



Domain Driven Technology Migration

Saving the Domain from Legacy Death

Dan Bergh Johnsson
Omegapoint Consulting AB, Sweden





Jazz Page: Inspirations

Abelson *“Refactoring”*
Sussman *“Domain Driven Design”*
“Anti-Patterns” **Kent Beck** **Bertrand Meyer**
GoF: “Design Patterns” **Martin Fowler**
Eric Evans *“The Pragmatic Programmer”* **“Structure and Interpretation of Computer Programs”**
“Object-oriented Software Construction” **Cons T Åhs**
“Mathematical Theory of Domains” *“The Formal Semantics of Programming Languages”*





Assumptions

- Experienced programmer
- Large code-bases
- Maintenance and extensions of systems
- Refactoring
- EJB
- Spring



Too-Usual Scenario

- Existing codebase
 - large system, perhaps developed under pressure
- Using old framework
- Young hip framework coming up
- Maintenance will be hard
 - Competence hard to find
- Old system will die slowly
- New system written from scratch
- Lots of coded understanding thrown away



Underlying Problem

- "Project" metaphor is broken
 - Assume development resource-high-intensive
 - Assume maintenance resource-low-intensive
- When does "development" end?
 - When project finish
- Simply not true
- Gives poor-maintained legacy systems



Legacy; Is That Bad?

- Legacy = transmitted ... from an ancestor
 - Merriam-Webster; examples: money and knowledge
- Legacy functionality
 - Encoded knowledge
- Legacy structure
- How does the code look?
- How would it have looked, had you written it today?



Alternatives in Scenario

- Throw away, re-implement
- Rip apart, reuse parts
- Restructure
 - separate application from framework
 - “purify” application logic
 - replace framework



Ambition

- Show realistic to do
- Will not cover all details
- Give Hope!



Migration Projects

- EJB 2.1 -> Hibernate
- Struts 1 -> JSF
- JSF -> Struts 2
- Home-grown -> Struts 1
- EJB 2.x -> Spring
- Spring -> EJB 3
- ... or *vice versa*



Example

- “Old” EJB 2.x
- “New” Spring
- Chosen because well known
- Imagine “typical” legacy-code
- Notes
 - Not EJB-bashing
 - Will not result in “best Spring”



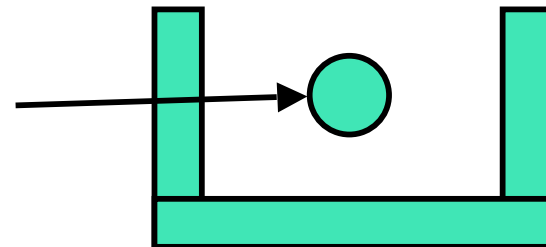
Method

- Domain Driven Refactorings
 - *Make implicit knowledge encoding explicit*
- Separation of concerns
 - application logic
 - framework specific code
- Extract framework-independent classes
 - Reuse in new framework
 - *i.e.* replace old framework

- Disclaimer: Small example
 - not all aspects of large project

Container and Component

- Component
 - Application logic
 - Technical code
- Container
 - Services
 - Naming
 - Transactions
 - Contact to outside



Enterprise JavaBeans vs Spring Framework

- JSR Standard w providers
 - Geronimo, BEA, IBM, ...
- Component
 - Component interface
 - extends EJBObject
 - Home interface
 - Component class
 - implements SessionBean
 - ejb-jar.xml
 - transaction attributes
 - dependency declarations
- Container
 - lots of services
 - application xml
 - Supporting classes generated at deploy
- Open Source framework
 - SourceForge
- Component
 - Component interface
 - Component class
 - implements interface
- Container
 - wiring framework
 - beans.xml



Example: Night at Bar

- Accumulated drinking
 - Price list
 - One tab at a time
 - Credit control
-
- Note: state during long-running business process



Roadmap

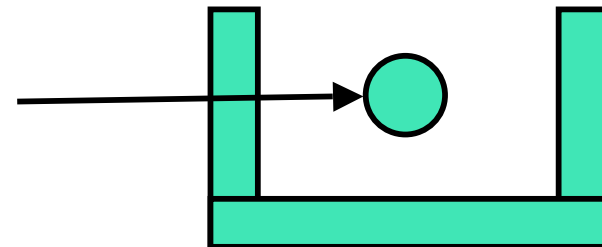
- Start scenario
- EJB and Spring
- Study complications

EJB Version

```
class BarNightSessionBean
    implements SessionBean
void openTab()
void putOnTab(String drink,
    int qty)
    throws CreditException
void closeTab(String creditcard)
void ejbCreate() throws
    CreateException /**/
void setSessionContext(...) /**/
void ejbRemove()
void ejbActivate()
void ejbPassivate()
```

```
public interface BarNightSession
    extends EJBLocalObject
```

```
void openTab();
void putOnTab(String drink,
    int qty)
    throws CreditException;
void closeTab(String s);
```





EJB Version *(Continued)*

```

<?xml version="1.0"
  encoding="UTF-8"?>
<!DOCTYPE ...>
<ejb-jar>
  <enterprise-beans>
    <session>
      <description>
        Drinking a night at a bar
      </description>
      <display-name>
        Bar Night
      </display-name>
      <ejb-name>
        BarNightSessionEJB
      </ejb-name>
      <local-home>
        se.op.bar.service.BarNightSess
        ionHome
      </local-home>

```

```

<local>
  se.op.bar.service.BarNightSess
  ion
</local>
<ejb-class>
  se.op.bar.service.BarNightSess
  ionBean
</ejb-class>
<session-type>Stateful</...>
<transaction-type>
  Container
</transaction-type>
</session>
</enterprise-beans>
</ejb-jar>

```



EJB Version; Code



Problem in EJB

- Class build for container use
- Some extra methods
- Not reusable
 - At least not immediately
 - Perhaps mediately
- Hard to test
 - not impossible



Spring Version

class BarNightImpl implements
BarNight

public void openTab()

public void putOnTab(
String drink,
int qty) throws
CreditException

public void closeTab(
String creditCard)

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="...  
    springframework.org/.../beans"  
        xmlns:xsi="..."  
        xsi:schemaLocation="http://...">  
  
    <!-- This is not a singleton -->  
    <bean id="barNight"  
        class="se.op.bar.impl.BarNightImpl"  
        singleton="false"/>  
</beans>
```



Spring Version; Code





Difference

- Some “extends” and “implements”
- Some extra methods

- Tempting “easy” solution
 - Hack away
 - Does not scale to non-trivial situations
 - Need high-discipline method



Complication 1: Framework Dependency



Domain Driven Design

- Express domain understanding in model
- Refactor towards domain
- Make domain knowledge *explicit* in code

Solution expressed using language of problem

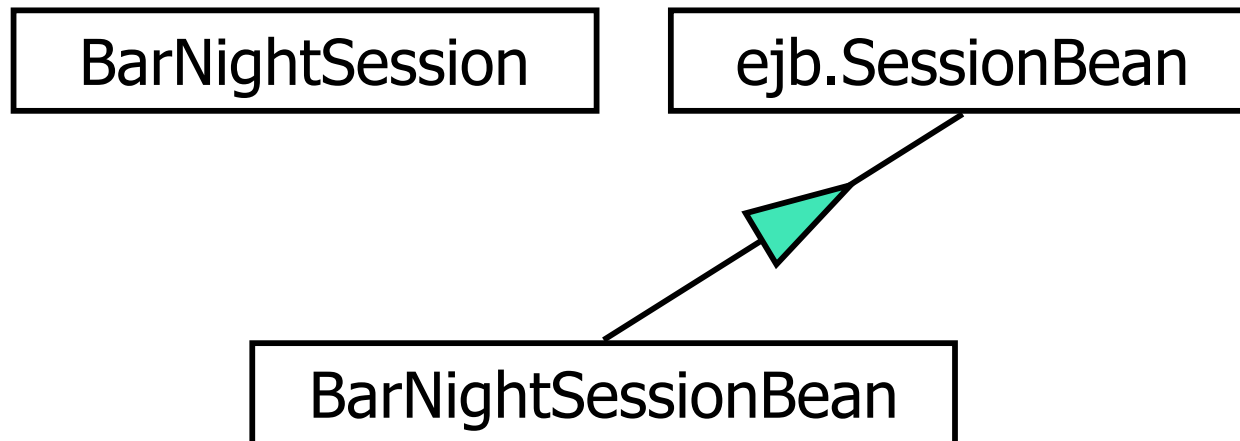


Method

- Extract pure interface (BarNight)
 - Not obviously motivated
 - Helpful
 - Necessary in Spring
- Pull apart implementation
 - domain logic code – framework plugin code
 - by delegation to domain object
 - When to create delegate?

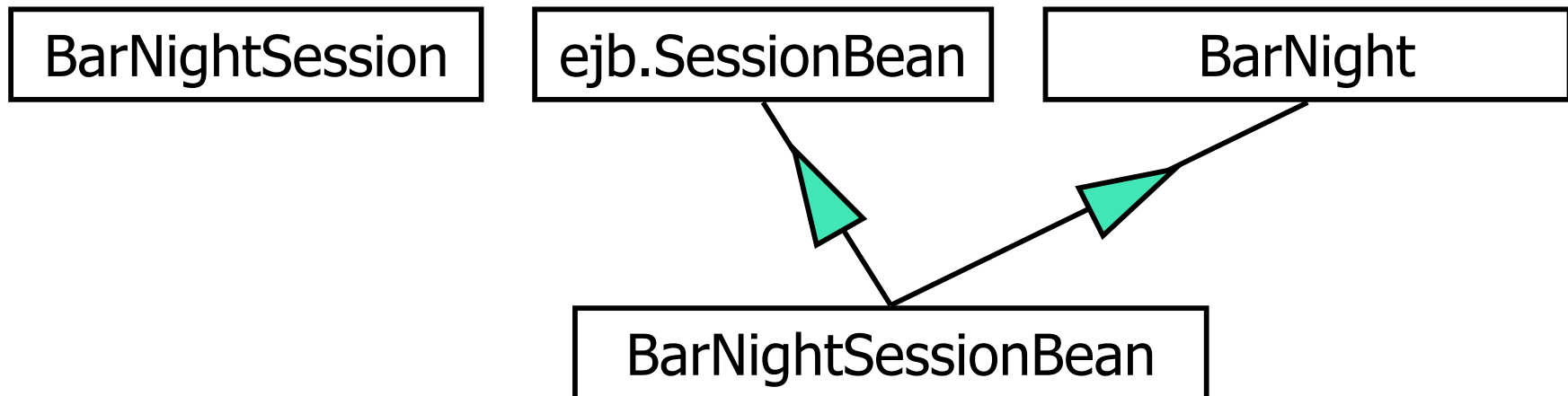


Design “Direct Implementation”

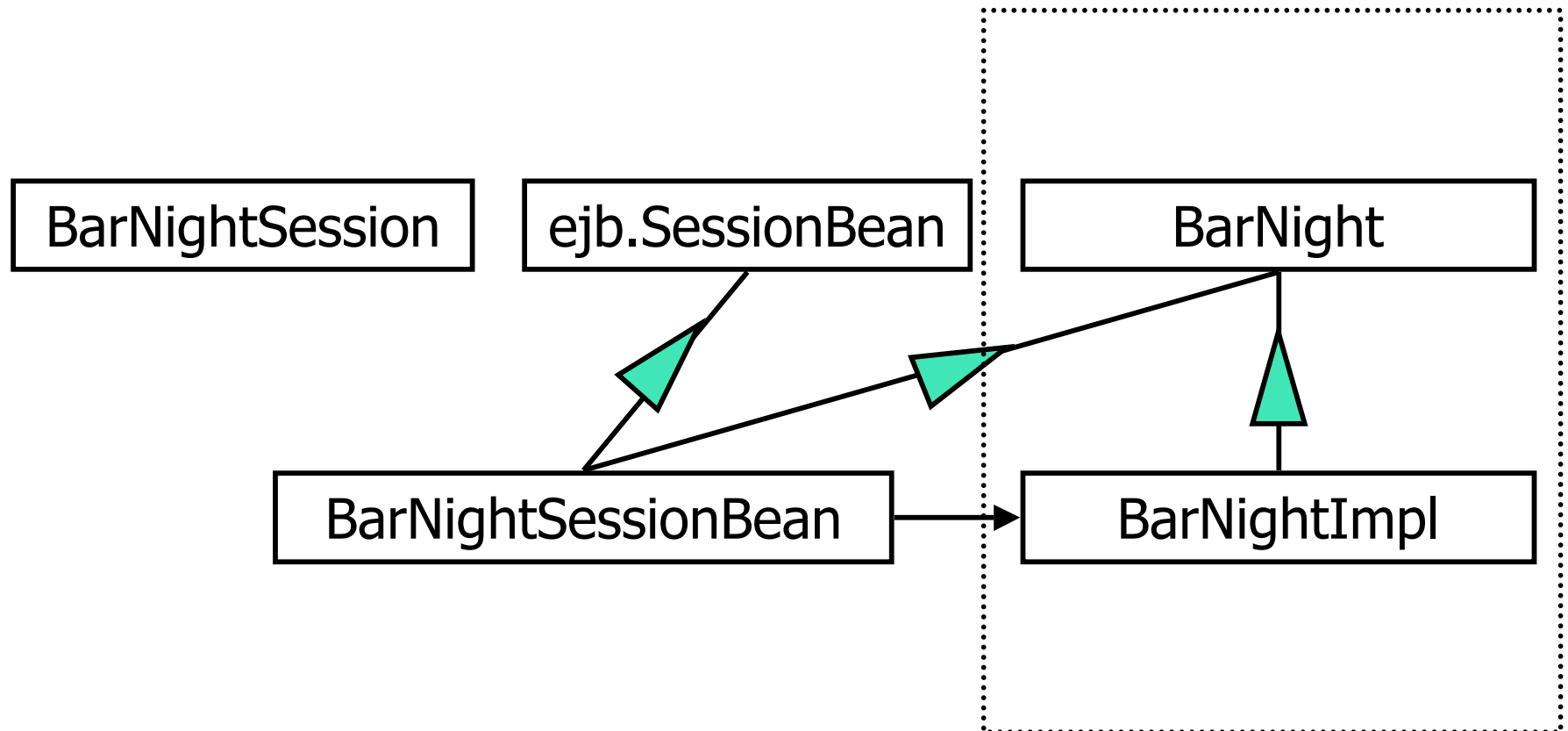




Design – Extract Interface



Design – Delegation (GoF: Proxy)





Demo





Method in Retrospect

- Domain Driven Refactoring
- Extract through *implementation delegation*



Complication 2: Component Dependencies



Ex: Separate Price List

- Price List deployed “on its own”
- Wiring/Plumbing needed
- Coded or declared



EJB Version; Code



Spring Version; Code



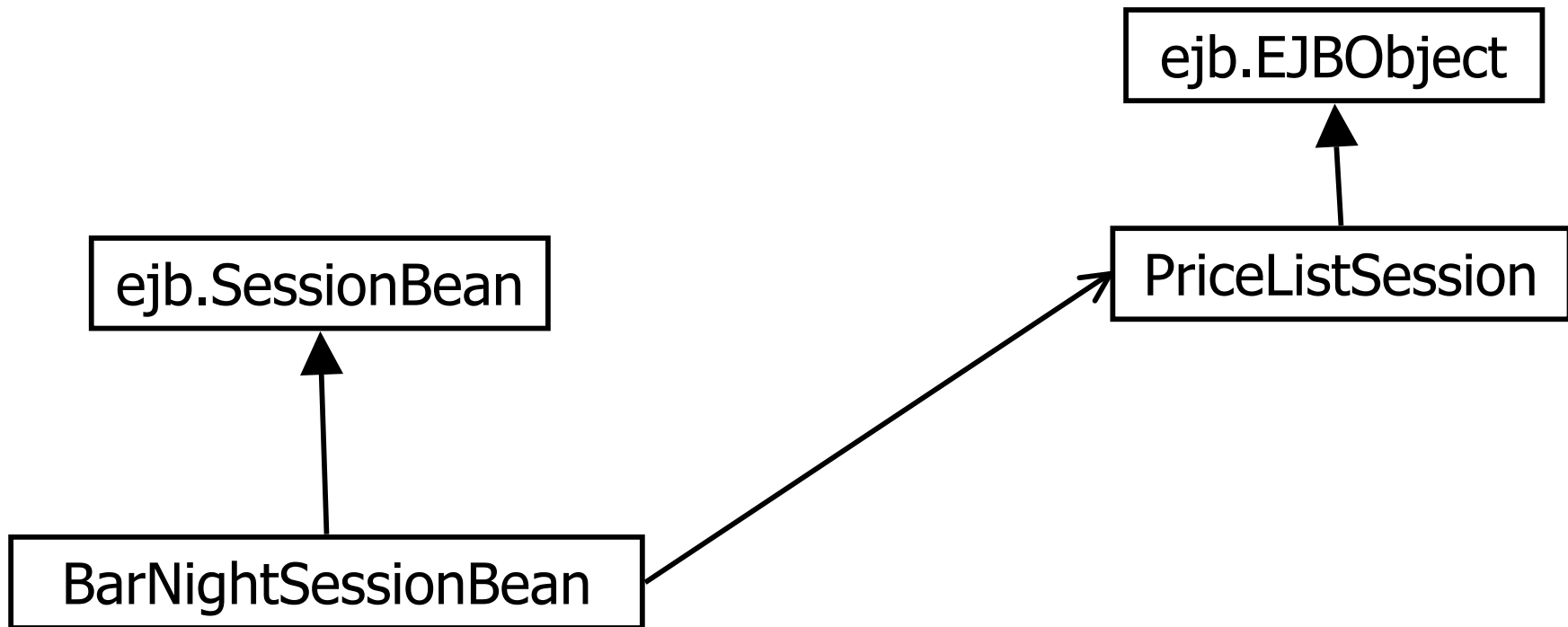


Method

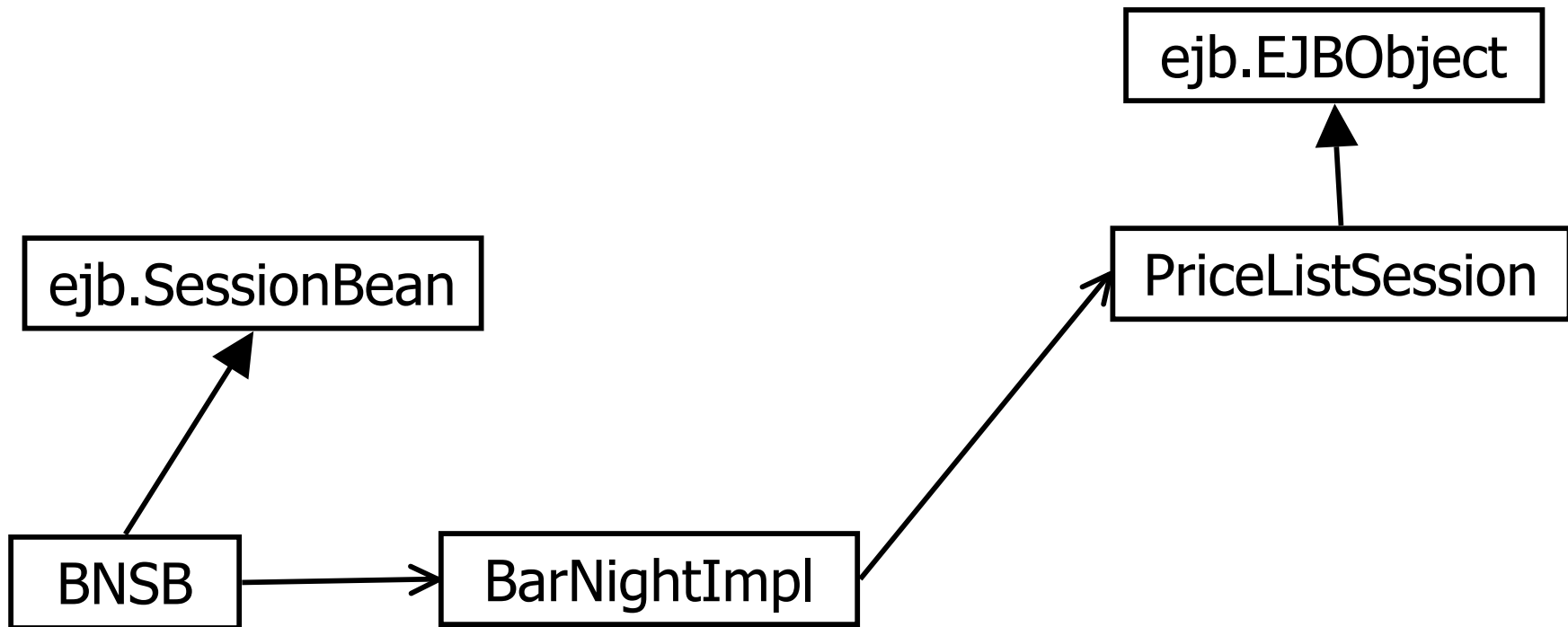
- Plumbing as separate step
 - Make it initial (why?)
- Break away domain logic
 - apply previous good ideas (proxy delegation)
 - Plumbing remains in container class (BNSB)
 - Dependency is injected into BarNightImpl
- Purify domain logic classes
 - Independent of framework



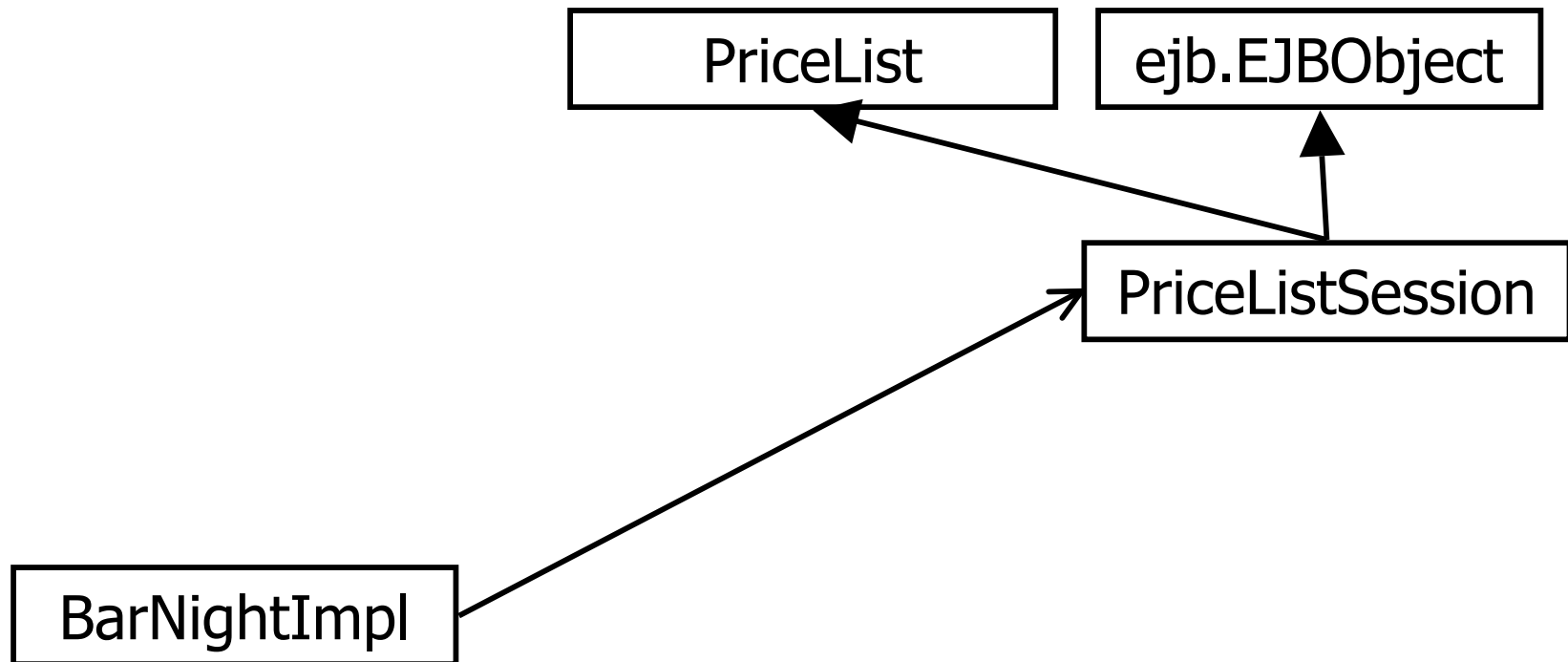
Design



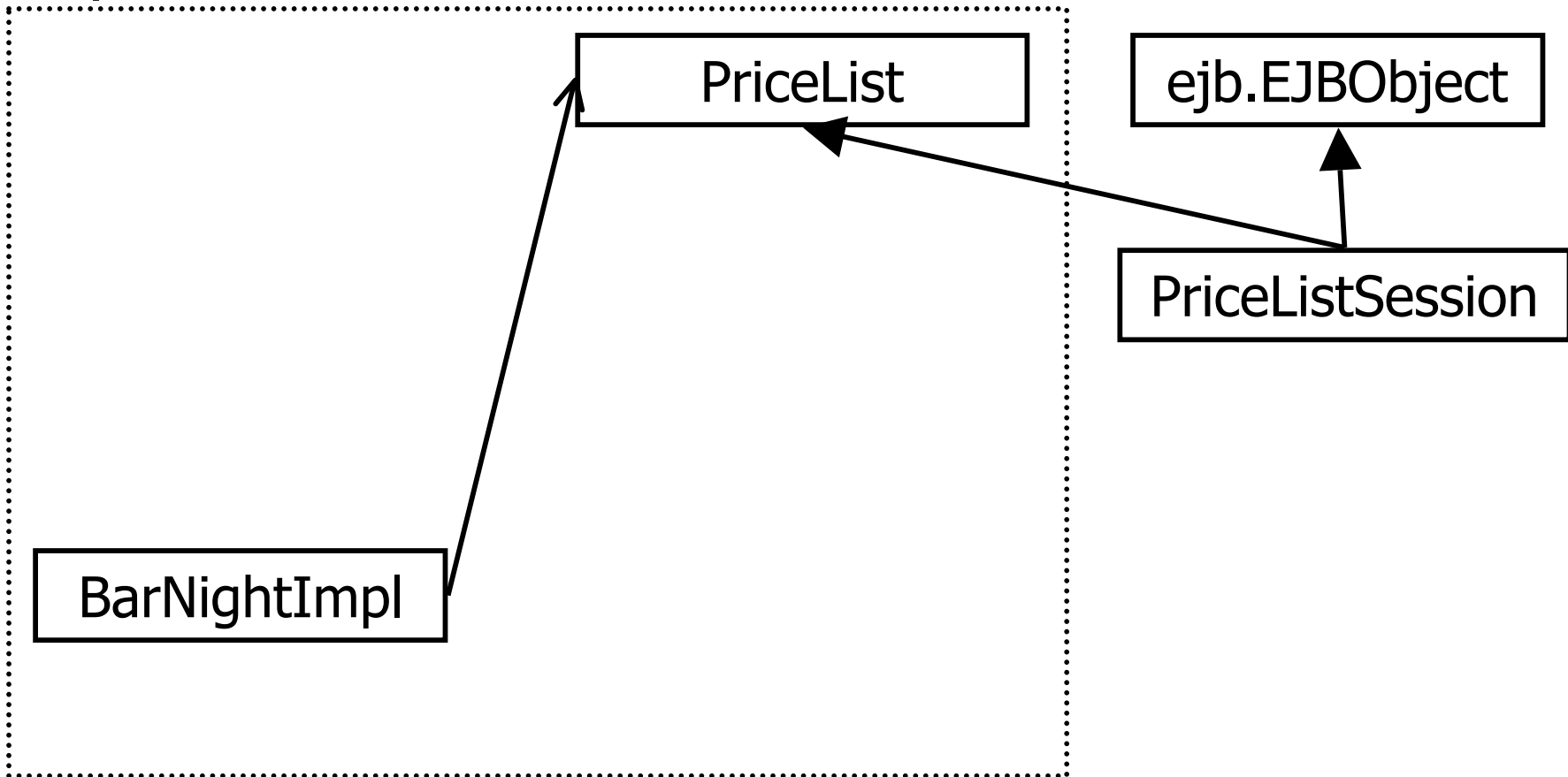
Design – Delegation + Dependency Injection



Design – Purify Interface



Design – Purify Domain Class





Demo





Method in Retrospect

- Separated plumbing logic by extract through implementation delegation
- Separated framework interfaces by extract through *interface inheritance*

- Note on injections:
 - EJB ended up with constructor injection
 - Spring typically use setter injection



Complication 3: Client API



EJB Version; Code



Spring Version; Code





Method

- Separate domain usage from framework code
 - Framework object lookup into factory
 - Factory has domain interface (cfr DDD factory/registry)
- Purify component interface
- Client usage now framework independent

- Result: Framework dependency in factory class
- Framework choice is wiring



Complication 4: Database Connection



Challenges

- Get access to database handle
 - to create connections
- DataSource
 - managed connection factory



EJB Version; Code

- JNDI lookup to get managed datasource from EJB container
- Plumbing through code

```
class BarNightSessionBean ... {  
    void pay(...) {  
        ...  
        Context ctx = new InitialContext();  
        DataSource paymentDs = (DataSource) ctx.lookup(...)  
        ... ds ...  
    }  
}
```



Spring Version; Code

- Plumbing as configuration
 - DataSource declared as bean
 - “Lifted” from application container into Spring world
 - To take advantage of connection pooling
 - JndiObjectFactoryBean
- DI injects datasource into bean that needs db contact
 - Similar to accessing PriceList

```
class BarNightImpl ... {  
    private DataSource paymentDs;  
    public void setPaymentDs(...) {...}
```



Method

- Treat resource lookup as plumbing
- Apply “break away domain logic”
- DataSource remains injected (from framework class)

```
class BarNightImpl ... {  
    ...  
    public BarNightImpl (PriceList priceList,  
                        DataSource paymentDs)  
    ...  
}
```



DataSource and DDD

- DataSource is technical construct
- DDD prefer domain view
- Build domain abstraction component
 - DAO “halfways” (PaymentDAO.insert)
 - Service Object (PaymentService.registerPay)
 - DDD Repository (see “DDD, Eric Evans”)
 - Will encapsule/use DataSource

```
public BarNightIml(PriceList priceList,  
                  PaymentService paymentService)
```



Sketches of Other Complications



Complication: Remote Components

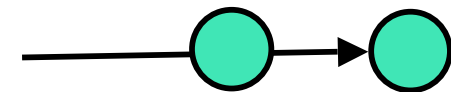
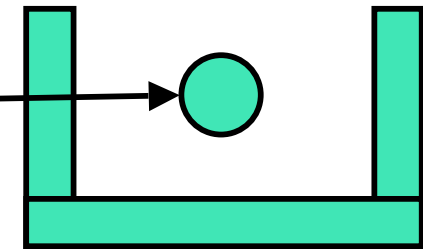
- EJB interface: RemoteException
 - Makes failures explicit
 - Enforces exception handling code (hrm ...)
- EJB class: fail/retry/repair-logic
- Spring: framework support for remote calls
 - Local proxy wired in using DI
 - (turns RemoteExceptions to runtime exceptions)
 - RmiProxyFactoryBean w/ attrib
"serviceInterface"

Complication: Remote Components

- Method:
 - fail-/retry/repair is kind of “network domain” strategy
 - Refactor into interface decorator
 - Use to decorate local proxy
 - Can now be applied in Spring
- Note: can/should be applied to DB failure strategies

Complication: Transaction Support

- EJB:
 - transaction attributes declared in DD
 - classes generated at deploy
- Spring:
 - AOP decoration declared in bean.xml
 - TransactionProxyFactoryBean



- Spring 2.0: @Transactional
- Sometimes things just resolve 😊



Conclusion

- Possible to save application logic
- Domain driven design suitable approach
 - Make implicit knowledge explicit
- Side-effect: testability
- There is Hope!



Comments? Reflections?

- Afterthoughts...

- dan.bergh.johnsson@omegapoint.se

