



**J2EE™ Best Practices  
& Design Patterns**



1

Welcome. This session is about J2EE Best practices and design patterns.



**Sang Shin**  
**[sang.shin@sun.com](mailto:sang.shin@sun.com)**

**Technology Evangelist**  
**Sun Microsystems, Inc.**

2

# Agenda

- ? **Choosing tiers**
- ? **Evolution of Web application frameworks**
- ? **J2EE™ application design strategies**
  - **local versus remote**
  - **persistent strategies**
  - **transaction strategies**
- ? **J2EE™ Design Patterns**
  - **presentation tier**
  - **business tier**

3

So what are we going to talk about? First, we will talk about choosing tiers, specifically when you want to use EJB tier. Then we will go over the evolution of Web application frameworks.

We will then talk about J2EE application design strategies in various areas. Finally we will spend some time talking about design patterns.





# Choosing Tiers



4

So now let's talk about choosing tiers.

## Do You Need an EJB Tier?

- ? **Maybe not, for applications whose main function is reading database tables**
- ? **Yes, if you want to leverage **middleware features** provided by EJB container**
  - **Resource management, concurrency control and threading**
  - **Persistence, transaction and security management**
  - **Messaging (reliability, asynchronous)**
  - **Scalability, Availability**
- ? **Yes, if you want to build **portable and reusable** business components**

5

Many people are asking if they need to use EJB technology. The answer is of course “it depends”. If what the application does is mainly reading database tables in synchronous fashion, there is really no need to use EJB.

Now, as the needs and the complexity of a web-tier only application grow, you, as a designer, have to deal with many middleware system-level issues such as resource management, concurrency control. In many cases, you have to deal with persistence, transaction, security handling. Also web-tier only application cannot deal with asynchronous message. Finally you also have to be concerned about scalability and availability as the number of clients increase.

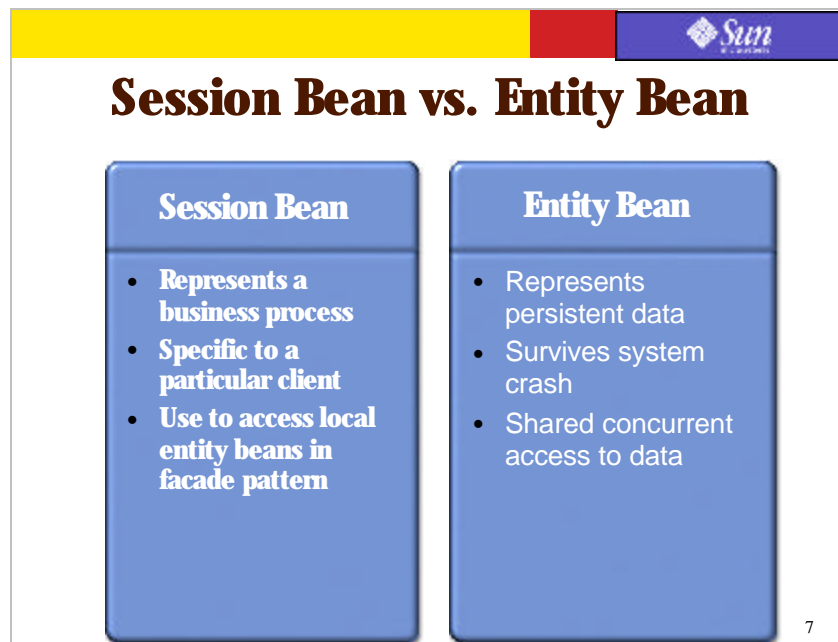
Now under EJB architecture, these middleware features are provided by the container as built-in functions, which means, you as a developer can focus your development effort in building business logics in the form of components.

Another reason you might want to consider using EJB technology is EJB provides better portability and reusability because it is based on component model.



# Choosing Bean Types





First I would like to compare session bean against entity bean. As most of you already know, session bean is a good fit for representing a business process or control while entity bean represents a persistent data which can survive system crash.

Session bean is associated with a particular client while entity bean can be shared by multiple clients.

So how do session bean and entity bean work together? As we will say several times in this presentation, a recommended usage form is that session bean is used as a facade to local entity beans.

## Designing for Client conversation

### ? **Desired properties**

- Managing **client identity**
- Maintaining specific **client state**
- **Short lived: only persistent for duration of conversation**

### ? **Your choices:**

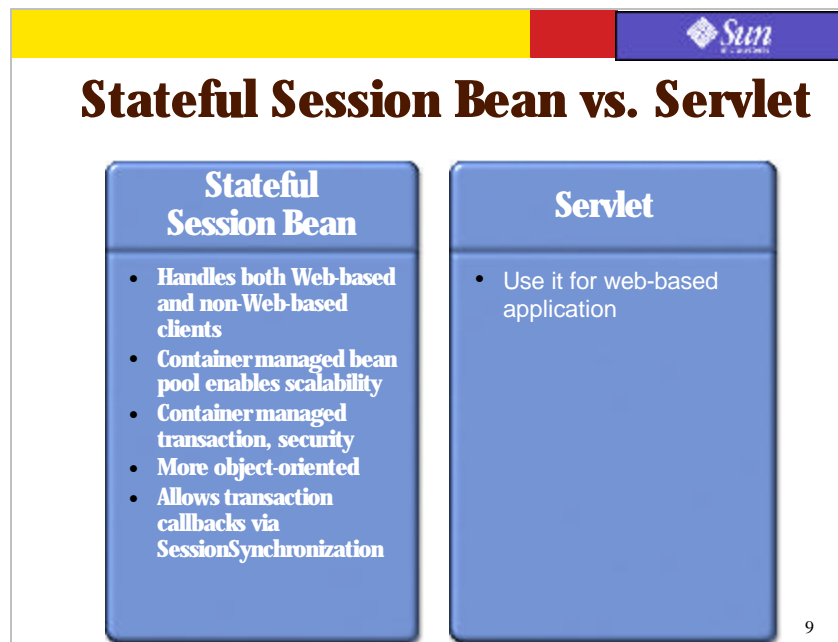
- **Stateful session bean**
- **Stateless session bean with caveat**
- **Servlet with HttpSession object**

8

Now let's say you are trying to design a system in which a client conversation state has to be maintained. That is, client identity as well as client state has to be maintained across multiple method invocations.

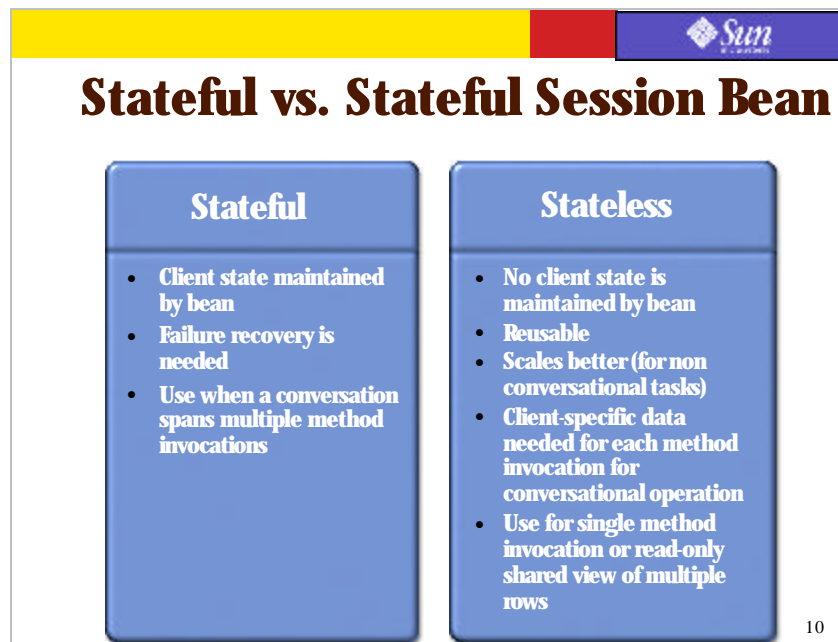
The choice you have include stateful session bean, stateless session bean with some caveat which we will talk about in the following slide. Or you could have a servlet with HttpSession.





In terms of stateful session bean versus servlet, the differences are stateful session bean is object-oriented while servlet is procedure based. Because Stateful session bean is managed by container, it could provide more scalable system.

In a case where you have to deal with both web-based and non-web-based clients, **stateful session**



Now let's compare stateful session and stateless session bean. In stateful session bean, client state is maintained by bean while stateless session bean does not maintain the client-specific state. Because stateless session bean does not have to maintain the state, it can be reused and consequently scales better.

Now if you have to maintain a conversation across multiple invocations and if you want to use stateless session bean, one way you can do is to pass client-specific information on each method call. This approach is not really recommended, however, **since** it could make the client code rather complicated.



So use stateless session bean for single method invocation or read-only operation where client state does not have to be maintained.

## Designing for Event-Driven Interactions


- ? **Desired properties**
  - **Loose coupling** among participants
  - **Asynchronous** communication model
  - **Reliable** communication
  - **Many to many** communication model
- ? **Your choice:**
  - **Message Driven Beans (MDB)**
- ? **Serves as an **asynchronous facade** to a subsystem or an application**

11

If you are designing an application which requires loose coupling among participants and asynchronous communication and reliable communication, then the choice is rather obvious, message driven beans. Message driven beans serve as asynchronous facade to other parts of the application.



# Evolution of Web Application Frameworks



12

Now I would like to spend some time talking about evolution of Web application frameworks.

## **Evolution of MVC Architecture**

**1.No MVC**

**2.MVC Model 1**

**3.MVC Model 2**

**4. Web application framework (based on MVC Model 2)**

? **Struts, Sun ONE Application framework**

**5. Web application framework**

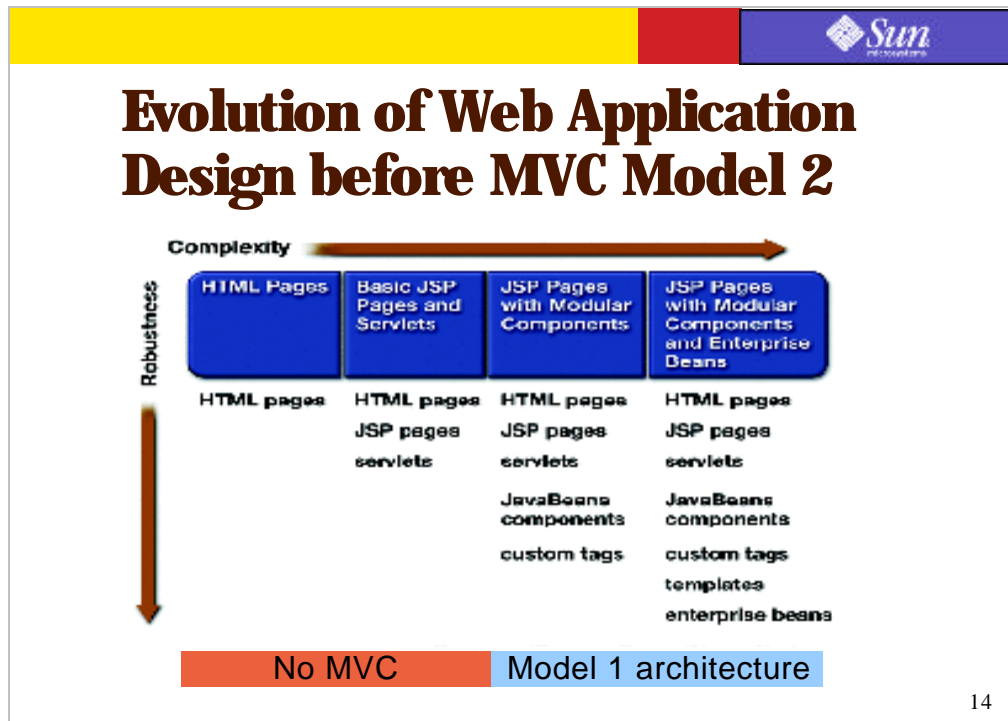
? **JavaServer Faces (JSR-127)**

13

Now when we talk about Web application framework, we are basically talking about the evolution of MVC architecture, which stands for Model, View, and Controller.

So in the beginning, we used no MVC. Then we had Model1 and Model 2 architecture. And people came up with so called Web application frameworks such as Apache Struts based on Model 2 architecture. And finally we are at the phase there will be a standard based Web application framework.

So let's talk about these in a bit more detail.

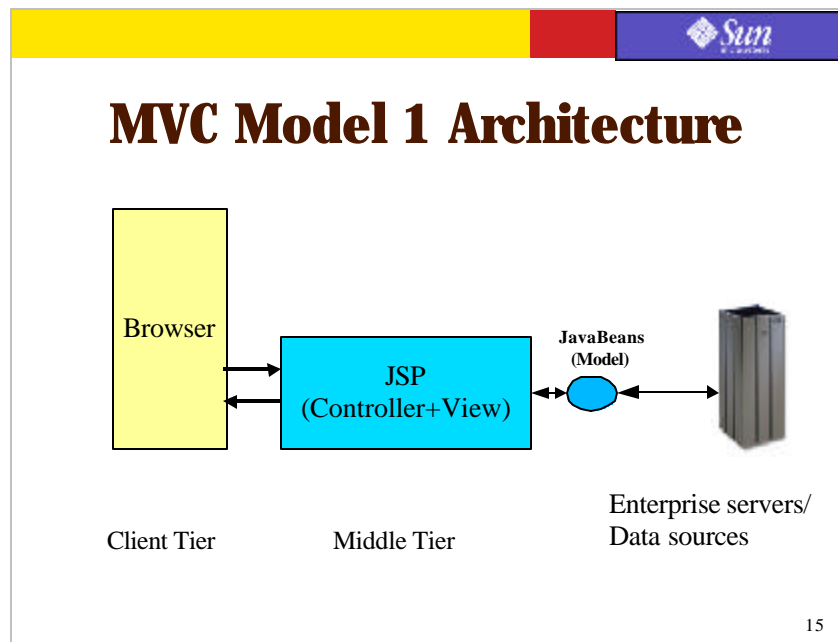


This picture shows the evolution of web application starting from a simplest one which then evolves into more sophisticated and more robust design.

So in the first phase of the evolution, just static HTML pages were used to display static information. Then in the subsequent evolution phase, dynamic contents generation technologies such as CGI initially, then servlet and JSP are introduced to generate and display dynamic contents along with static contents.

When you are using JSP pages, you can use only the basic features that come with it. Or you can leverage more sophisticated features such as component-based dynamic contents generation, for example, leveraging JavaBeans or custom tags, which provide more reusable, more maintainable, more flexible development and deployment options.

Then in a more sophisticated environment, people use so-called template based design or eventually they might want to delegate their business logic processing to EJB tier.



The literature on Web-tier technology in the J2EE platform frequently uses the terms “Model 1” and “Model 2” without explanation. This terminology stems from early drafts of the JSP specification, which described two basic usage patterns for JSP pages. While the terms have disappeared from the specification document, they remain in common use.

Model 1 and Model 2 simply refer to the absence or presence (respectively) of a controller servlet that dispatches requests from the client tier and selects views.

A Model 1 architecture consists of a Web browser directly accessing Web-tier JSP pages. The JSP pages access Web-tier JavaBeans that represent the application model. And the next view to display (JSP page, servlet, HTML page, and so on) is determined either by hyperlinks selected in the source document or by request parameters.

In a Model 1 architecture, view selection is decentralized, because the current page being displayed determines the next page to display. In addition, each JSP page or servlet processes its own inputs (parameters from GET or POST). And this is hard to maintain, for example, if you have to change the view selection, then several JSP pages need to be changed.

In some Model 1 architectures, choosing the next page to display occurs in scriptlet code, but this usage is considered poor form.

## Why MVC Model 2 Architecture?

- ? **What if you want to present different JSP pages depending on the data you receive?**
  - JSP technology alone even with JavaBeans and custom tags (Model 1) cannot handle it well
- ? **Solution**
  - Use Servlet and JSP together (Model 2)
  - Servlet handles initial request, partially process the data, set up beans, then forward the results to one of a number of different JSP pages

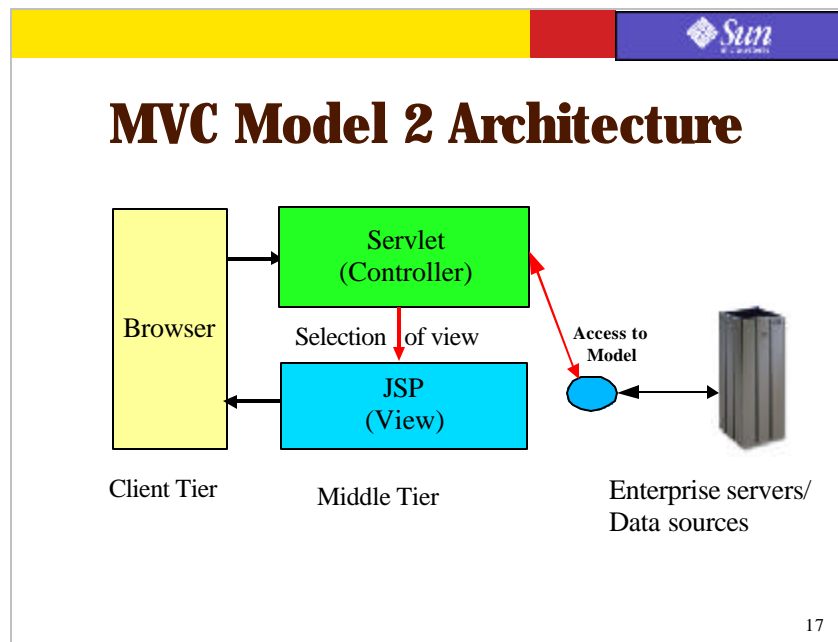
16

I mentioned in previous slide, under Model 1 architecture, a view selection is done by each JSP page. And this poses a maintenance problem.

Now there is another limitation of using Model 1 architecture. In many cases, you want to select JSP pages depending on the data you received from the client. This means there has to be some software entity that handles the processing of the data and then selection of the view. And JSP is not really a good place that you can put this programming logic.

So what is the solution? Model 2 architecture. In Model 2 architecture, both servlet and JSP are used together. In other words, Servlet handles initial request, partially process the data, then forward the results to different JSP pages.





A Model 2 architecture introduces a controller servlet between the browser and the JSP pages.

The controller centralizes the logic for dispatching requests to the next view based on the request URL, input parameters, and application state. The controller also handles view selection, which de-couples JSP pages and servlets from one another.

Model 2 applications are easier to maintain and extend, because views do not refer to each other directly. The Model 2 controller servlet provides a single point of control for security and logging, and often encapsulates incoming data into a form usable by the back-end MVC model.

For these reasons, the Model 2 architecture is recommended for most web applications.

## Web Application Frameworks

- ? **Based on MVC Model 2 architecture**
- ? **Web-tier applications share common set of functionality**
  - **Dispatching HTTP requests**
  - **Invoking model methods**
  - **Selecting and assembling views**
- ? **Provide classes and interfaces that can be used/extended by developers**

18

Now as people are gaining more experience, they found that most Model 2 architecture based web applications share a common set of functionality. For example, they all do receive and dispatch HTTP requests, invoking model methods, selecting and assembling views.

Well, if everybody is doing the same thing over and over every time they write Model 2 based application, then people thought why don't we create a common framework that support these set of functionality so only thing you have to do is basically using or extending the frameworks using common interface and classes.

## Why Web Application Framework?

- ? **De-coupling of presentation tier and business logic into separate components**
- ? **Provides a central point of control**
- ? **Provides rich set of features**
- ? **Facilitates unit-testing and maintenance**
- ? **Availability of compatible tools**
- ? **Provides stability**
- ? **Enjoys community-supports**
- ? **Simplifies internationalization**
- ? **Simplifies input validation**

19

In a more concrete terms, these are the benefits of using a common Web application frameworks. Framework decouples presentation tier from business logic, which would be useful for maintenance and reusability of the code. It provides a central point of control. Popular frameworks also come with other extra features

## Web Application Frameworks

- ? **Apache Struts**
- ? **Sun ONE Application Framework**
- ? **JavaServer Faces (JSR-127)**
  - **Standard-based Web application framework**

20

Now one of the most popular Web application framework is Apache Struts. Sun ONE Application framework has also popular.

Now Java community is almost done with JavaServer Faces (JSR-127) and this will unify all existing web application frameworks such as Struts and Sun ONE application framework into a single standard web application framework.



**J2EE Application  
Design Strategies**



21

Now let's move on and talk about J2EE application design strategies.

## Design Strategies

- ? **Local vs. Remote calls**
- ? **Persistence strategies**
- ? **Transaction strategies**

22

First, let's talk about when to use local interface and when to use remote interface. Second, persistence strategy. Third, transaction strategies.

## Local vs. Remote Calls: Issues

- ? **Performance**
  - **Minimize remote calls**
  - **Network latency, serialization**
- ? **Granularity**
  - **Coarse-grained vs. fine-grained**
- ? **Deployment flexibility**
  - **Deployment in distributed environment**
- ? **Programming models**
  - **Compile time decision not runtime decision (for client)**

23

What are the issues you have to think about when you are deciding whether you want to use local or remote calls.

First, performance. In distributed environment, you want to minimize number of remote calls your make because remote calls are much more expensive than local calls due to network latency and overhead involved in serialization and deserialization.

Second, granularity. When do you want to use coarse-grained and when do you want to use fine-grained calls? The recommendation is that you want to make coarse-grained calls for remote operation while it is OK to use fine-grained calls for local operations since local operations are much less expensive compared to remote calls.

Third, deployment flexibility. Your deployment scheme of your beans might change over time. So how flexible you can deploy your beans could be important.

Finally, programming models. Clients have to make compile time decision on whether they want to use remote calls or local calls because the way programs are written will be different. For example, for local calls, there is no RemoteExceptions.

## Remote Calls

### ? **Advantages**

- **Location independence**
- **Loose coupling between client and bean**
- **Flexibility in distribution of components**

### ? **Disadvantages**

- **Remote calls are more expensive**
- **Handling of RemoteException's**

### ? **Use coarse-grained interfaces for remote beans**

- **Minimize number of remote calls**

24

What are the advantages of remote interface?

First, location independence. Because a remote bean is designed with an assumption that it is going to be called remotely, it can be deployed independent of the location its client. So it has location independence.

Also remote bean does not have to be tightly coupled with its client. Because of the location independence, you can deploy it more flexibly compared to local bean. And this would be important in a deployment environment in which beans should be able to be deployed over multiple EJB containers for the purpose of fault-tolerance or load-balancing.

The obvious disadvantage of remote bean is it is expensive to make a remote call. In fact, this is the reason why local bean concept was introduced in EJB 2.0. Because it is remote operation, the client has to deal with remote exception.

So recommendation is that you want to use remote beans only for coarse-grained operation.



## Local Calls

### ? Advantages

- More **efficient** access due to co-location
- Ability to share data between client and bean through **call by reference**

### ? Disadvantages

- **Tight coupling** of client and bean
- **Less flexibility** in distribution

### ? Use local interface beans (as opposed to remote beans) for **fine-grained operations**

Now local interface. The most distinguished advantage of local bean is its efficiency. Because it is located in the same address space as its client, the cost of method invocation is marginal. And because they are co-located, they can pass the data as “call by reference” semantics, which is more efficient than call by value semantics because you don't have to do serialization and deserialization.

The disadvantage is that the bean is now tightly coupled with its client, thus there is no flexibility in distribution. That is, the local bean has to be co-located with its client.

So the recommendation is that you can use fine-grained interfaces for local beans because the cost is very minimum.

## Recommendations

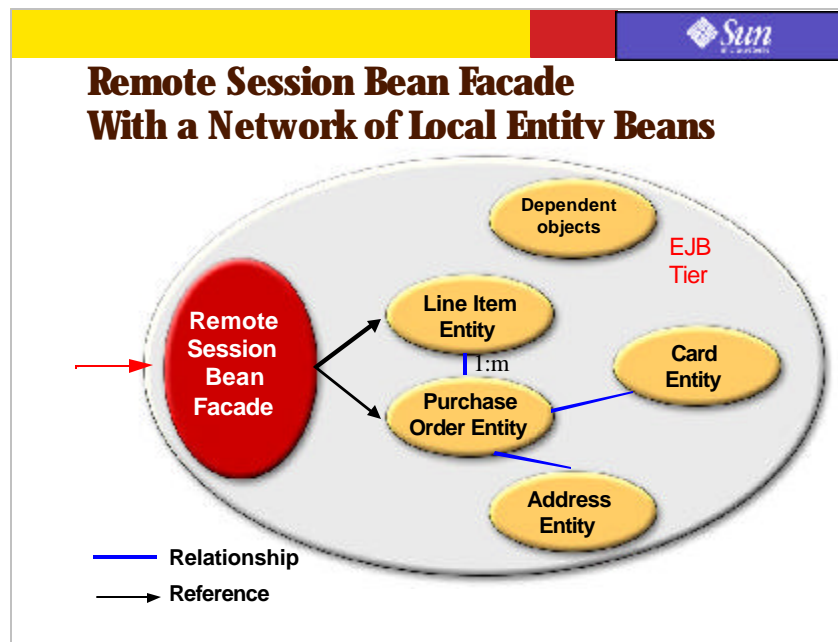
- ? **Use local calls** whenever possible
  - Create islands of local components (local entity beans and their dependent objects)
- ? **Use facade pattern** in which a remote interface session bean (for synchronous operations) or message driven bean (for asynchronous calls) invokes local entity beans
- ? **Use remote call for loose coupling**

26

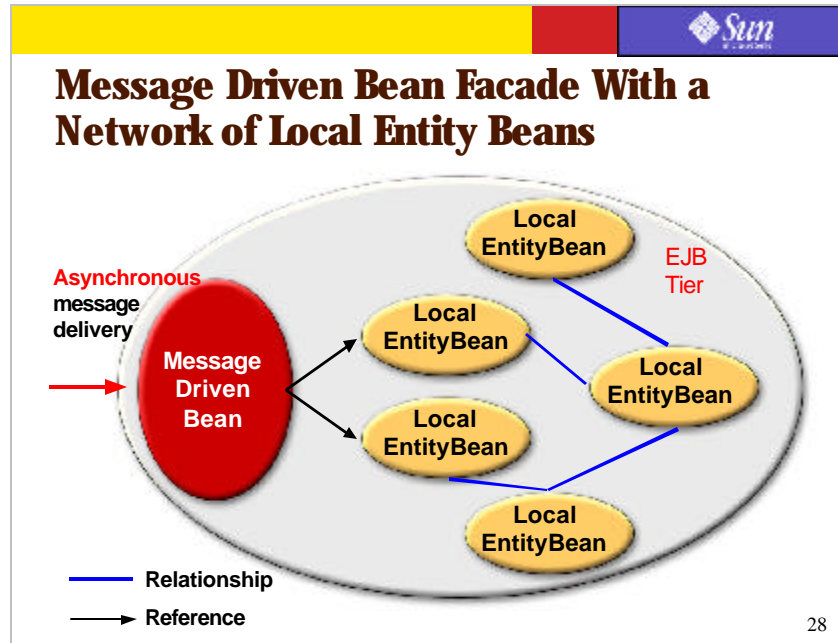
So how do you design your application? In most of the cases, you want to use local interface whenever possible. That is, you can create islands of local beans.

Then create a remote session bean which functions as a facade to these local beans. So the remote session bean has coarse-grained methods while local beans can have fine-grained methods.

If you have a need to support loosely coupled relationship between a client and its bean, then you might want to use remote interface.



So this picture illustrates the recommended architectural model in which session bean is used to communicate with client while within the EJB tier, the communication is via local interface. This provides efficient processing as well as loose coupling between the client and the backend business components.



This picture shows how Message Driven Bean can be used to handle asynchronously delivered messages and play the role of a facade to local entity beans.

## Persistence Strategies: Issues

- ? **Performance**
- ? **Portability**
- ? **Database independence**
- ? **Schema independence**
- ? **Relationship modeling**
- ? **Ease of development**
- ? **Caching**
- ? **Persistence model (O/R mapping)**
- ? **Migration**

29

Now let's talk about persistence strategies. What are the issues that are related with persistence?

First and foremost, performance. The persistence strategy has a great impact to J2EE performance. Next is portability of your code. As we will talk about later on, BMP or even the code based on CMP 1.0 are not really portable.

Next, Database independence. If you are in the business providing beans to your customer, typically you don't know which database your customer will use in advance so database independence is quite important. **Next Schema independence. Database schema do change as the business needs evolve. How can you then design your code so that it gets minimal impact from schema changes?**

Next is relationship modeling. Until CMP 2.0, there was no portable way of describing the relationship among entity beans themselves, which means you have to code manually your self to build the relationship.

Ease of development is always important factor. Caching and persistence modeling also make big difference to performance.

Next, migration from CMP 1.0 to CMP 2.0 could be important factor for you.

## **Persistence Strategies: Choices**

- ? **CMP 2.0**
  - **Entity beans or Dependent objects**
- ? **CMP 1.0**
- ? **BMP**
- ? **Session beans with JDBC or JDO**
- ? **Servlet/JSP with JDBC or JDO**

30

What are the choices you have in terms of persistence? There could be 5 choices.

First, CMP 2.0 which was introduced as part of EJB 2.0. Under CMP, you can model the persistent data either in entity bean or dependent objects.

Other choices are CMP 1.0 and BMP. Or you can have your database access code as part of session bean or even as part of servlet/JSP either by JDBC or JDO.

## Advantages of CMP 2.0 for Component Providers

- ? **Rich modeling capability on relationships**
  - **Referential integrity**
  - **Cardinality**
  - **Cascading delete**
  - **Container manages the relationships not you!**
- ? **Freedom from maintaining interactions with the data store**
- ? **EJB<sup>TM</sup> Query Language (EJB QL)**
- ? **Truly portable code**

31

Now I would like to talk about the advantages of using CMP 2.0. In CMP 2.0, rich modeling capability with relationship is provided to component providers. The benefits of this rich modeling capability include referential integrity, cardinality, cascading delete.

Referential integrity is the assurance that a reference from one entity to another entity is valid. For example, let's say a company, department, and position each have relationships with an employee. If the employee is removed, all references to it must also be removed, or your system must not allow the removal.

Cardinality specifies how many participants are involved in a relationship. There are three flavors of cardinality: one-to-one, one-to-many, and many to many. Again, with CMP 2.0, you can describe these relationships in declaratively manner in the deployment descriptor, which means the burden of managing the relationship is now in the hands of the container now in the hands of the component provider.

Now cascading delete. When you have a relationship between two entity beans, you need to think about whether the relationship is an aggregation or composite relationship.

## Advantages of CMP 2.0 for Container Vendors

- ? **Optimization** is possible because CMP fields are only accessible through their setters and getters
  - Lazy loading
  - Dirty checking
  - Optimistic locking
- ? Optimization is possible in **query operation** because Query is defined in deployment descriptor via EJB QL



## Advantages of BMP

### ? **Dealing with Legacy**

- Database and/or other persistence store
- Familiar mapping tool
- Previously written complex BMP application

### ? **Connecting to Connector driven data stores, ES**

### ? **Complicated operations beyond scope of the EJB 2.0 specification**

- Bulk updates
- Multi object selects
- Aggregates (like sorting)

## Recommendations

- ? **Use CMP 2.0 whenever possible!**
  - **It performs better than BMP**
  - **It improves portability, performance over CMP 1.0**
  - **It is easier to develop and deploy than BMP**
  - **It produces portable code over multiple databases**
  - **There is no reason not to use CMP 2.0 now!**
- ? **If you have to build BMP entity bean, subclass CMP 2.0 bean**
  - **Easy migration to CMP later on**

34

So our recommendation is this. Use CMP 2.0 whenever possible. Because now the containers provide optimized database access and persistence implementation, in most cases, CMP 2.0 provide better performance than BMP.

The portability and performance improvement of CMP 2.0 over CMP 1.0 is very significant. And for those of you who were reluctant to use CMP because of the limitations of CMP 1.0, now you can be assured that CMP 2.0 removes all those limitations. So use CMP 2.0 instead of CMP 1.0.

Of course since you do not have to provide your persistence logic in your code, CMP provides portable and a lot easier to develop. Besides the database access logic code is provided by the container, your code is now database independent.

So the bottom line is this. There is no reason for you not to use CMP 2.0. Now if you have to implement BMP entity bean for some reason, you can subclass CMP 2.0 bean. Because the database access logic in CMP 2.0 are abstract methods, you can implement them in BMP code. This will allow you easy migration from BMP to CMP code later on.

## Transaction Strategies: Issues

- ? **Which transaction style to use?**
  - **Declarative (Container-managed)**
  - **Programmatic (Bean-managed)**
  - **Client-controlled**
- ? **How to detect doomed transaction?**
- ? **How to get notifications for stateful session beans on transactional events?**
  - **Via SessionSynchronization**

35

Now let's talk about transaction. What are the issues related to transaction?

First, which transaction style do you want to use? There are three different ways of performing transaction. First declarative transaction style. In this style, you are specifying your transaction requirement in declarative fashion. This is called container-managed transaction because the actual transactional task is performed by the container. Next, programmatic transaction. In this style, you provide transactional code in your beans. So it is called bean-managed transaction. Finally client code can start and end the transaction code.

Second, how do you detect doomed transaction? What is a doomed transaction? Suppose there are 10 beans that are to be invoked sequentially in a single transaction and let's assume the first bean somehow has to abort the transaction, then it does not make any sense for the rest of those 9 beans to do any transactional task. So it is a good programming practice for those beans to check if the transaction has been aborted or not before performing any business logic tasks.

## Recommendations


- ? **Use declarative transaction for compact code**
  - Use `setRollbackOnly()` method to abort transaction
  - Use `getRollbackOnly()` to detect doomed transaction
- ? **Use programmatic transaction for **fine-grained** transactional control**
  - Complete your transaction in the same method that you began them

36

These are the set of recommendations for transaction. For the same reason you are using CMP over BMP, you want to use declarative transaction for compact code.

Now if you are using declarative transaction, how do you abort a transaction? Use `setRollbackOnly()` method in this case. Throwing an exception does not guarantee the abort of the transaction. In order to detect doomed transaction, use `getRollbackOnly()` method.

Now when do you want to use programmatic transaction? You want to use it when you need a fine-grained transactional control within a method call.



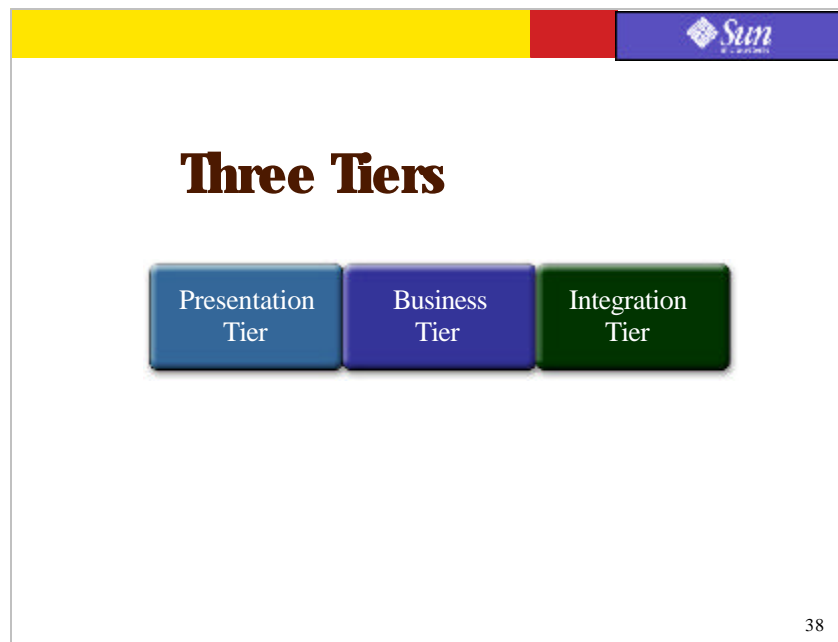
# **Design Patterns**

**(source: Core J2EE Patterns  
written by Alur, Crupi, Malks)**



37

S.



Just to give a quick sense of how a service can be implemented, it can be divided into three tiers – presentation tier, business tier and integration tier.

## Presentation-Tier Patterns

- ? **Intercepting Filter**
- ? **Front Controller**
- ? **View Helper**
- ? **Composite View**
- ? **Service to Worker**
- ? **Dispatcher View**

39

This is the list of patterns identified at the presentation-tier. They include intercepting filter pattern, front controller pattern, view helper pattern, composite view pattern, service to worker pattern, and dispatcher view pattern.

## Business-Tier Patterns

- ? **Business Delegate**
- ? **Service Locator**
- ? **Session Facade**
- ? **Transfer Object**
  - (was Value Object)
- ? **Transfer Object Assembler**
  - (was Value Object Assembler)
- ? **Composite Entity**
- ? **Value List Handler**

40

This is the list of business tier patterns.



## Integration-Tier Patterns

- ? **Connector**
- ? **Data Access Object**
- ? **Service Activator**

41

And integration-tier pattern include data access object pattern and service activator pattern.



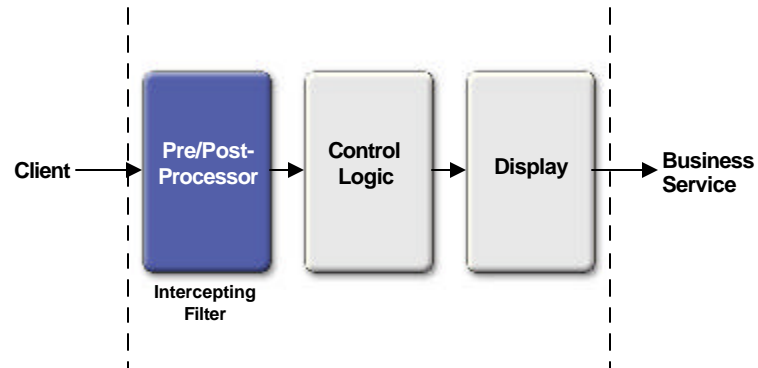
# Presentation-Tier Design Patterns



42

S.

## Presentation Tier Processing



## Intercepting Filter: Forces

- ? **Each service request and response requires common pre-processing and post-processing**
  - logging, authentication, caching, compression, data transformation
- ? **Adding and removing these “pre” and “post” processing components should be flexible**
  - deployment time installation/configuration

44

So the forces behind Decorating Filter pattern...

Common system services are required to execute before and after servicing the request. Centralizing these common services is desirable.

Also such system service components should be easy to add and remove independent of each other. These services typically handle tasks such as logging, authentication, authorization, debugging, transformation of output, compression of data.

## Intercepting Filter: **Solution**

- ? **Create pluggable and chainable filters to process common services such that**
  - **Filters intercept incoming and outgoing requests and responses**
  - **Flexible to be added and removed without requiring changes to other processing code**
- ? **Examples**
  - **Servlet filters** for HTTP requests/responses
  - **Message handlers** for SOAP requests/responses

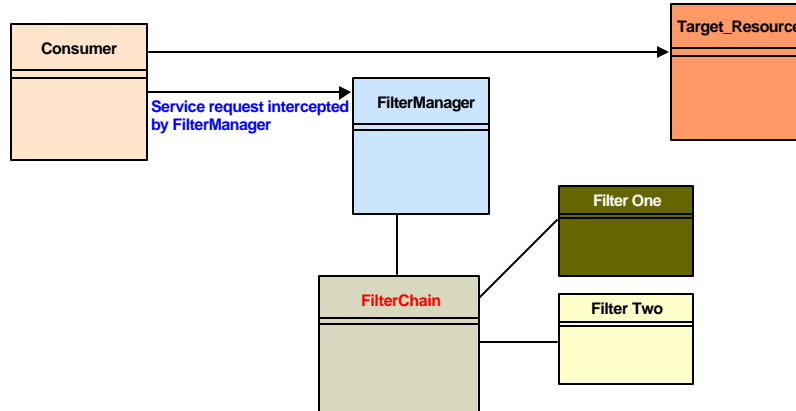
45

These forces can be resolved by introducing pluggable filter components to process common services in a standard manner without requiring changes to the core request processing code. The idea here is to "decorate" our main processing code with the common system services.

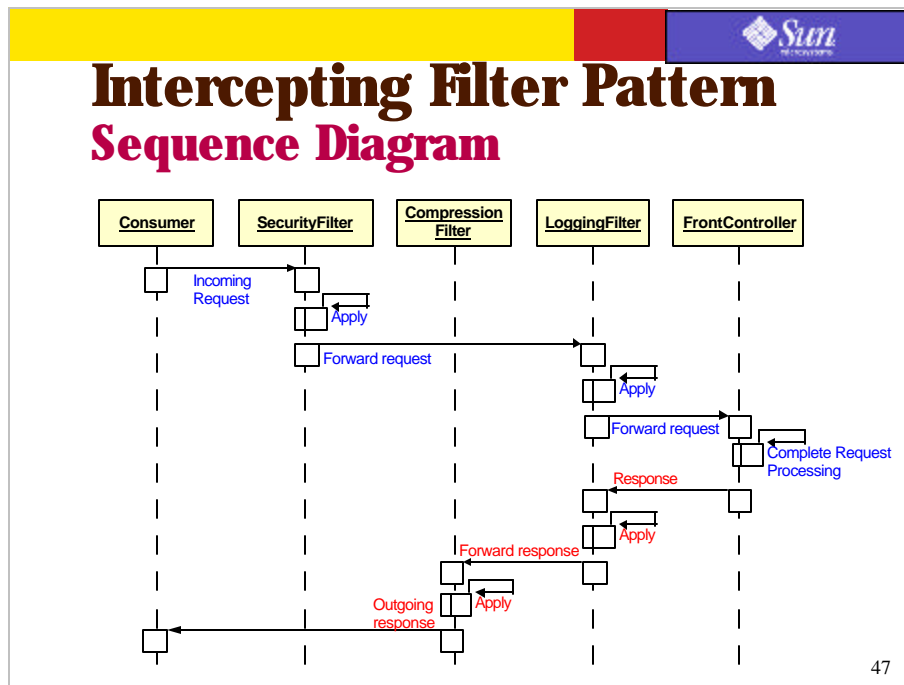
The filters intercept incoming and outgoing requests / responses allowing pre-processing and post-processing. We are able to add and remove these filter components independently without requiring changes to the main processing code, may be in a declarative fashion. One can also set up a chain of filters declaratively in the deployment configuration file. This configuration file can also include specific URIs mapped to this filter chain. Which means that when a consumer requests a resource that matches this configured URI mapping, the filters in the chain are each processed in the order, before the requested resource can be invoked.

This pattern works mainly along with the Front Controller pattern of the ONE architecture.

# Intercepting Filter: Class Diagram



46



This sequence diagram represents a scenario where Filter components are used for pre and post processing of requests and responses.

We have an incoming request from the consumer bounded to a specific service URI. But, before the request can reach its target, it is intercepted by two filters that are configured in a chain in the deployment configuration file. The first filter, Security Filter, performs authentication and authorization checks for the consumer. It can do this either by using, for example, JAAS APIs or can use the container's facilities for performing these functions. This filter may modify the request data and then would forward the request to the second filter, that performs logging functionality. It may use some logging API such as log4j, or J2SE 1.4's inherent logging API to facilitate logging of the service request. Finally, after passing through LoggingFilter, the request reaches FrontController, its intended destination.

Front controller co-ordinates with other components down the service chain and then sends the response back to the consumer. This response is however, intercepted by two filters, before it is sent back to the consumer. The CompressionFilter component can use some compression algorithm to shrink the size of data that is sent on the wire. For example, this filter can compress the XML that goes on the wire, for performance gains. Finally, the response is sent to the consumer.

# Intercepting Filter Pattern

## Sample code for writing Servlet 2.3 Filter

```
Public final class SecurityFilter implements Filter{
    public void doFilter(ServletRequest req,
                        ServletResponse res,
                        FilterChain chain)
        throws IOException, ServletException{
        // Perform security checks here
        ....

        // Complete the filter processing by either passing the control
        // to the next servlet filter in chain or to the target URI .
        chain.doFilter(modified_req, modified_res);
    }
}
```

```
<filter-mapping>
  <filter-name>SecurityFilter</filter-name>
  <servlet-name>ControllerServlet</servlet-name>
</filter-mapping>
```

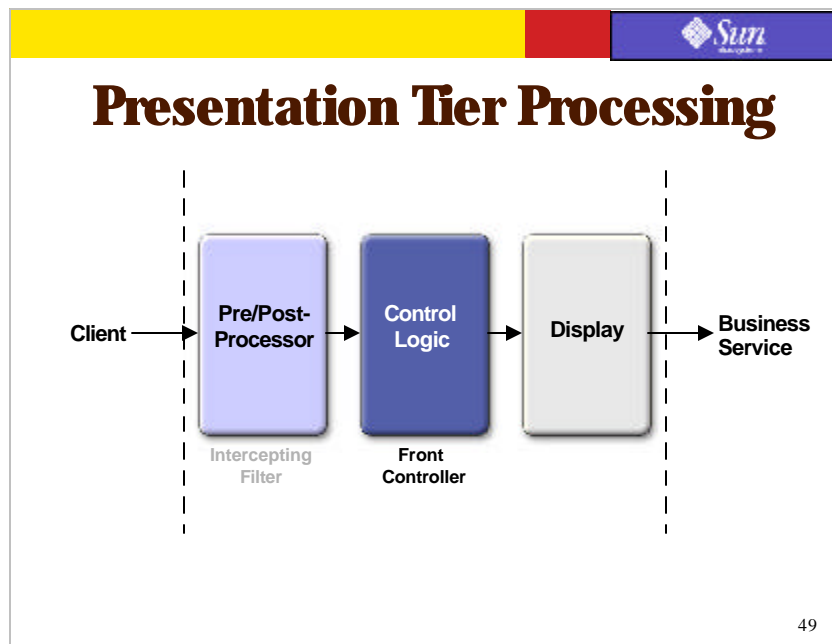
48

This is the sample code of implementing a servlet filter based on Servlet 2.3 specification.

The filter class should implement a Filter interface that defines init, destroy and doFilter methods. The doFilter() is the method of interest particularly. It does the filter processing and either calls the next filter in chain or passes the control to the target URI.

On the bottom part of the slide, there is a fragment of a deployment descriptor of the web application. Here we specify that SecurityFilter would intercept each incoming request directed to resource ServletController.





- ? Pre-Processor includes: - Decode
  - Transform
  - Compress
- ? Control includes:
- ? Request Handling, Command Processing and View Navigation

## Front Controller: Forces

- ? **Multiple views** are used to handle a similar business request
  - View is required to handle logic for content retrieval and navigation
- ? **Common system services** are rendered to each request
  - Example: Authentication, authorization, Logging

50

So the forces behind Front Controller pattern...

There are multiple views corresponding to different types of consumers and all these views typically handle the same type of service request. The view is also required to handle logic for content retrieval and navigation apart from generating user interface supported by the end consumer.

Also for each incoming and outgoing request, a common set of system services such as authentication/authorization, logging, etc. are required to be performed. Now if the consumers were allowed to directly access the views, then each of these view would also require to provide logic for these system services. This leads to a redundancy of system service logic among multiple views that handle the same kind of request. At the same time, this leads to a very complex view component since it now handles content retrieval, view management, navigation and system services functions.

## Front Controller: Solution

- ? **Use a controller as an centralized point of contact for all requests**
  - Promote code reuse for invoking system services
- ? **Can have multiple front controllers, each mapping to a set of distinct services**
- ? **Works with other patterns**
  - Command, Dispatcher, View Helper

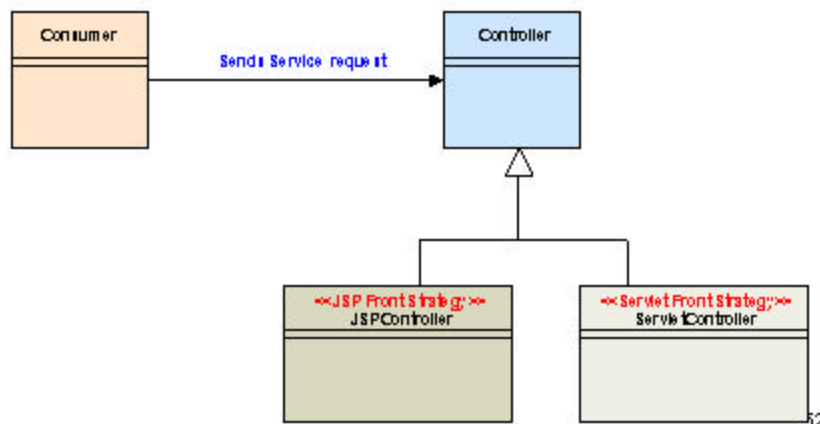
51

These forces can be resolved by introducing a Front controller component that provides a centralized entry point for all incoming requests and probably outgoing responses. This would promote code reuse at the system services level. Controller component would invoke system services such as authentication, authorization, logging and error handling appropriately for each request. It can do so by relying on other components such as Filter components or can include the logic itself, if it is not too complex. Controller then delegates the request to other components in the service chain for further request processing or view selection or content creation.

This pattern does not limit the number of Front Controller components to just one. In fact, a design can have multiple controller components each acting as an entry point for a related set of services. For example a design can have two front controllers; one mapped as an entry point for servicing audio processing requests and another front controller can be mapped as an entry point for all requests regarding online content management.

Front Controller works in co-ordination with other patterns such as Dispatcher View, Decorating Filter, View Helper as we would see in the sequence diagram for this pattern.

## Front Controller: Implementation Strategy



There are different ways of implementing Front controller component. FrontController can be implemented as a Servlet or a JSP. However implementing front controller in a JSP is not recommended since it requires a web developer to work with markup page that has nothing to do with controller logic and is oriented towards formatting display.

We can have a base controller in case of multiple controllers in the design, and then this base controller can then be inherited by the other controllers.

# Front Controller Sample Code

## Servlet-based Implementation

```
Public class EmployeeController extends HttpServlet{  
  
    //Initializes the servlet  
    public void init(ServletConfig config) throws ServletException{  
        super.init(config);  
    }  
    //Destroys the servlet  
    public void destroy(){}  
  
    //Handles the HTTP GET Requests  
    protected void doGet(HttpServletRequest request,  
                          HttpServletResponse response)  
        throws ServletException, java.io.IOException{  
        processRequest (request, response);  
    }  
  
    //Handles the HTTP POST Requests  
    protected void doPost(HttpServletRequest request,  
                          HttpServletResponse response)  
        throws ServletException, IOException{  
        processRequest (request, response);  
    }  
}
```

# Front Controller Sample Code

## Servlet Front Controller with Command Pattern

```
//Processes requests for HTTP Posts and Gets
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException{
    String page;

    // Create a RequestHelper object that represent the client request specific information
    RequestHelper reqHelper = new RequestHelper(request);

    /*****
    * Create a Command object. Command object is an implementation of the Command
    * Pattern. Behind the scenes, implementation of getCommand() method would be like
    * Command command = CommandFactory.create(request.getParameter("op"));
    *****/
    CommandHelper cmdHelper = reqHelper.getCommand();

    // CommandHelper performs the actual operation
    page = cmdHelper.execute(request, response);

    // Dispatch control to the view
    dispatch(request, response, page);
}
```

## Front Controller Sample Code

### Servlet Front Strategy with Dispatch Pattern

```
//Implement the dispatch method
protected void dispatch(HttpServletRequest request,
                        HttpServletResponse response, String page)
                        throws ServletException, IOException {
    RequestDispatcher dispatcher
        = getServletContext().getRequestDispatcher(page);
    dispatcher.forward(request, response);
}
```





# Business-Tier Design Patterns



56

S



## Business Delegate Pattern: Forces

- ? **Business service interface (Business service APIs) change as business requirements evolve**
- ? **Coupling between the presentation tier components and business service tier (business services) should be kept to minimum**
- ? **It is desirable to reduce network traffic between client and business services**

57

So the forces behind Business Delegate pattern...

Business requirements evolve over time as a result of which business service interfaces also change. Now if the components in Service Interaction Layer, mainly in the formk of helper components, interact directly with the core service layer components. This direct interaction increases coupling between a business service component and the client component. As a result, these components are vulnerable to changes in the implementation of business services.

This direct interaction also exposes a lot of business service implementation specific details such as distributed nature of core service components, to the client components.

Additionally there may be a detrimental impact on the network performance if client components, that use the business service components, make too many invocations over the network. This can happen if client components use the service API directly with no client side caching mechanism such as value objects, or coarse grained aggregating service.

## **Business Delegate Pattern:** **Solution**

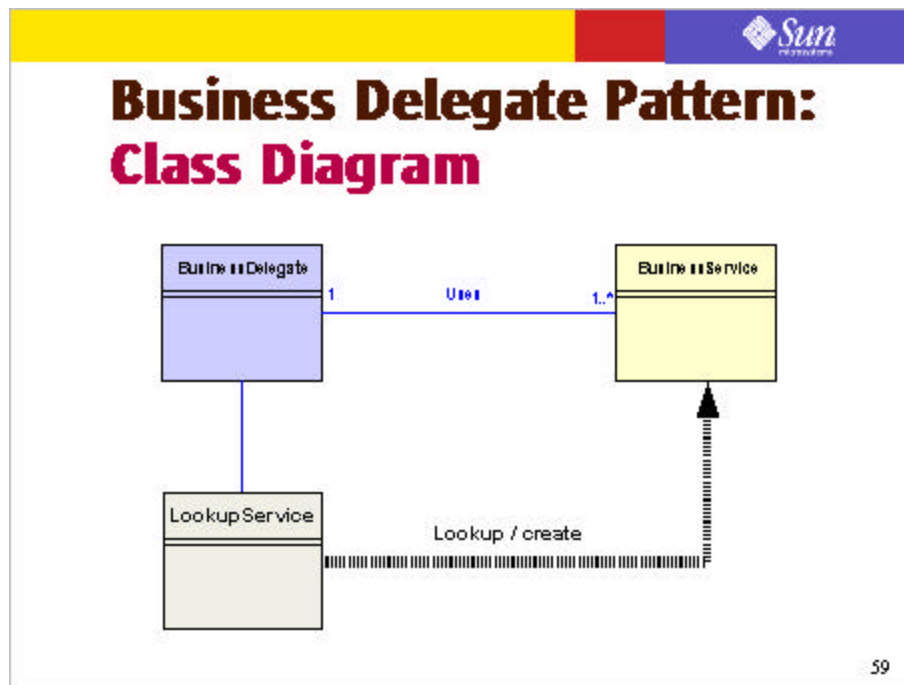
- ? **Use a Business Delegate to**
  - **Reduce coupling between presentation-tier and business service components**
  - **Hide the underlying implementation details of the business service components**
  - **Cache references to business services components**
  - **Cache data**
  - **Translate low level exceptions to application level exceptions**

58

These forces can be resolved by introducing a Business Delegate component that acts as a client side business abstraction thus hiding the implementation details of the business services. This reduces coupling between interaction layer components and business services. This potentially reduces changes in the client components that can result from changes in the business service API. However, business delegate component may need to change, whenever the underlying business service interface changes.

A business delegate component can potentially hide the details of underlying service. For example, the client component can be totally transparent of the naming and lookup mechanisms, exception handling mechanisms that are involved in using a business service component implemented as an Enterprise Java Beans. The implementation of business delegate component would deal with these intricacies of using business service components.

Business Delegate pattern can work with Service Locator pattern.



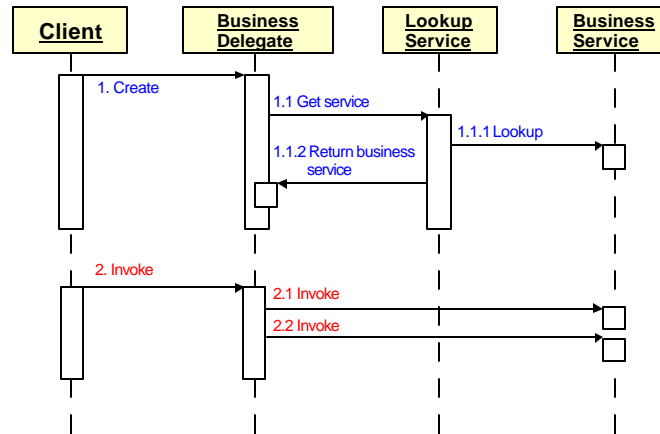
A Business Delegate uses a component called the Lookup Service. The Lookup Service is responsible for hiding the underlying implementation details of “looking up” a business service. The lookup service can be written as part of business delegate however it is recommended to create a separate component. Lookup service component is often an implementation of Service Locator pattern.

Business Service components can be an Session or entity bean providing the required service to the client.

Business Delegate component creates and uses multiple business service components. Often this logic of interacting with multiple business service components, might be encapsulated in a session facade, in which case there exist a 1-1 relationship between business delegate and session facade.

# Business Delegate Pattern

## Sequence Diagram



60

This sequence diagram represents a scenario where a client uses business delegate component as an higher level abstraction for multiple underlying service objects. The business delegate uses lookup service to locate and create the required service objects. The client then invokes business methods on the Business delegate which in turn co-ordinates with other finer grained business service objects to get the work done.

## Business Delegate Pattern Implementation Strategies

### ? Delegate Adapter Strategy

- Integrating two **disparate** systems require an **Adapter**
  - ? Adaptor changes XML request to native request
- Perfectly fits for **B2B environments that use XML**

### ? Delegate Proxy Strategy

- Business Delegate **proxies** to the Session bean it is encapsulating
- May **cache** necessary data such as **home or remote object handles** to improve performance

## Business Delegate Pattern

### Sample Code using Delegate Proxy

```
public class ResourceDelegate{

    // Reference to Session Facade
    private ResourceSession objResourceSession;

    // Session facade's home object class
    private static final Class homeClass
        = myExamples.resourceSession.ResourceSessionHome.class;

    // Default constructor. Looks up Session facade home and creates a new one.
    public ResourceDelegate() throws ResourceException{
        try{
            ResourceSessionHome resourceSessionHome =
                (ResourceSessionHome)ServiceLocator.getInstance().getHome(
                    "Resource", homeClass);
            objResourceSession = resourceSessionHome.create();
        }catch(ServiceLocatorException ex){
            //Translate ServiceLocator Exception into an Application Exception
            throw new ResourceException(...);
        }
        ...
    }
}
```

62

# Business Delegate Pattern

## Sample Code using Delegate Proxy

```
// Another constructor that accepts a Handle ID and reconnects to the a priori
// obtained session bean instead of creating new one
public ResourceDelegate(String id) throws ResourceException{
    super();
    reconnect(id);
}

// Method to reconnect using EJB Handle
public void reconnect(String id) throws ResourceException{
    try{
        //Obtain an instance of ServiceLocator object
        ServiceLocator objServiceLocator = ServiceLocator.getInstance();

        // Obtain the service that corresponds to the given ID. Each ID
        // corresponds to serialized EJBObject handle
        objResourceSession
            = (ResourceSession)ServiceLocator.getService(id);
    }catch(ServiceLocatorException ex){
        //Translate the Remote Exception into an Application Exception
        throw new ResourceException(...);
    }
}
```

# Business Delegate Pattern

## Sample Code using Delegate Proxy

```
// Business methods proxied to the Session Facade. If any service exception arises, these methods
// convert them into application specific exceptions such as ResourceException, SkillSetException, etc.
public ResourceVO setCurrentResource(String resourceId) throws ResourceException{
    try{
        return objResourceSession.setCurrentResource(resourceId);
    }catch(RemoteException ex){
        throw new ResourceException(...);
    }
}

public ResourceVO getResourceDetails() throws ResourceException{
    try{
        return objResourceSession.getResourceDetails();
    }catch(RemoteException ex){
        throw new ResourceException(...);
    }
}

//Remaining proxy methods to session facade ResourceSession
...
}
```



## Service Locator Pattern: Forces

- ? **Service lookup and creation involves complex interfaces and network operations**
- ? **Service lookup and creation operations are resource intensive and redundant**

65

So the forces behind Service Locator pattern...

Clients of business services which are typically implemented as EJB and JMS component, need to locate and create these service components first. For example, client of an EJB must first locate the EJB's home object which the client then uses to find object, create or remove one or more EJBs. Similarly a JMS client must first locate a JMS connection factory to further create JMS connections or JMS sessions. Thus, for using any service component, the client needs to use JNDI services. Which means that JNDI code for looking up and creating the services in all these clients is the same. This is a huge amount of duplication of code.

Also creating a JNDI initial context object and performing a lookup on an EJB Home object utilizes significant resources. If multiple clients repeatedly require the same bean home object then such duplicate copies of home object can negatively impact the application performance.

Also implementation of context factories that create initial context objects is vendor specific. This introduces vendor dependency in the client components that need to lookup the EJBs or JMS components.

## **Service Locator Pattern:** **Solution**

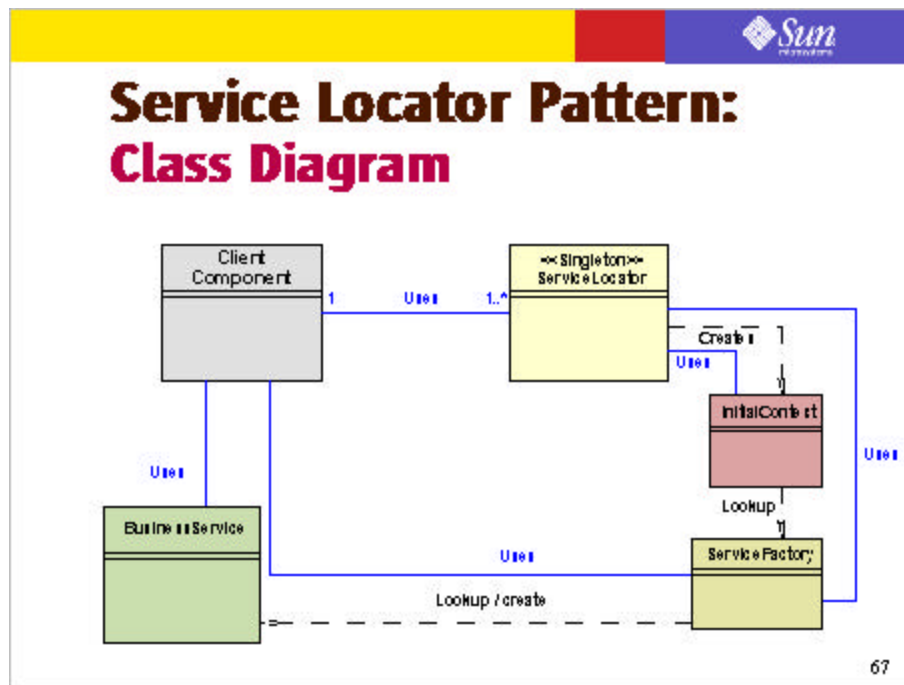
- ? **Use a Service Locator to**
  - **Abstract naming service usage**
  - **Shield complexity of service lookup and creation**
  - **Promote reuse**
  - **Enable optimize service lookup and creation functions**
- ? **Usually called within Business Delegate object**

66

So time to resolve these forces...

Introduce a service locator component. So what would it do? It would abstract all the JNDI naming service usage and would hide the complexities of creating Initial Context objects, looking up EJB Home objects, and creating and recreating EJB objects. This would boost reuse of the code because now multiple clients would reuse the same object for performing their lookup/creation operations. Also, service locator implementation can optimize these operations by caching Home objects, for example. So now, client component's code is less complex since it doesn't deal with using naming service, doesn't deal with writing against APIs for looking up and creating services.

This pattern works normally with Business Delegate, Session Facade and Value Object Assembler patterns.



This diagram shows relationships between objects in this pattern. Let us examine the responsibilities of each of these objects.

The client component mostly is the Business Delegate component.

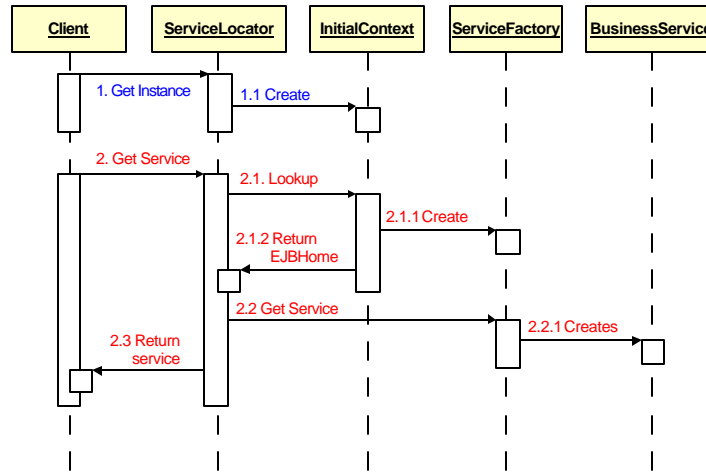
InitialContext is the starting point in any lookup operation. A naming service would start looking up a service from this context onwards.

A ServiceFactory object manages the lifecycle of the Business service objects. For example, EJBHome is service factory for EJB objects.

A BusinessService is the actual component that the client actually wants to access. Business service is created, removed or looked up by the service factory. It is either an EJB or JMS component.

So let us examine a scenario where these objects interact, in the next slide.

# Service Locator Pattern: Sequence Diagram



68

## **Service Locator Pattern: Implementation Strategies**

### **? Implementation strategies for Service Locator**

- **EJB Service Locator Strategy**
- **JMS Queue Service Locator Strategy**
- **JMS Topic Service Locator Strategy**
- **Combined EJB and JMS Service Locator Strategy**

69

Locators can be implemented separately based on the kind of business service it is looking up and creating. For example we can specifically have EJB service locators or JMS Queue/Topic service locators.

Service locators are implemented using this strategy in designs which deal with a certain kind of business service objects all the time. For example, in a design that does not involve JMS at all, there is no need to make the service locator JMS aware.

However, locators can be implemented to lookup and create both kinds of business services, if required.

Service Locators are typically implemented using Singleton pattern.

# Service Locator Pattern

## Sample code using EJB Service Locator

```
public class ServiceLocator
{
    private static ServiceLocator me;
    InitialContext context = null;

    private ServiceLocator() throws ServiceLocatorException{
        try{
            context = new InitialContext();
        }catch(NamingException ex){
            throw new ServiceLocatorException(...);
        }
    }

    // Returns the instance of ServiceLocator class (singleton)
    public static ServiceLocator getInstance() throws ServiceLocatorException{
        if (me == null){
            me = new ServiceLocator();
        }
        return me;
    }
}
```

## Service Locator Pattern: Sample code using EJB Service Locator Strategy

```
// Convert the given string into EJBHandle and then to EJBObject
public EJBObject getService(String Id) throws ServiceLocatorException{
    if (Id == null){
        throw new ServiceLocatorException(...);
    }

    try{
        byte[] bytes = new String(Id).getBytes();
        InputStream io = new ByteArrayInputStream(bytes);
        ObjectInputStream is = new ObjectInputStream(io);
        javax.ejb.Handle handle = (javax.ejb.Handle)is.readObject();
        return handle.getEJBObject();
    }catch(Exception ex){
        throw new ServiceLocatorException(...);
    }
}

// Returns the string Id that represents the given EJBObject's handle
// in serialized format
public String getId(EJBObject session) throws ServiceLocatorException{
    ...
}
```

# Service Locator Pattern

## Sample code using EJB Service Locator Strategy

```
// Converts the serialized string into EJBHandle and then to EJBObject
public EJBObject getHome(String name, Class homeClass)
    throws ServiceLocatorException{
    try{
        Object objRef = context.lookup(name);
        EJBHome home
            = (EJBHome)PortableRemoteObject.narrow(objRef,
                                                    homeClass);
        return home;
    }catch(NamingException ex){
        throw new ServiceLocatorException(...);
    }
}

// Other methods pertaining to getting service using a string ID or
// getting a string ID based on the given service
...
}
```



# Service Locator Pattern

## Client code using EJB Service Locator

```
public class SampleServiceLocatorClient{
    public static void main(String[] args){
        ServiceLocator objServiceLocator = ServiceLocator.getInstance();
        try{
            ResourceSessionHome objResourceSessionHome
                = (ResourceSessionHome)objServiceLocator.getHome(
                    "ejb/ResourceSession",
                    myExamples.resourceSession.ResourceSessionHome.class);
        }catch(ServiceLocatorException ex){
            // Client handles exception
            ...
        }
    }
}
```





# Resources



74

S

## Resources on J2EE

- ? **J2EE Home page**
  - <http://java.sun.com/j2ee>
- ? **EB Home page**
  - <http://java.sun.com/products/ejb>
- ? **J2EE Blueprints**
  - <http://java.sun.com/j2ee/blueprints>
- ? **Articles and books on J2EE best practices and design patterns**
  - [http://www.javapassion.com/j2eeindex0.html#Best\\_practice](http://www.javapassion.com/j2eeindex0.html#Best_practice)



# Q & A



76

S