

AspectJ2EE = AOP + J2EE

Towards an Aspect Based, Programmable and
Extensible Middleware Framework

<http://www.forum2.org/tal/AspectJ2EE.pdf>

Work by *Tal Cohen, Joseph Gil*

Seminar Presenter:
Giovanni Azúa Garcia
bravegag@hotmail.com

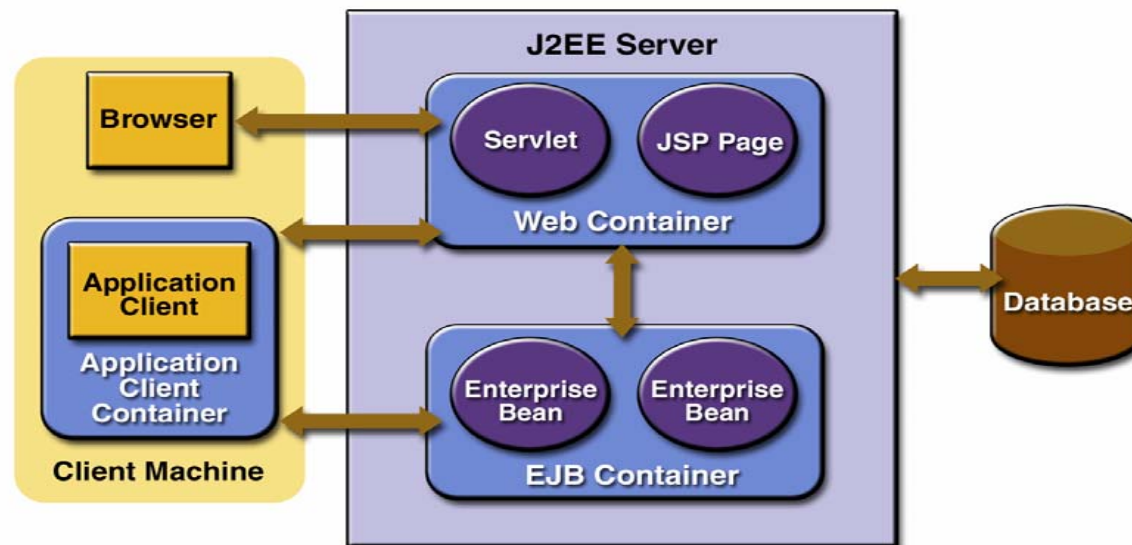
May 30, 2005

Outline

- ✓ J2EE Overview
- ✓ Why J2EE needs remedy?
- ✓ AOP to the rescue!
- ✓ Natural Marriage? J2EE + AOP
- ✓ Introducing AspectJ2EE
- ✓ Implementation of AspectJ2EE
- ✓ Personal Viewpoint
- ✓ Conclusion

J2EE Overview

- Middleware Architecture and supporting tools
- Platform for building Enterprise Applications:
Persistence, Transaction management, Security, Concurrency, Networking, Resource management, Messaging and Deploy-time customization

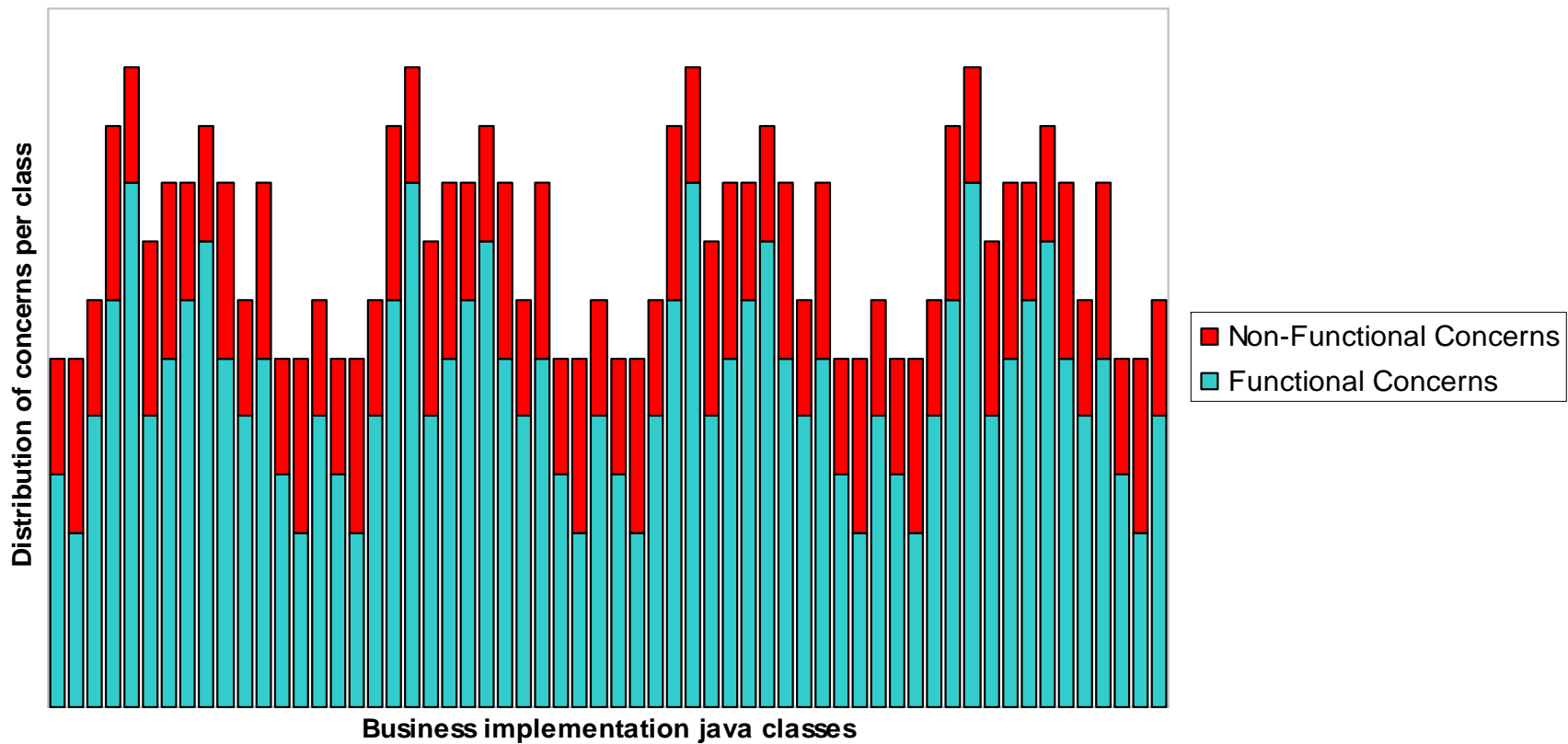


Why J2EE needs remedy?

- Scattered and Tangled code, consequence of crosscutting non-functional concerns e.g. logging, performance monitoring, memoization
- Lack of tailorability:
 - Developers must always burden with all services
 - Server implementations are black boxes i.e. are not extendible nor its services can be redefined
 - Services provided by different vendors can not be combined into an optimally customized Server e.g. Concurrency from BEA WebLogic, Messaging from IBM WebSphere, persistence from Oracle OC4J, etc.
- J2EE lacks support for introducing new services which are not part of its specification

Why J2EE needs remedy? (cont.)

Scattered and Tangled Example



AOP to the rescue!

- Supports decomposition of a system with additional dimensions, orthogonal to the class decomposition i.e. non-functional concerns
- Ability to modularize cross cutting concerns
- Provides two levels of modularity:
 - Base language (OO, Functional)
 - Extension (language/external tool) providing an extra dimension of concern
- Does not replace the primary decomposition
- It is all about modularizing

Natural Marriage? J2EE + AOP

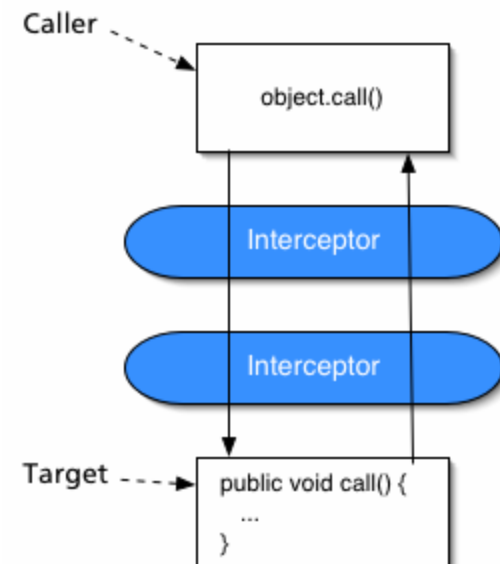
- Realization of J2EE can be decomposed into **aspects**:
 - *Persistence*
 - *Transaction management*
 - *Security*
 - *Concurrency*
 - *Networking*
 - *Resource management*
 - *Messaging*
 - *Deploy-time customization*
- Additional vendor-specific aspects may also be included:
 - *Load balancing*
 - *Failover*
 - *Performance monitoring*
 - *Memoization*
 - *Logging*

Introducing AspectJ2EE

- New AOP language that address specifically implementation of J2EE servers and applications
- Novel *deploy-time weaving* mechanism that preserves the object model and steps away from specialized JVM and byte-code manipulation
- Introduces *parametrized aspects* which are more reusable than aspects using AspectJ
- Support for *tier-cutting concerns* i.e. enables localization of concerns that cross not only program modules but also program tiers e.g. encryption and compression

Aspects weaving

- Weaving is the process of inserting relevant code from various aspects into designated locations, known as *join points*
- Weaving can be implemented by:
 - Special preprocessor
 - Post-compile processor
 - Load time
 - Specialized JVM
 - Residual Runtime Instructions?
 - Combination of the above
- Weaving introduces major conceptual bottleneck, reading the source is not sufficient for understanding its runtime behavior



Aspects weaving (cont.)

- Weaving implementation mechanisms explained before, known as *Unbounded Weaving*, transgress the boundaries of the object model:
 - Confuses language processing tools e.g. debuggers
 - Have adverse effects on generality and portability
- There are other approaches to Aspect Weaving within the dominion of OOP e.g. *Aspect Moderator* framework, utilizing the Proxy Design Pattern implemented in terms of Inheritance, Polymorphism and Dynamic Binding

AspectJ2EE deploy-time weaving

- Application of aspects is achieved by subclassing, during deployment stage, the Bean class that contains the business logic
- AspectJ2EE generate classes that inherit from, rather than replace, the core Bean classes
- The generation of sub- and support classes is declaratively governed by the extended deployment descriptors
- Sequence of aspect applications is translated into a chain of inheritance starting at the main bean class

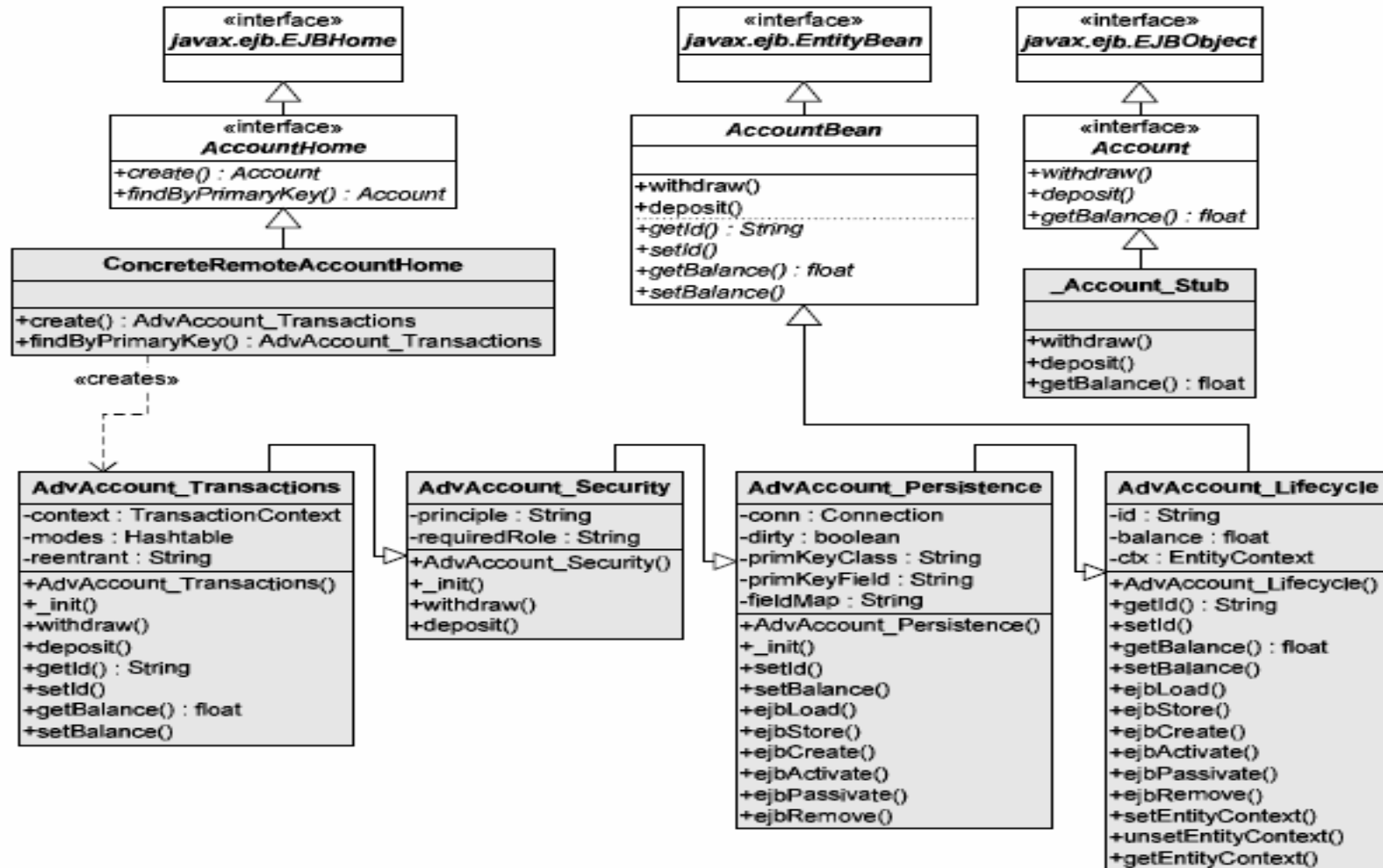
AspectJ2EE deploy-time weaving (cont.)

- With deploy-time weaving the code is unmodified, both at the source and binary level
- Targets Enterprise Beans only, whereas AspectJ targets Java classes
- Integrates well with J2EE applications by extending the XML declarative deployment descriptors for specifying aspect pointcuts and advices
- Does not support field read and write join points but only J2EE attributes i.e. read and write methods (getX and setX bean-type)

The Core Aspects Library

- AspectJ2EE's definition includes a "standard" library of core aspects:
 - **Aspectj2ee.core.Lifecycle** provides default implementation to the J2EE lifecycle methods e.g. **setEntityContext**, **getEntityContext**, **unsetEntityContext**
 - **Aspectj2ee.core.Persistence** provides a CMP-like persistence service. Advices some of the lifecycle EJB methods
 - **Aspectj2ee.core.Security** limit the access to various methods based on user authentication
 - **Aspectj2ee.core.Transactions** used to provide transaction management capabilities to all business-logic methods

AspectJ2EE deploy-time weaving: Example



AspectJ2EE deploy-time weaving: Pros and Cons

- Pros
 - OOP compliant implementation of AOP that not compromise stability, unmodified original code at source and binary levels e.g. tools accessibility (debuggers, profilers, code metrics)
 - Integrates seamlessly with EJB deployment process
- Cons
 - Very limited applicability of aspects:
 - Can only be applied to the Bean class
 - Can only be applied to public, non-final and non-static methods i.e. can not be intercepted using polymorphism
 - Functional concerns suggested to be moved to aspects in order to solve the *exceptional case issue* (Bean instances exempt from non-functional services)

Aspects as Generics and Mixins

- Generics are class of classes where the concrete class type is resolved once all of the generic parameters are provided whether in instantiation or subclassing
- Mixins programming is a style of software development where units of functionality are created in a class and then mixed in with other classes, conceptually equivalent to `class Aspect<bean B> extends B { /* */ }`

"The term mixin comes from an ice cream store in Sommerville, Massachusetts, where candies and cakes were mixed into the basic ice cream flavors. This seemed like a good metaphor to some of the object-oriented programmers who used to take a summer break there, especially while working with the object-oriented programming language SCOOPS." (*SAMS Teach Yourself C++ in 21 Days*, 4th ed., p. 458.)

Aspects as Generics and Mixins (cont.)

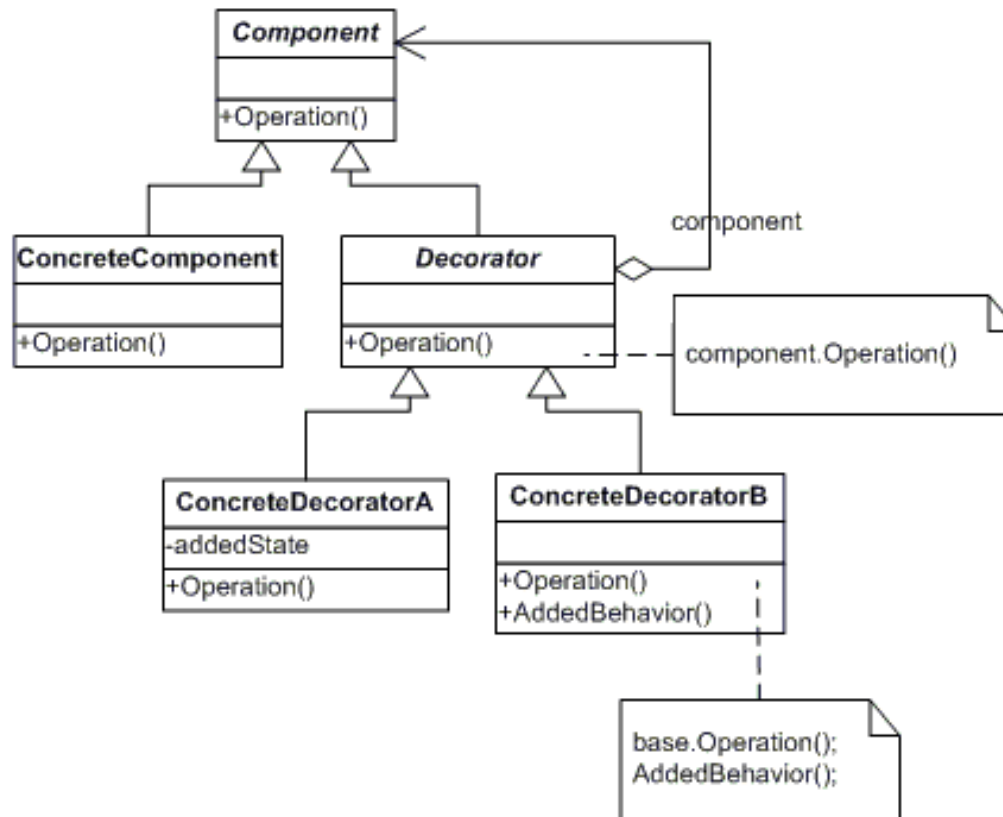
- AspectJ2EE is asserted to be conceptually equivalent to applying generics as aspects to the Bean class e.g.

`Persistence<Lifecycle<Account Bean>>`

- AspectJ2EE is also asserted to show that aspects can be implemented using a mixins -like mechanism
- The Decorator Design Pattern (a.k.a. Wrapper) could have much better served the purpose of weaving the different aspects to the Bean class as implementation of AspectJ2EE without conceptually requiring inexistent Java language features i.e. mixins or misusing the generics support new in Java 1.5

... just my 5ct

Aspects as Generics and Mixins (cont.)



Parametrized aspects

- AspectJ2EE introduces the new notion of parametrized aspects, enabling *abstract pointcuts* whose definition is provided at deployment time, modeled by the following form: `class Aspect<B,P1, ... ,Pk> extends B { /* ... */ }` where each `Pi, i= 1..k` is a formal parameter to the aspect representing an abstract pointcut
- Parametrized aspects also include what is defined as an *abstract field*, a field whose initial value is specified at deployment time, and modeled by the form: `class Aspect<B,P1, ... ,Pk,V1, ... ,Vn> extends B { /* ... */ }` where each `Vi, i= 1..n` is a formal parameter to the aspect representing an abstract field

Parametrized aspects (cont.)

- Abstract fields makes possible to apply the same aspect more than once to a single bean class, with each repeated application providing a distinct new extension e.g.

```
Security<Security<Account, Pteller, "teller">, Pclient, "client">
```

Support for Tier-Cutting concerns

- AspectJ2EE introduces a new join point keyword **remotecall** semantically similar to AspectJ's **call** join point designator, defining a join point at a method invocation site, but it only applies to remote calls; local calls are unaffected
- Remote call join points are implemented by affecting the stub generated at deploy time for use by EJB clients e.g.

(a) Sample advice for a method's **remotecall** join point.

(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.");  
}
```

Innovative uses of AOP in Multi-tier

- Client-side checking of preconditions: Remote call join points make possible to have client-side preconditions checked and violations flagged right before the remote call takes place, this approach avoids the remote call overhead
- Symmetrical data processing: Ability to add code both at the sending and receiving ends of remotely-invoked methods, as additional layers in the communication stack enables implementation of e.g. compression and encryption among other interesting uses ...
- Memoization: Caching method results, although is recommended to be done with special care as the client tier do not have a way to know when the data becomes stale

Personal viewpoint

- The paper title, description and project name is totally misleading since the AspectJ2EE framework **only** applies to EJB Containers and not to all J2EE technologies e.g. J2EE Web Containers
- The Case for AOP in J2EE was a bit **too forced**. There was no mention to using existing and proven J2EE Core Design Patterns for modularizing some crosscutting non-functional concerns e.g. BMP using Data Access Objects
- Suggested use of AspectJ2EE to blur separation between the two main types of EJB is **by no means!** an advantage. Entity and Session Beans have totally different semantics:
 - Entity Beans' only concern is persistence
 - Session Beans represent processes and may manipulate Entity Beans

Personal viewpoint (cont.)

- Suggested use of aspects to cover crosscutting functional concerns i.e. business logic, introduces anomalies e.g.
 - Scatters functional concerns over different dimensions!
 - Obscures intent, affects maintainability
 - Testing becomes a daunting task!
- Misconceptions related to the J2EE platform e.g.
 - CMP Beans can only map to a physical table (**wrong!**)
 - CMP Beans of no use with non-ER data e.g. XML (**wrong!**)
 - Described set of services officially supported by J2EE specification e.g. Load Balancing (**wrong!**)
 - Each Bean represents one component of the underlying data model (**wrong!**)

Personal viewpoint (cont.)

- There is a major pitfall with the whole idea of providing default Lifecycle implementation aspects “Core Aspects” in AspectJ2EE e.g. adding these aspect to a Bean will **override and overwrite** the developer’s implementation of the lifecycle methods:
 - Developing the Bean is the job of the *Bean Provider*
 - Assembling the Bean is the job of the *Application Assembler*
 - Deploying the Bean is the job of the *Deployer* who also happens to customize the deployment descriptors

Conclusion

- AspectJ2EE introduces very innovative and sound applications of AOP specifically in the context of multi-tier applications
- Although the AspectJ2EE proposed implementation is not completely wrong, it has serious pitfalls and limitations and needs a more elaborated approach e.g. completely neglects the proven and widely accepted applicability of J2EE Core Design Patterns
- Aspects in general as every technology has its limitations:
 - Risk of scattered concerns among dimensions
 - Runtime behavior can not be deduced from OOP anymore
 - Magnifies the complexity of e.g. Deployment, Debugging and Testing specially in the J2EE platform

Questions?

