Real World Java EE Patterns - Rethinking Best Practices

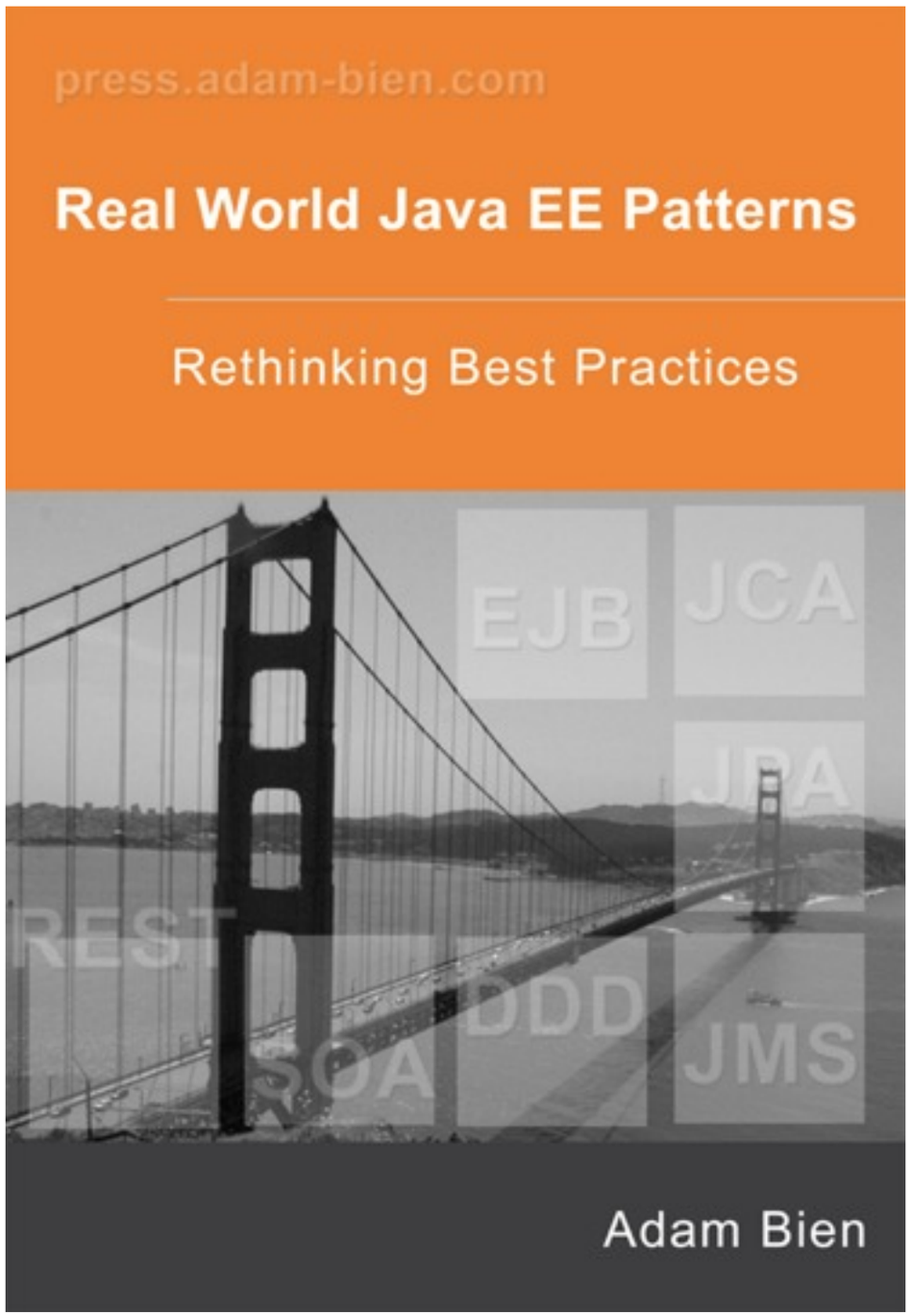
www.adam-bien.com
blog.adam-bien.com

designed by www.graphikerin.com

Introduction

- Expert Group Member (jcp.org) of Java EE 6, EJB 3.1, Time and Date and JPA 2.0
- Java Champion,Netbeans Dream Team Member, (JavaONE) speaker, freelancer, consultant and author: 7 German books + working on "**Real World Java EE Patterns– Rethinking Best Practices**" http://press.adam-bien.com
- Trainer, Developer and Architect (since JDK 1.0)
- Project owner/committer: greenfire.dev.java.net, p4j5.dev.java.net, fishfarm.dev.java.net

Based on:



Java™ Champions

blog.adam-bien.com

1. EJBs Are Heavyweight

EJBs Are Heavyweight

Configuration is mainly gone, because of conventions... (there was no XML in my last projects except persistence.xml)

EJB 3 are just annotated Java classes (if you love XML you can even use just Deployment Descriptors instead of annotation)

Container “services” like transactions, security, concurrency or state are implemented with aspects (often realized with dynamic proxies)

EJBs Are Heavyweight

"POJOs" are just JavaBeans maintained by another container, using similar techniques as EJB 3.1

EJB 3 containers are surprisingly small. Glassfish v3 EA comes with two jars (688kB + 8kB = 796kB). The EJB 3 container is an OSGI bundle...

The whole EJB 3.1 API is about 47 kB.

2. EJBs Are Not Portable

EJBs Are Not Portable

J2EE 1.4 was underspecified :-) - EJB 3.X / JPA specs cover more real world stuff (locking, optimistic concurrency etc.).

Vendor specific deployment descriptor were painful for migration - they are basically gone.

In most cases a EJB-JAR module is nothing but a JAR without any XML descriptors (neither ejb-jar.xml nor vendor specific)

Vendor specific annotations are not needed to develop a Java EE application.

There is **NOTHING** vendor specific in an EAR. The portability is really good.

3. EJBs Are Not Extensible

EJBs Are Not Extensible

How to inject a Guice component into an EJB 3:

```
@WebService
@Stateless
@Local(ServiceFacade.class)
@Interceptors(PerMethodGuiceInjector.class)
public class ServiceFacadeBean implements ServiceFacade{

    @Inject
    private MessageProvider message;

    public String getHello(String msg){
        return msg + " " + message.getMessage();
    }
}
```

EJBs Are Not Extensible

The Guice (fluent) configuration:

```
import com.google.inject.Binder;
import com.google.inject.Module;

public class MessagingModule implements Module{
    public void configure(Binder binder) {
        binder.bind(MessageProvider.class).to(MessageProviderImpl.class);
    }
}
```

EJBs Are Not Extensible

You only need an interceptor:

```
/**
 * @author adam-bien.com
 */
public class PerMethodGuiceInjector{

    private Injector injector;

    @PostConstruct
    public void startupGuice(InvocationContext invocationContext) throws Exception{
      invocationContext.proceed();
      this.injector = Guice.createInjector(new MessagingModule());
    }

    @AroundInvoke
    public Object injectDependencies(InvocationContext invocationContext) throws Exception{
        this.injector.injectMembers(invocationContext.getTarget());
        return invocationContext.proceed();
    }
}
```

EJBs Are Not Extensible

Interceptors are able to access the Bean instance directly.

Having an instance available - you can manipulate it; inject members use reflection to invoke methods, or set fields...

It is very interesting for the integration of existing “legacy” IoC frameworks :-)

4. EJBs Are Slow

EJBs Are Slow

The throughput of the EJB 3 solution was 2391 transactions/second. The slowest method call took 7 milliseconds. The average wasn't measurable. Please keep in mind that in every request two session beans were involved - so the overhead is doubled.

POJO: The throughput of the POJO solution was 2562 requests/second (request - there are no transactions here). The slowest method call took 10 ms.

The difference is 171 requests / seconds, or 6.6%

5. EJBs Are Too Complex

EJBs Are Too Complex

Java EE is distributed and concurrent platform per definition.

It mainly abstracts already existing products (messaging, EIS, relational databases)

Distributed programming with shared state is always a challenge.

In the Cloud Computing / SOA era non-functional requirements like: monitoring (JMX), management, fail-over or elasticity become more and more important.

Think about the ratio between the essential and accidential complexity...

6. EJBs Are Hard To Develop

EJBs Are Hard To Develop

Simplest possible EJB 3.1:

```
@Stateless
public class SimpleSample{
    public void doSomething() { /*business logic*/  }
}
```

EJBs Are Hard To Develop

How to compile:

You will need the the EJB 3.0 / 3.1 API in the classpath, or at least the @Stateless annotation.

How to deploy:

Just JAR the class and put the JAR into e.g: [glassfishv3-prelude-b23]\glassfish\domains\domain1\autodeploy

How to use:

```
import javax.ejb.EJB;
public class MyServlet extends HttpServlet{
@EJB
private SimpleSample sample;
}
```

Agile Manifesto

blog.adam-bien.com

Agile Manifesto in Java EE Context

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

[**Pragmatic solutions** over infinite layers indirections, frameworks and patterns] (decorated by Adam Bien)

That is, while there is value in the items on the right, we value the items on the left more.

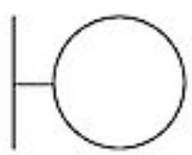
Entity Control Boundary

Entity Control Boundary

The “lightweight” way to design applications:

Entity: persistent object (“domain objects” from conceptual model)

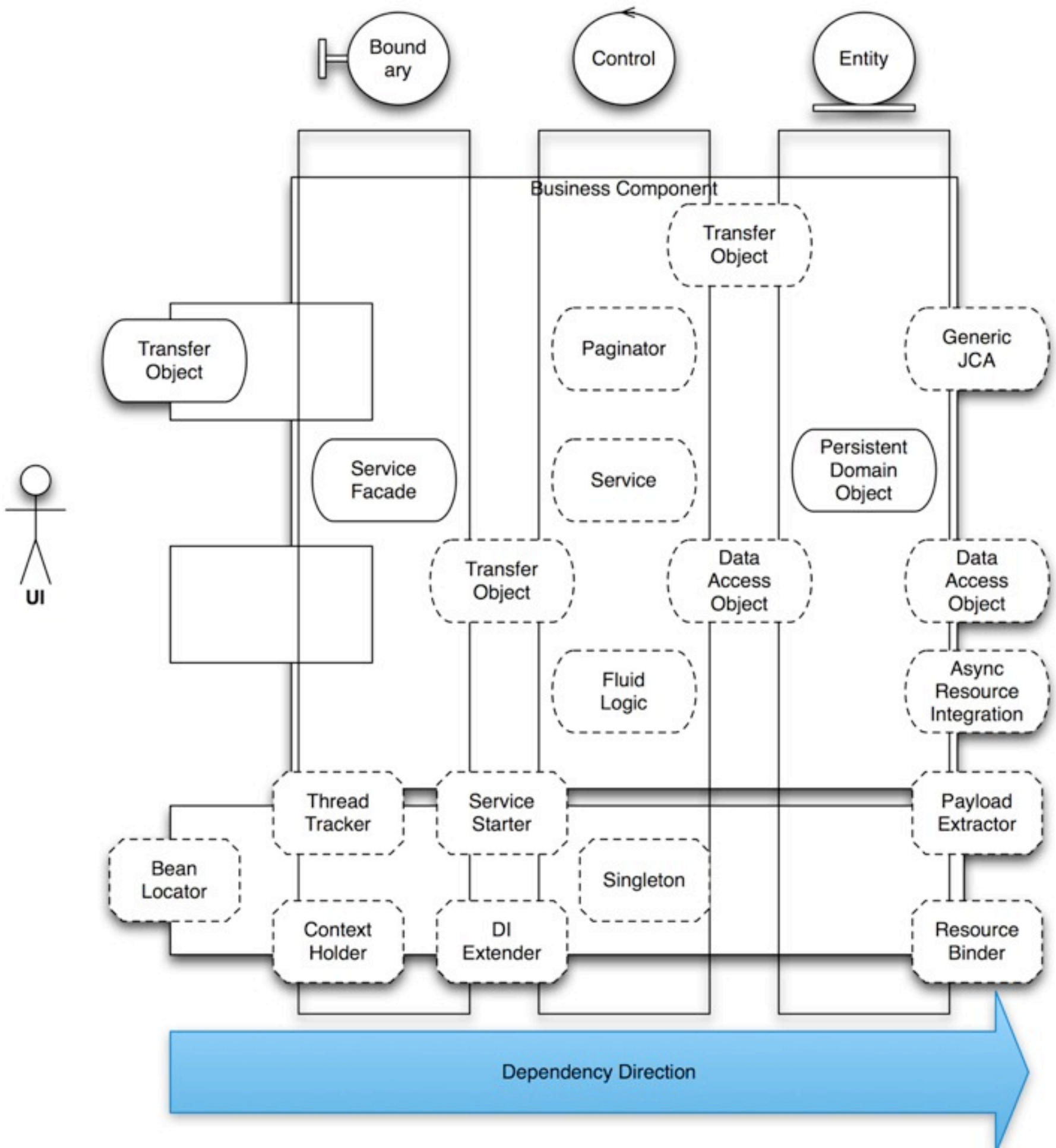
Control: process knowledge - entity independent logic, the glue between the boundary and the entity



Boundary: the interface between the actor and the use case

SOA Architecture

SOA Architecture



Service Facade

Service Facade (Context)

“In the J2EE era Session Facades were just wrappers of Entity Beans or DAOs. They were motivated rather by the shortcoming of the spec, than by design best practices. The technical nature of the Session Façade was the reason for their thin logic. Application Service was the use case controller or façade, which coordinated multiple Session Facades. The landscape, however, changed in Java EE. “

Service Facade

In Java EE an explicit remote and transactional boundary is still needed. The exposure of fine grained business logic over remote interfaces simply doesn’t work.

Network latency is too high for fine grained access and it is hard to execute fine grained methods in a transaction context over the network.

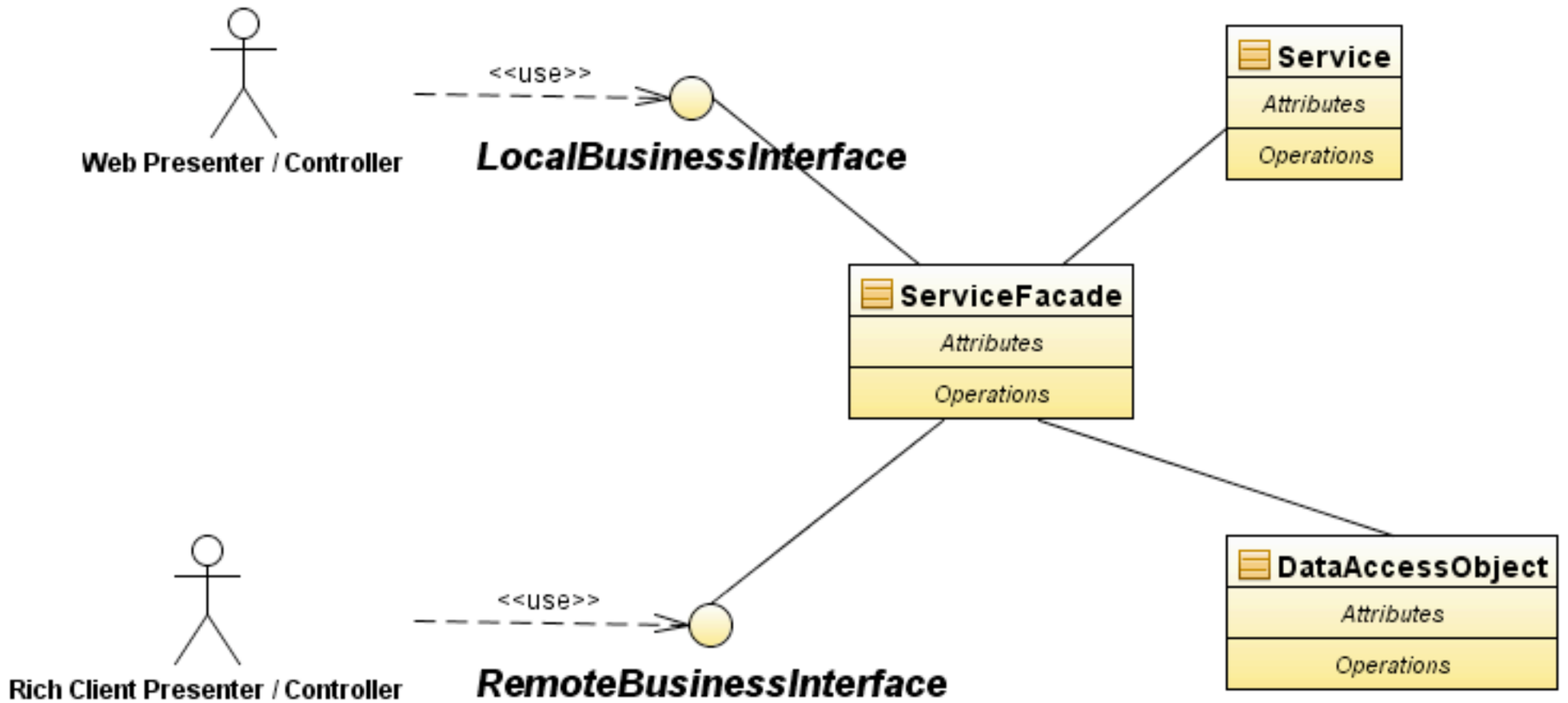
Service Facade (Solution)

Service Façade is a Stateless, in exceptional cases Stateful Session Bean with a local business interface.

A remote business interface should be only provided if it is going to be used from outside the JVM and not injected into Servlet, Backing Bean or other web component.

An Eclipse or Netbeans RCP (Rich Client Platform) application would be one example for that.

Service Facade



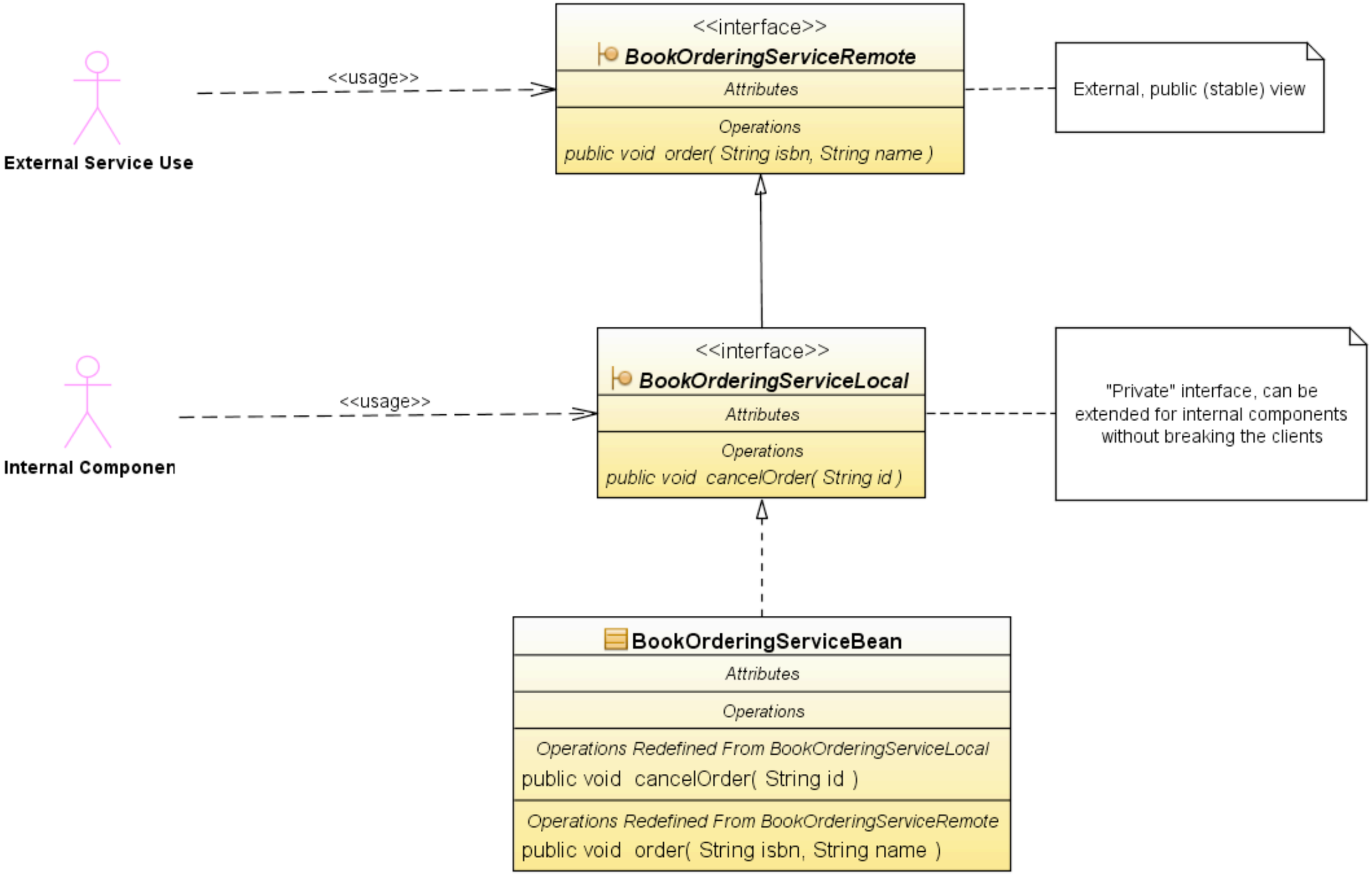
Service Facade (Conventions)

Service Façade resides in a component which is realized as Java-package with domain-specific name e.g. ordermgmt.

The realization of the façade (business interface and the bean implementation) resides in a sub-package with the name facade e.g. ordermgmt.facade. This makes the automatic verification of the architecture easier.

The business interface is named after business concepts, without the obligatory local or remote suffix e.g. OrderService and OrderServiceBean.

Dual View Service Facade



Service

Service (Context)

The origin context of a Session Facade (SF) was defined in the Core J2EE pattern as following:

“Enterprise beans encapsulate business logic and business data and expose their interfaces, and thus the complexity of the distributed services, to the client tier.”

- http://java.sun.com/blueprints/corej2eepatterns/Patterns/SessionFacade.html

Service (Context)

The context changed in Java EE quite a bit:

- A Service is a procedural activity.
- It realizes activities or sub processes.
- In an object oriented, domain driven context, a Service realizes cross cutting, domain object independent logic.
- In a SOA a Service plays the main role and implements the actual business logic.

Service (Forces)

Services should be independent of each other.

The granularity of a Service is finer than of a Service Façade.

The Services are not accessible and even visible from outside the business tier.

A Service should be not accessible from an external JVM. A remote Service invocation doesn't make sense and should be avoided.

A Service is aimed to be reused from other component or Service Façade.

The execution of a Service should be always consistent. Its methods should either have no side effects (be idempotent), or be able to be invoked in a transactional context.

Service (Solution)

A Service is always local and comes with the `TransactionAttributeType.MANDATORY` transaction attribute:

```
@Stateless
@Local(DeliveryService.class)
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public class DeliveryServiceBean implements DeliveryService {
}
```

Service (Conventions)

A Service is a local, stateless session bean.

Service resides in a component which is realized as Java-package with domain-specific name e.g. ordermgmt.

The realization of the service (business interface and the bean implementation) resides in a sub-package with the name “service” e.g. ordermgmt.service. This makes the automatic verification of the architecture easier.

The business interface is named after business concepts, without the obligatory local or remote suffix e.g. OrderService and OrderServiceBean.

Service (Conventions)

It is not required to use the term “Service” in the name of the bean – its redundant. For identification purposes you could use a @Service annotation.

The Service is always invoked in the context of an existing transaction. It is deployed with the Mandatory transaction attribute.

Domain Driven Design

Domain Driven Design

Domain-driven design (DDD) is an approach to the design of software, based on the two premises that complex domain designs should be based on a model, and that, for most software projects, the primary focus should be on the domain and domain logic (as opposed to being the particular technology used to implement the system).

Domain Driven Design

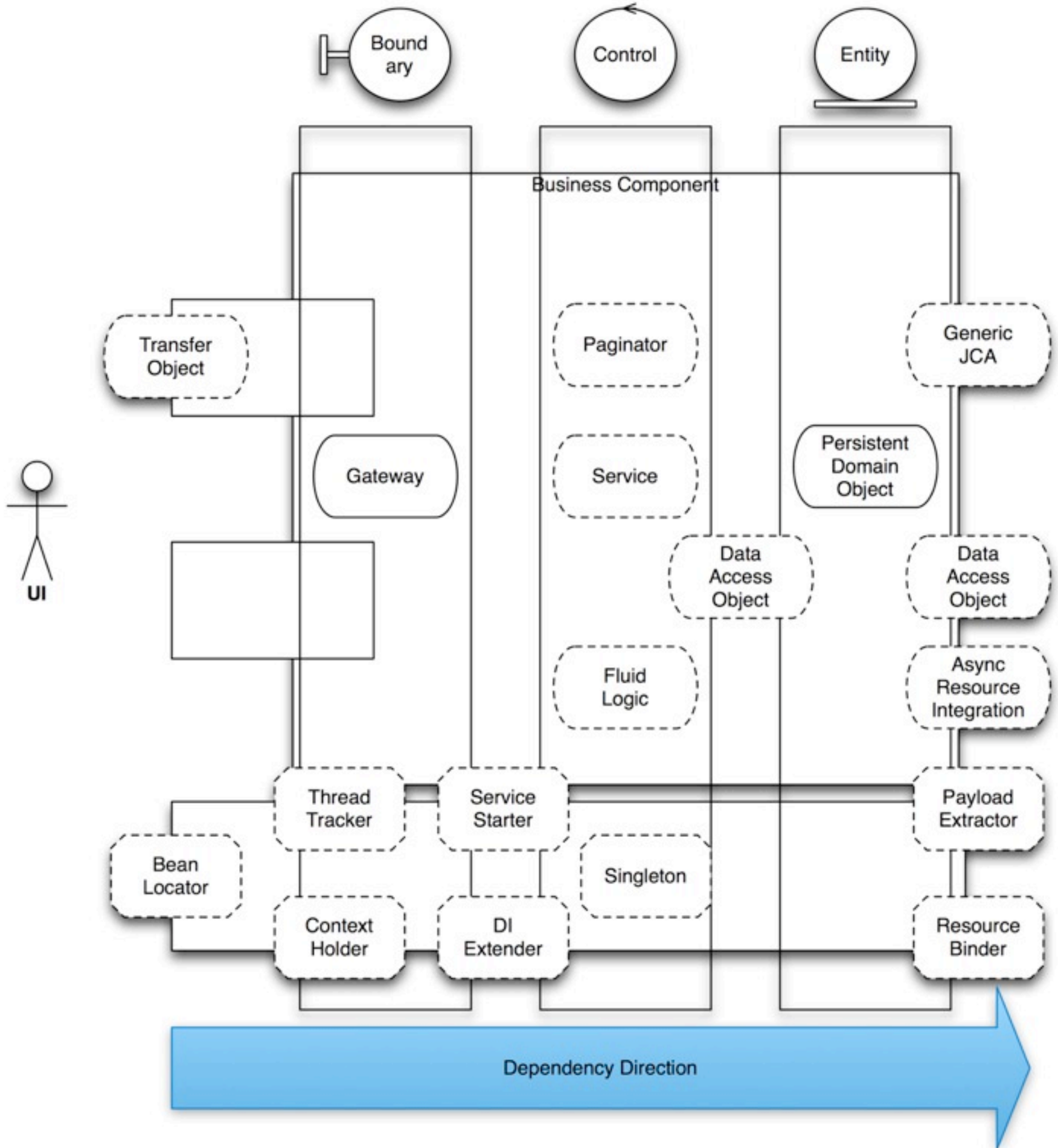
The idea:

The domain model should form a common language given by domain experts for describing system requirements, that works equally well for the business users or sponsors and for the software developers.

Domain Driven Architecture

blog.adam-bien.com

DD Architecture



Persistent Domain Object

Persistent Domain Object (Context)

The origin problem description in J2EE Core Patterns was short and sound: “You have a conceptual domain model with business logic and relationship.” http://www.corej2eepatterns.com/Patterns2ndEd/BusinessObject.htm

Even in the origin description of the Business Object J2EE Pattern the realization of the conceptual model with procedural approaches was considered as dangerous regarding to bloating, code duplication spread over different modules and therefore hard to maintain.

Persistent Domain Object (Problem)

The vast majority of J2EE applications were build in the procedural way.

The business logic was decomposed into tasks and resources, which were mapped into Services and anemic, persistent entities.

The procedural approach works surprisingly well until type specific behavior for domain objects has to be realized.

Persistent Domain Object (Problem)

The attempt to realize object oriented algorithms with procedural techniques ends up in many `instanceof` checks and / or lengthy if-statements.

Such type checks are required, because the domain objects are anemic in the procedural world, so that inheritance doesn't really pays off.

Even in case inheritance was used for designing the domain model, the most powerful feature – polymorphic behavior — and so in Services or Service Facades.

Persistent Domain Object (Forces)

Your business logic is complex.

The validation rules are domain object related and sophisticated.

The conceptual model can be derived from the requirements and mapped to domain objects.

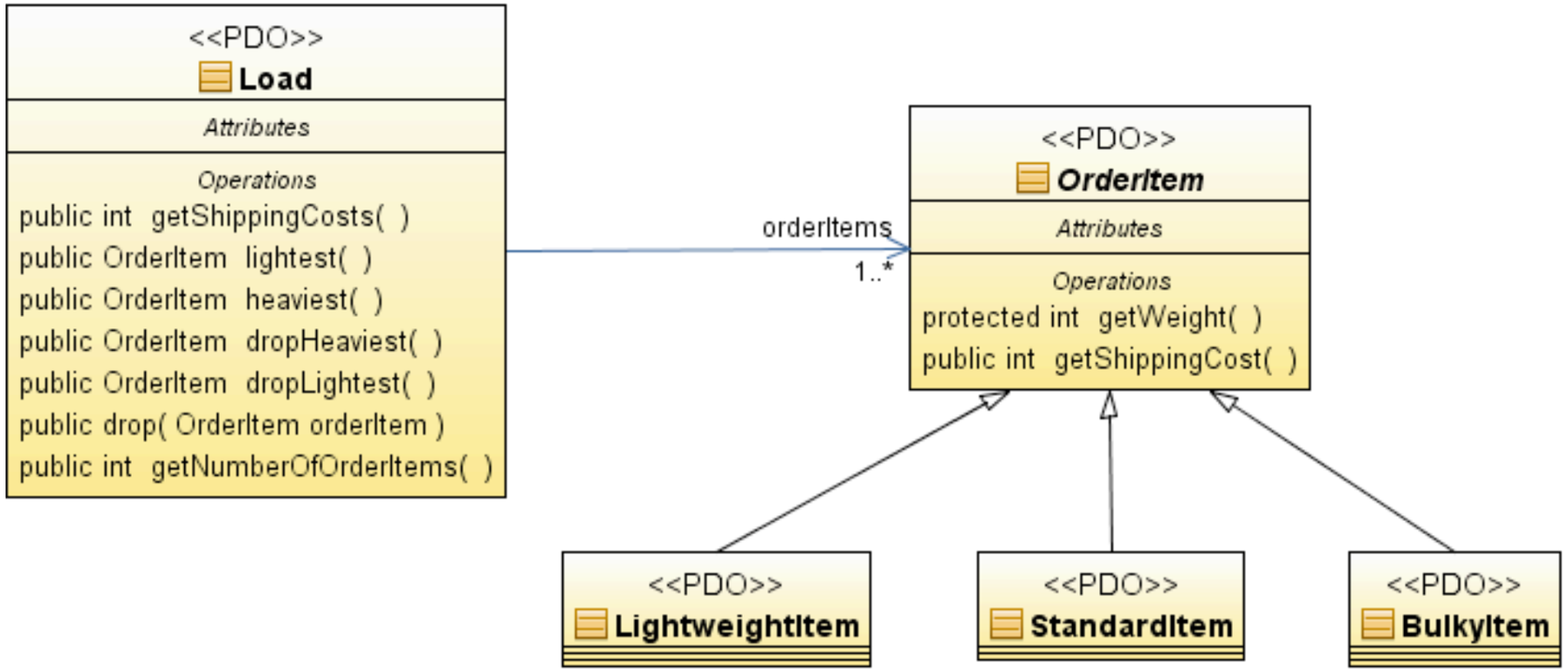
The domain objects have to be persisted in relational database (it’s the common case).

The Use Cases, User Stories or other specification documents already describe the target domain in object oriented way. The relation between the behavior and the data can be directly derived from the specification.

Persistent Domain Object (Forces)

It is a green field project, or at least the existing database was designed in way that allows the use of JPA. It means: the tables and columns are reasonable named and the database is not overly normalized.

Persistent Domain Object - sample



Persistent Domain Object - procedural type checks...

```
int computeShippingCost(Load load){
       int shippingCosts = 0;
       int weight = 0;
       int defaultCost = 0;
       for (OrderItem orderItem : load.getOrderItems()) {
           LoadType loadType = orderItem.getLoadType();
           weight = orderItem.getWeight();
           defaultCost = weight * 5;
           switch (loadType) {
               case BULKY:
                   shippingCosts += (defaultCost + 5);
                   break;
            case LIGHTWEIGHT:
                shippingCosts += (defaultCost - 1);
                break;
            case STANDARD:
                shippingCosts += (defaultCost);
                break;
            default:
              throw new IllegalStateException("Unknown type: " + loadType);
           }
       }
       return shippingCosts;
}
```

Type checks - the object oriented way

```
public int getShippingCosts() {
        int shippingCosts = 0;
        for (OrderItem orderItem : orderItems) {
            shippingCosts += orderItem.getShippingCost();
        }
        return shippingCosts;
}
```

Inheritance does the work

```
public class BulkyItem extends OrderItem{
 public BulkyItem(int weight) {
        super(weight);
    }
    @Override
    public int getShippingCost() {
        return super.getShippingCost() + 5;
    }
}
```

The procedural construction

```
Load load = new Load();
OrderItem standard = new OrderItem();
standard.setLoadType(LoadType.STANDARD);
standard.setWeight(5);
load.getOrderItems().add(standard);
OrderItem light = new OrderItem();
light.setLoadType(LoadType.LIGHTWEIGHT);
light.setWeight(1);
load.getOrderItems().add(light);
OrderItem bulky = new OrderItem();
bulky.setLoadType(LoadType.BULKY);
bulky.setWeight(1);
load.getOrderItems().add(bulky);
```

...and the fluent way

```
Load build = new Load.Builder().
    withStandardItem(5).
    withLightweightItem(1).
    withBulkyItem(1).
    build();
```

Persistent Domain Object (Conventions)

PDOs are JPA entities with emphasis to domain logic and not the technology.

PDO resides in a component which is realized as Java-package with domain-specific name e.g. `ordermgmt`.

The PDO resides in a sub-package (layer) with the name `domainordermgmt.domain`. This makes the automatic verification of the architecture easier.

The name of the domain object is derived from the target domain.

Getters and setters are not obligatory – they should be only used in justified cases.

Gateway

Gateway (Context)

PDOs are already consistent, encapsulated objects with hidden state. There is no need for further encapsulation – they can be directly exposed to the presentation.

A Gateway provides an entry point to the root PDOs.

A Gateway could be even considered as an anti-Service Façade – in fact its responsibilities are inverted.

Gateway (Problem)

PDOs are passive artifacts.

It is not possible to access them directly without an execution context.

The next problem is the stateless nature of most Java EE applications...

After a method invocation of a transaction boundary (e.g. a Stateless Session Bean) all JPA-entities (PDOs) become detached. The client loses its state.

Gateway (Problem)

This forces you to transport the whole context back and forth between the client and the server, which leads to the following problems:

- Heavily interconnected PDOs become hard to merge.
- Even for fine grained changes, the whole graph of objects has to be transported back to server.
- It is not always possible to merge the graph automatically and even consistently.

Gateway (Solution)

The solution is very simple. Just create a perfect “Anti Service Façade”.

Instead of cleanly encapsulating the PDOs, just try to as conveniently for the UI as possible expose PDOs to the adjacent layer.

Allow the user to modify the PDOs directly without any indirection.

The described approach above actually contradicts the common J2EE principles, where encapsulation seemed to be the only way to achieve maintainability. This is only true for perfect abstractions and encapsulations, which are very hard to find in real world projects.

Gateway (Solution)

The inverse strategy works even better for some Use Cases – just get rid of any layer which is probably leaky anyway and expose the business logic directly to the presentation tier.

Every change in the structure of the persistence layer would be immediately visible in the UI – this makes the implementation of feature requests really easy.

Gateway (Solution)

Your presentation is coupled to the particular implementation of the business logic, but the concrete implementation is already encapsulated.

JPA abstracts from the particular provider, and EJBs are nothing else than annotated POJOs.

The concrete state and implementation of domain specific logic is well encapsulated too – it’s the main responsibility of the PDOs.

Gateway - the solution again...

The solution for the problem is the introduction of state on the server side.

A stateful Gateway can keep the PDOs attached with an `EntityManager` declared as `PersistenceContext.EXTENDED`.

The `EntityManager` needs a transaction only as a trigger to flush the changes to the database, which can be started by a method which overrides the class default.

Gateway - sample:

```java
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
@Local(OrderGateway.class)
public class OrderGatewayBean implements OrderGateway{

    @PersistenceContext(type=PersistenceContextType.EXTENDED)
    EntityManager em;

    private Load current;

    public Load find(long id){
       this.current = this.em.find(Load.class, id);
       return this.current;
    }

    public Load getCurrent() {
        return current;
    }

    public void create(Load load){
        this.em.persist(load);
        this.current = load;
    }

    public void remove(long id){
        Load ref = this.em.getReference(Load.class, id);
        this.em.remove(ref);
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void save(){
        //nothing to do
    }

    public void update(){
        this.em.refresh(this.current);
    }

    public void removeCurrentLoad(){
        this.em.remove(this.current);
        this.current = null;
    }

    @Remove
    public void closeGateway(){

    }
}
```

Gateway (Conventions)

A Gateway resides in a component which is realized as Java-package with domain-specific name e.g. ordermgmt.

The Gateway resides in a sub-package (layer) with the name “facade” e.g. ordermgmt.facade. This makes the automatic verification of the architecture easier. The Gateway resides therefore in the same sub-package as a Service Façade.

A Gateway is often named after the cached root entity – it is not necessary to keep the name “Gateway”.

Thank you!

blog.adam-bien.com

Interested in „highend“ trainings, coaching, consulting?

…just send me an email => abien@adam-bien.com