI) compiler, `rmic` [23]. Even though some J2EE application servers (e.g., JBoss [16]) generate support class binaries directly (without going through the source), these always conform to the standard object model.

Figure 1 compares the development cycle of traditional and J2EE application. We see in the figure 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.

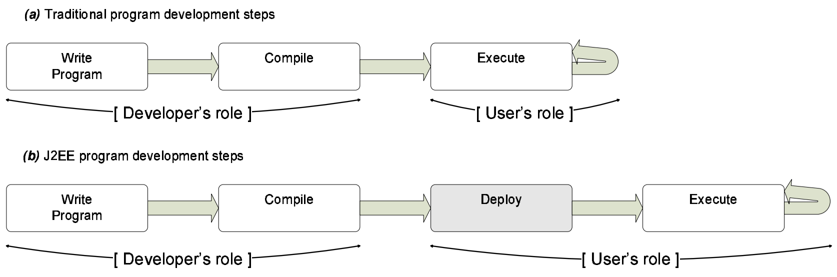

Fig. 1. (a) the development steps in a traditional application, **(b)** the development steps in a J2EE application.

Deployment is the magic by which J2EE services are welded to applications. Therefore, the generation of sub- and support classes is governed by deployment descriptors.

The idea behind deploy-time weaving is to extend this magic, by placing rich AOP semantics in government of this process. Naturally, this extension also complicates the structure and inter-relationships between the generated support classes.

To better understand plain deployment, consider first Figure 2, which shows the initial hierarchy associated with an ACCOUNT CMP bean. This bean will serve as a running example for the rest of this article. While technically, it is defined as a CMP entity EJB, we shall see later that the use of AspectJ2EE completely blurs the lines between CMP and BMP entity beans, and between entity beans in general and session beans (both stateful and stateless). The nature of each bean is derived simply from the aspects that are applied to it.

Interface `Account` is written by the developer in support of the remote interface to the bean[3]. This is where client-accessible methods are declared.

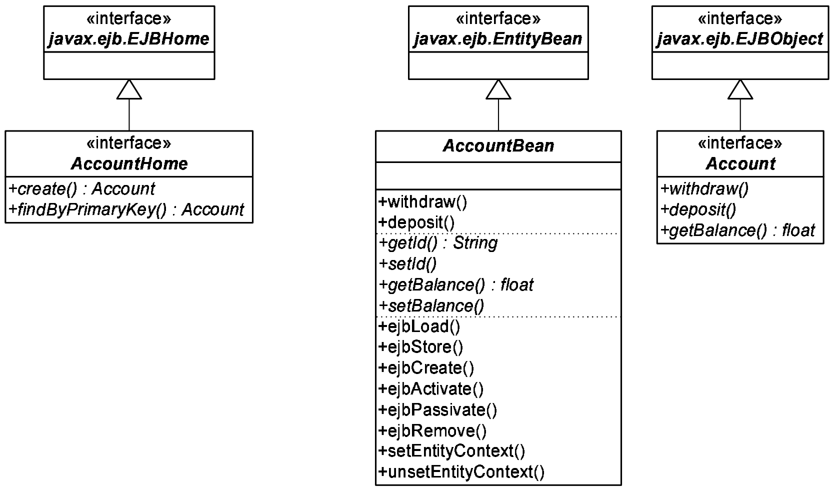

Fig. 2. Classes created by the programmer for defining the ACCOUNT EJB.

The developer's main effort is in coding the abstract class `AccountBean`. The first group of methods in this class consists the implementation of business logic methods (`deposit()` and `withdraw()` in the example).

In addition to regular fields, an EJB has *attributes*, which are fields that will be governed by the persistence service in the J2EE server. Each attribute `*attr*` is represented by abstract setter and getter methods, called `set*Attr*()` and `get*Attr*()` respectively. Attributes are not necessarily client accessible. By examining the second group of methods in this class, we see that ACCOUNT has two attributes: `id` (the primary key)

[3] For the sake of simplicity, we assume that ACCOUNT has a remote interface only, even though since version 2.0 of the EJB specification [2], beans can have either a local interface, a remote interface, or both.

and `balance`. From the `Account` interface we learn that `id` is invisible to the client, while `balance` is read-only accessible.

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 when the CMP service is used. 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.

Interface `AccountHome` 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. Figure 3 shows some of the classes generated by IBM's WebSphere Application Server (WAS) [24] version 5.0 when deploying this bean.

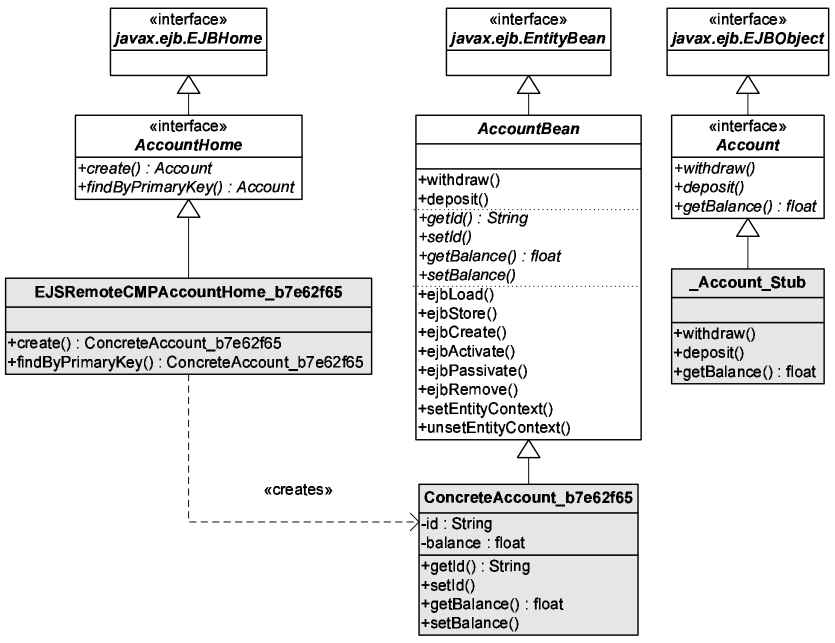

Fig. 3. UML diagram of the ACCOUNT EJB classes defined by the programmer, and a partial set of the support classes (in gray) generated by WebSphere Application Server during the deployment stage.

`ConcreteAccount_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 `EJSRemoteCMPAccountHome_-b7e62f65`, which implements the factory interface `AccountHome`. Finally, _Account_Stub is a COBRA-compliant stub class to the bean, to be used by the bean's clients.

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 various services that the EJB 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.

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, 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 [2,10] on what programmers must, and must not, do while defining EJBs. These dictations are that attributes must be represented by abstract getter and setter methods, rather than by a standard Java class member; 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 An OOP-Compliant Implementation of AOP

Having described deployment as a weaving process, we are ready to explain how AspectJ2EE is implemented without breaking the object model.

Figure 4 shows how the application of four aspects to the ACCOUNT bean is realized. Comparing the figure to Fig. 3 we see that the definition of class AccountBean is simplified by moving the lifecycle methods to a newly defined class, AdvAccount_Lifecycle. In AspectJ2EE the programmer is not required to repeatedly write token implementations of the lifecycle methods in each bean. Instead, these implementations are packaged together in a standard Lifecycle aspect. Class AdvAccount_Lifecycle realizes the application of this aspect to our bean.

In general, for each application of an aspect to a class the deploy tool generates an *advised class*, so called since its generation is governed by the advices given in the aspect. There are three other advised classes in the figure: AdvAccount_Persistence, AdvAccount_Security and AdvAccount_Transactions, which correspond to the application of aspects Persistence, Security and Transactions to ACCOUNT.

The sequence of aspect applications is translated into a chain of inheritance 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.

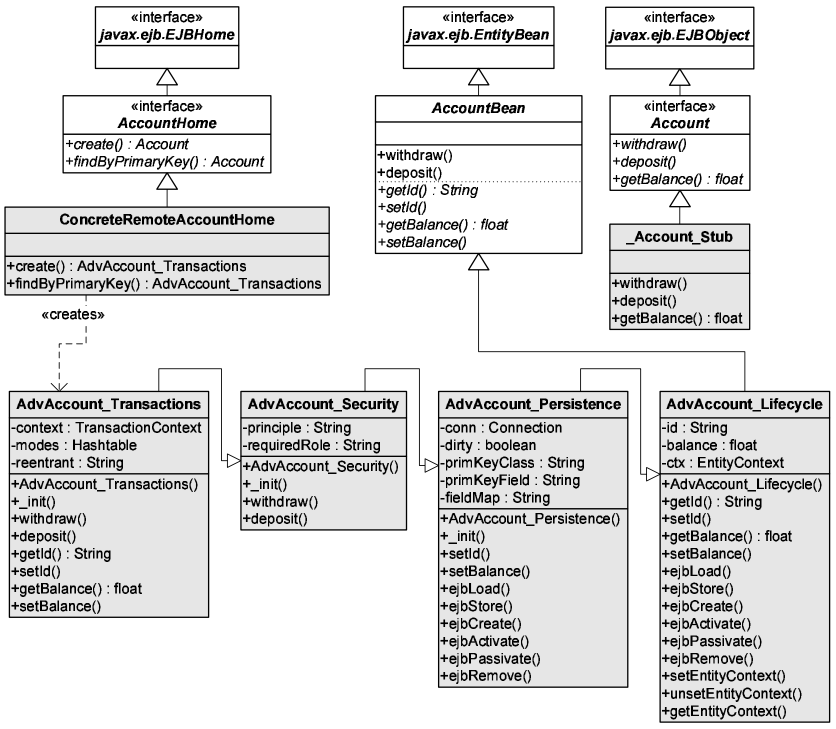

Fig. 4. The class hierarchy of bean ACCOUNT including programmer defined classes and interfaces and the support classes (in gray) generated by the AspectJ2EE deployment tool.

The restriction of aspect targets to classes is one of the key features of AspectJ2EE which made it possible to reify *aspect application* as a class. In contrast, AspectJ is inclined to reify each *aspect* as a class, with rich and somewhat confusing semantics of instantiation controlled by a list of dedicated keywords (**`perthis`**, **`pertarget`**, **`percflow`**, **`percflowbelow`** and **`issingleton`**).

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`. 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.

4.1 Aspects as Mixins and Generics

The AspectJ2EE approach draws power from being similar in concept to familiar mechanisms such as generics. In this interpretation, aspect `Persistence` is a generic class definition. The application of an aspect to a class is modelled then as the application of the corresponding generic definition to that class, yielding a concrete, instantiable class. Thus, class `Lifecycle<AccountBean>` is the conceptual equivalent of applying aspect `Lifecycle` to `AccountBean`, class

```
Persistence<Lifecycle<AccountBean>>
```

corresponds to the application of aspect `Persistence` to the result, etc.

The generic declaration of an aspect `Aspect` would be written, in JDK 1.5-like[4] syntax, as

```
class Aspect<bean B> extends B { /* ...*/ }.        (1)
```

The form (1) allows us to draw parallels between the implementation of aspects in AspectJ2EE and another very familiar programming construct, namely *mixins* [25]. Curiously, Kiczales *et. al.* mention [8, Sect. 7.4] that mixins can be implemented with aspects. AspectJ2EE in effect shows the converse, namely that aspects can be implemented using a mixin-like mechanism, where the parallel drawn between **`before`**`()`, **`after`**`()` and **`around`**`()` advices and variations of overriding: "before" and "after" demons, and plain refining.

Interestingly, the programming pattern (1) was one of the prime motivations in the development of MIXGEN [26], a next-generation implementation of generics for Java and other OO languages. In extrapolating the evolution of generics mechanisms we see it supporting our embedding of aspects in the object model, while preserving the solid formal foundation.

4.2 Inheritance of Aspects, Abstract and Parameterized Aspects

Despite the similarities, we note that AspectJ2EE aspects are more flexible and expressive than mixins and generics. The main difference is that the body of an aspect (the "`/* ... */`" in (1)) may contain a pointcut definition, which may specify e.g., that a single advice applies to a range. In contrast, generics and mixins implementations do not allow specialization based on actual parameter.

We therefore rely on the form (1) as a conceptual model, which should help in understanding the semantics of AspectJ2EE, rather than any means for syntax definition.

[4] Note that the Java SDK version 1.5 does not actually support this construct; the type parameter cannot serve as the superclass of the generic type.

This form is beneficial for example in modelling aspect inheritance. An aspect `A1` inheriting from an aspect `A2` is simply written as

$$\texttt{\textbf{class} A1<B> \textbf{extends} A2<B> \{ /* \ldots */ \}}. \tag{2}$$

Furthermore, with this perspective we can easily distinguish between the two kinds of **`abstract`** aspects of AspectJ. Abstractness due to abstract methods is modelled by prefixing the class definition (1) with keyword **`abstract`**. The more interesting case of abstractness is when a pointcut is declared but not defined. In AspectJ, abstract aspects of this kind must be made concrete by way of inheritance before they are applied. In contrast, AspectJ2EE coins the term *parameterized aspects* for these, and allows missing pointcut definitions to be provided at application time, as modelled by the following form

$$\texttt{\textbf{class} Aspect<B,}\mathcal{P}_1,\ldots,\mathcal{P}_k\texttt{> \textbf{extends} B \{ /* \ldots */ \}} \tag{3}$$

where each $\mathcal{P}_i$, $i = 1, \ldots, k$, is a formal parameter to the aspect representing an abstract pointcut.

Parameterized aspects are similar to Caesar's *passive* pointcuts and advice [12], providing a separation between aspect implementation and aspect binding and hence enjoy similar reusability benefits. A typical example is that of a transaction management aspect with an **`abstract`** pointcut but specific advice for each of the transactional modes of methods. Such modes in the J2EE standard are `required` (method must execute within a database transaction; if no transaction exists, start one), `requiresnew` (the method must execute within a new transaction), etc.

The most crucial advantage of our approach over Caesar's is that in the sake of combat against scattered and tangled code problems, we forbid invocation and binding of aspects at runtime.

Parameterized aspects are not limited to abstract pointcut definitions. An abstract aspect can also include what can be thought of as an "abstract field"—a field whose initial value is specified at application time, as modelled by the form

$$\texttt{\textbf{class} Aspect<B,}\mathcal{P}_1,\ldots,\mathcal{P}_k,\mathcal{V}_1,\ldots,\mathcal{V}_n\texttt{> \textbf{extends} B \{/* \ldots */\}} \tag{4}$$

where each $\mathcal{V}_i$, $i = 1, \ldots, n$, is a formal parameter to the aspect representing an abstract field.

The form (4) makes it possible to apply the same aspect more than once to a single bean class, with each repeated application providing a distinct new extension. For example, consider a parameterized security aspect that accepts two parameters: a pointcut definition, specifying in which join points in the class should security checks be placed; and a field value, specifying the role to require at each of these join points. A single application of this aspect to the bean ACCOUNT could look like this:

$$\texttt{Security<Account,}P_{\text{teller}}\texttt{,"teller">}$$

where P_{teller} is a concrete pointcut definition specifying the execution of methods that require teller authorization. An additional application of the same aspect to the bean can then be used to specify which methods require client authoriation:

$$\texttt{Security<Security<Account,}P_{\text{teller}}\texttt{,"teller">,}P_{\text{client}}\texttt{,"client">}$$

where P_{client} is also a concrete pointcut definition.

5 The AspectJ2EE Programming Language

In this section we describe the implementation details of AspectJ2EE, and in particular how aspect application is described and how advice weaving is accomplished by way of subclassing.

5.1 The Aspect Binding Specification Language

One of the key issues in the design of an AOP language is the binding of aspects to the core functional code. This binding information includes the specification of the list of aspects which apply to each software module, their order of application, and even parameters to this application.

In AspectJ, this binding is specified in a declarative manner. However, the programmer, wearing the hat of the *application assembler* [2, Sect. 3.1.2], must take specific measures to ensure that the specified binding actually takes place, by compiling each core module with all the aspects that may apply to it. Thus, an aspect with global applicability may not apply to certain classes if these classes are 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 AspectJ2EE, aspect application is defined per bean class. The application also allows the application assembler to provide parameters to abstract aspects (including concrete pointcut definitions and initial field values).

Conceptually, aspect application in AspectJ2EE can be achieved using the generics-like syntax used in Sect. 4. However, when a large number of aspects is applied to a single bean (as is common in enterprise applications), the resulting syntactic constructs can be unwieldy. Hence, AspectJ2EE employs a semantically equivalent syntax based on XML deployment descriptors, following the tradition of using deployment descriptors to specify the application of services to EJBs in J2EE. Listing 1 gives an example.

As in the J2EE specification, bean ACCOUNT is defined by the `<entity>` XML element, and internal elements such as `<home>` specify the Java names that make this bean. In our extension of the deployment descriptor syntax, there is a new XML element, `<aspect>`, for each aspect applied to the bean. The `<pointcut>` element is used for binding any abstract pointcut, and the `<value>` element for specifying the initial value of fields.

As can be seen in the listing, four aspects are applied to our bean: `Lifecycle`, `Persistence`, `Security`, and `Transactions`. All four aspects are drawn from the `aspectj2ee.core` aspect library. Deploying this bean would result in the set of support classes depicted in Fig. 4.

The order of aspect specification determines the precedence of their application. Therefore, AspectJ2EE does not recognize the AspectJ statement **`declare precedence`**. The AspectJ2EE approach is more flexible, since it allows the developer to select a different order of precedence for the same set of aspects when applied to different beans, even in the same project. Intra-aspect precedence (where two or more advice from the same aspect apply to a single join point) is handled as per regular AspectJ.

Listing 1. 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>
    <aspect>
        <aspect-class>aspectj2ee.core.Lifecycle</aspect-class>
    </aspect>
    <aspect>
        <aspect-class>aspectj2ee.core.Persistence</aspect-class>
        <value name="primKeyClass">java.lang.String</value>
        <value name="primKeyField">id</value>
        <value name="fieldMap">id:ID,balance:BALANCE</value>
    </aspect>
    <aspect>
        <aspect-class>aspectj2ee.core.Security</aspect-class>
        <pointcut name="secured">execution(*(..))</pointcut>
        <value name="requiredRole">User</value>
    </aspect>
    <aspect>
        <aspect-class>aspectj2ee.core.Transactions</aspect-class>
        <value name="reentrant">false</value>
        <pointcut name="requiresnew">execution(deposit(..)) ||
                                     execution(withdraw(..))</pointcut>
        <pointcut name="required">execution(*(..)) && !requiresnew()</pointcut>
    </aspect>
</entity>
```

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.

5.2 Implementing Advice Using Deploy-Time Weaving

AspectJ2EE supports each of the join point kinds defined in AspectJ, except for **`handler`** and **`call`**. We shall now describe how, for each supported type of join point, advice can be woven into the entity bean code.

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 Listing 2 (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 Listing 2 (b).

Listing 2. (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 Listing 3 demonstrates the support for **`after() throwing`** advice. The advice, listed in part (a) of the listing, 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 listing. It shows how **`after() throwing`** advice are implemented by encapsulating the original implementation in a **`try`**/**`catch`** block.

Listing 3. (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, exception message: "
                       + ex.getMessage());
    throw ex;
}
```

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

The execution join point cannot refer to **`private`** or **`static`** methods, since the invocation of these methods cannot be intercepted using polymorphism. The AspectJ2EE compiler issues a warning if a pointcut matches only the execution of such methods.

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 class to which it is applied.

Unlike regular methods, constructors are limited with regard to the location in the code where the inherited implementation (**`super`**`()`) must be invoked. In particular, the invocation 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`**`()`) must adhere to these rules.

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`**`()`, and is possible only if the advised class contains a constructor that does not take any arguments.

It is generally preferable to affect the object initialization process simply defining a constructor in the aspect, rather than by applying advice to constructor execution join points.

Class Initialization Join Points. EJBs must not contain read/write static fields, making static class initialization mostly mute. Still, the **`staticinitialization`**`(`*`typeSignature`*`)` join point can be used (with **`after`**`()` advice only), resulting in a static initialization block in the advised class.

Field Read and Write Access Join Points. Field access join points match references to and assignments of fields. In AspectJ2EE, field access join points apply to EJB *attributes* only. Recall that attributes are not declared as fields; rather, they are indicated by the programmer using abstract getter and setter methods in the bean class. 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 defined also 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.

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 implemented by affecting the stub generated at deploy time for use by EJB clients (such as `_Account_Stub` in Fig. 4). For example, the **`around`**`()` advice from Listing 4 (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 listing. 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.

Listing 4. (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(* * Account.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 Sect. 6.

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

5.4 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. 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[5].

[5] 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. 4) stems from the fact that this is the root advised class, and is not related to the `Lifecycle` aspect per se.

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.

6 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 Sect. 2, 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*.

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.

6.1 Client-Side Checking of Preconditions

Method preconditions [28] 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.).

Listing 5 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 `PreconditionFailedException` 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.

Listing 5. 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 Account.deposit(float)) ||
         remotecall(public void Account.withdraw(float)))  && args(amount);

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

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

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 business process point of view. Thus, preventing the withdrawal of excessive amounts should not be defined as a precondition, but rather as part of `withdraw()`'s implementation.

6.2 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, for increased security; or we can apply a compression scheme (compressing information at the client, decompressing it at the server) to reduce the communications overhead; and so forth.

Consider an EJB representing a university course, with the method `register()` accepting a `Vector` of names of students (`Strings`) to be registered to that course. The aspect in Listing 6 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).

Listing 6. 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);
    }

    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).

6.3 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 Listing 7 can be used to enable client-side caching of rates.

7 Conclusions and Future Work

We believe that AspectJ2EE opens a new world of possibilities to developers of EJB-based 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.

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.

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.

And in addition 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.

Listing 7. An aspect that can be used for caching results from a currency exchange-rates EJB.

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

    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 invaocation

        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);
    }
}
```

Work is currently underway to implement AspectJ2EE as a new deployment process for the IBM WebSphere Application Server, version 5.0. We will then use AspectJ2EE to define every service currently provided by WebSphere as an aspect. Ideally, the result would be a standard-compliant J2EE server that can easily be extended using Aspect-Oriented Programming concepts.

References

1. Shannon, B., Hapner, M., Matena, V., Davidson, J., Davidson, J., Cable, L.: Java 2 Platform, Enterprise Edition: Platform and Component Specifications. Addison-Wesley (2000)
2. DeMichiel, L.G., Yalçinalp, L.U., Krishnan, S.: Enterprise JavaBeans specification, version 2.0. http://java.sun.com/j2ee/ (2001)
3. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York (1997) 220–242
4. Soares, S., Laureano, E., Borba, P.: Implementing distribution and persistence aspects with AspectJ. In: Proceedings of OOPSLA'02, Object Oriented Programming Systems Languages and Applications, ACM Press (2002)
5. Hao, R., Boloni, L., Jun, K., Marinescu, D.C.: An aspect-oriented approach to distributed object security. In: Proceedings of The Fourth IEEE Symposium on Computers and Communications, IEEE Press (1999)
6. Kim, H., Clarke, S.: The relevance of AOP to an applications programmer in an EJB environment. 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: 4th International Enterprise Distributed Object Computing Conference (EDOC 2000), IEEE Computer Society (2000) 252–261
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. Lecture Notes in Computer Science **2072** (2001) 327–355
9. American National Standards Institute, Inc.: Programming language – COBOL, ANSI X3.23–1985 edition (1985)
10. Shannon, B.: Java 2 platform enterprise edition specification, v1.3. http://java.sun.com/-j2ee/1.3/download.html#platformspec (2001)
11. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning, Greenwich (2003)
12. Pichler, R., Ostermann, K., Mezini, M.: On aspectualizing component models. Software – Practice and Experience **33** (2003) 957–974
13. Bodoff, S., Green, D., Haase, K., Jendrock, E., Pawlan, M., Stearns, B.: The J2EE Tutorial. Addison-Wesley (2002)
14. Duclos, F., Estublier, J., Morat, P.: Describing and using non functional aspects in component based applications. In: Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002). (2002) 22–26
15. Weinschenk, C.: The application server market is dead; long live the application server market. http://www.serverwatch.com/tutorials/article.php/2234311 (2003)
16. JBoss Group: JBoss product homepage. http://www.jboss.org/ (2003)
17. Burke, B., Brock, A.: Aspect-oriented programming and JBoss. http://www.onjava.com/-lpt/a/3878 (2003)
18. Mezini, M., Ostermann, K.: Conquering aspects with Caesar. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), ACM Press (2003)
19. Walker, R.J., Baniassad, E.L.A., Murphy, G.C.: An initial assessment of aspect-oriented programming. In: IEEE International Conference on Software Engineering (ICSE). (1999) 120–130
20. 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
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing. Addison-Wesley (1995)
22. Cohen, T., Hadad, E.: An enhanced framework for providing explicit support for crosscutting concerns in object-oriented languages. Submitted to Software – Practice and Experience (2004)
23. Sun Microsystems, Inc.: rmic - the Java RMI compiler. http://java.sun.com/j2se/1.4.2/docs/-tooldocs/solaris/rmic.html (2003)
24. IBM Corp.: IBM WebSphere Application Server product family homepage. http://www-3.ibm.com/software/info1/websphere/index.jsp?tab=products/appserv (2003)
25. Bracha, G., Cook, W.: Mixin-based inheritance. In Meyrowitz, N., ed.: Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming, Ottawa, Canada, ACM Press (1990) 303–311
26. Allen, E., Bannet, J., Cartwright, R.: A first-class approach to genericity. In: Proceedings of OOPSLA'03, Object Oriented Programming Systems Languages and Applications, ACM Press (2003)
27. Chan, P., Lee, R., Kramer, D.: The Java Class Libraries. 2 edn. Volume 1. Addison-Wesley (1998)
28. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice-Hall (1997)