Core J2EE Patterns
Best Practices and Design Strategies

Olivier Liechti
Java Architect
Sun Microsystems

Université de Fribourg
5 janvier 2004
olivier.liechti@sun.com
“A pattern is an idea that has been useful in one practical context and will probably be useful in others.”

Martin Fowler, 1997
Objectives of the presentation

• Focus on the **design** of J2EE systems
  – distributed enterprise systems
  – higher level than J2EE APIs (implementation)

• Present a catalog of J2EE **patterns**
  – Give an overview of the whole catalogue
  – Study some of the patterns in details

• Share experience from the **field**
  – Best practices and recommendations
  – Tools, frameworks used in the industry
Assumptions

• Knowledge of the J2EE platform
  – goals, high-level architecture, roles
• Basic knowledge of J2EE technologies
  – servlets, JSPs, EJBs, JDBC, JNDI, ...
• Skills in Object-Oriented design
  – reuse, encapsulation, coupling, cohesion
• Familiarity with design patterns (GoF)
Agenda

- Introduction
- The patterns catalog
  - Overview
  - Presentation tier patterns
  - Business tier patterns
  - Integration tier patterns
- Best practices and tools
- Conclusions
Introduction

- Motivation
- Patterns
- Software qualities
- Basic principles
Motivation

- J2EE is powerful, but complex!
- Learning the J2EE APIs is not enough!
  - programmer vs. developer vs. architect
- Learning how to design J2EE systems
  - is more difficult
  - is not an exact science
  - is a process that requires experience
- Design patterns make the *continuous* learning process easier!
Patterns...

• are observed through experience
• are written in a structured format
• prevent reinventing the wheel
• exist at different levels of abstraction
• undergo continuous improvement
• are reusable artifacts
• communicate designs and best practices
• can be used together to solve a larger problem
Enterprise system qualities

- Maintainability and flexibility
- Scalability and high-availability
- Portability
- Reliability and manageability
- Reusability and modularity
- Performance and efficiency
- Testability
Enterprise system qualities

- Application
- Virtual Platform
- Upper Platform
- Lower Platform
- Hardware
- Availability
- Scalability
- Manageability
- Maintainability
- Security
Core design principles

- Information hiding and encapsulation
  - Isolate clients from implementation details
  - Isolate clients from complexity

- Low coupling
  - Clear boundaries between tiers
  - Program against interfaces, not classes

- High cohesion
  - Separate presentation, business, persistence

- Don't forget the network!
  - Minimize round-trips, minimize data transfer
Definitions

- **Modularity**: the extent to which software is divided into components, called modules, which have high internal cohesion, low coupling between each other, and simple interfaces.
- **Cohesion**: a measure of the extent to which related aspects of a system are kept together in the same module, and unrelated aspects are kept out.
- **Coupling**: a measure of the extent to which interdependencies exist between software modules.
- **Encapsulation**: creating a module to contain some algorithm or data structure, thus hiding its details behind the module's interface. Allows changes to code to be more easily made since one can be confident that 'outsiders' are not relying on too many details.
- **Information hiding**: hiding details so as to reduce complexity.
- **Portability**: the ability for software to be run in a variety of different hardware or software environments with no or minimal changes.

http://www.site.uottawa.ca:4321/oose/index.html
Core J2EE Patterns

• Part 1: Patterns and J2EE
  – Introduction
  – Presentation tier design considerations
  – Business tier design considerations
  – Integration tier design considerations
  – J2EE refactorings

• Part 2: J2EE Pattern Catalog
  – Overview
  – Presentation tier (8)
  – Business tier (9)
  – Integration tier (4)

• Epilogue
  – Micro-architecture
Pattern template

- Problem
- Forces
- Solution
  - Structure (class and sequence diagrams)
  - Strategies (lower abstraction level)
- Consequences
- Sample code
- Related patterns
Five Tier Model (logical)

- **Client Tier**: Browsers, applets, fat clients and other UIs
- **Presentation Tier**: JSP, Servlets and other UI elements
- **Business Tier**: EJBs and other Business Objects
- **Integration Tier**: JMS, JDBC, Connectors and Legacy
- **Resource Tier**: Databases, external systems and legacy resources
Five Tier Model (logical)

- **Client Tier**
  Browsers, applets, fat clients and other UIs

- **Presentation Tier**
  JSP, Servlets and other UI elements

- **Business Tier**
  EJBs and other Business Objects

- **Integration Tier**
  JMS, JDBC, Connectors and Legacy

- **Resource Tier**
  Databases, external systems and legacy resources

- **Core J2EE Patterns**

- **Mini CyberCoach**
  EJBs + Bus. Delegates
## Five Tier Model (physical 1)

<table>
<thead>
<tr>
<th>Tier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Tier</td>
<td>Browsers, applets, fat clients and other UIs</td>
</tr>
<tr>
<td>Presentation Tier</td>
<td>JSP, Servlets and other UI elements</td>
</tr>
<tr>
<td>Business Tier</td>
<td>EJBs and other Business Objects</td>
</tr>
<tr>
<td>Integration Tier</td>
<td>JMS, JDBC, Connectors and Legacy</td>
</tr>
<tr>
<td>Resource Tier</td>
<td>Databases, external systems and legacy resources</td>
</tr>
</tbody>
</table>

A developer can run the entire system on his laptop...

Be careful not to forget the network!!
Five Tier Model (physical 2)

- **Client Tier**
  Browsers, applets, fat clients and other UIs

- **Presentation Tier**
  JSP, Servlets and other UI elements

- **Business Tier**
  EJBs and other Business Objects

- **Integration Tier**
  JMS, JDBC, Connectors and Legacy

- **Resource Tier**
  Databases, external systems and legacy resources

The Web container and the EJB container may run on the same machine. They may even run in the same process.
Five Tier Model (physical 3)

Client Tier
Browsers, applets, fat clients and other UIs

Presentation Tier
JSP, Servlets and other UI elements

Business Tier
EJBs and other Business Objects

Integration Tier
JMS, JDBC, Connectors and Legacy

Resource Tier
Databases, external systems and legacy resources

The Web container and the EJB container may run on different machines. A firewall may even be placed between them.
Deployment architecture

- **Load balancer**
- **Firewall**
- **HTTP Server**
  - Horizontal scalability
- **Web Container**
  - Horizontal scalability
- **EJB Container**
  - J2EE cluster, horizontal & vertical scalability
- **Database**
  - DB cluster, vertical scalability
The pattern catalog

Intercepting Filter
Front Controller
Context Object
Application Controller
View Helper
Composite View
Service To Worker
Dispatcher View

Business Delegate
Service Locator
Session Façade
Application Service
Business Object
Composite Entity
Transfer Object
Transfer Object
Assembler
Value List Handler

Data Access Object
Service Activator
Domain Store
Web Service Broker
Intercepting Filter intercepts incoming requests and outgoing responses and applies a filter. These filters may be added and removed in a declarative manner, allowing them to be applied unobtrusively in a variety of combinations. After this preprocessing and/or post-processing is complete, the final filter in the group vectors control to the original target object. For an incoming request, this is often a Front Controller, but may be a View.

Front Controller is a container to hold the common processing logic that occurs within the presentation tier and that may otherwise be erroneously placed in a View. A controller handles requests and manages content retrieval, security, view management, and navigation, delegating to a Dispatcher component to dispatch to a View.

Application Controller centralizes control, retrieval, and invocation of view and command processing. While a Front Controller acts as a centralized access point and controller for incoming requests, the Application Controller is responsible for identifying and invoking commands, and for identifying and dispatching to views.

Context Object encapsulates state in a protocol-independent way to be shared throughout your application. Using Context Object makes testing easier, facilitating a more generic test environment with reduced dependence upon a specific container.

View Helper encourages the separation of formatting-related code from other business logic. It suggests using Helper components to encapsulate logic relating to initiating content retrieval, validation, and adapting and formatting the model. The View component is then left to encapsulate the presentation formatting. Helper components typically delegate to the business services via a Business Delegate or an Application Service, while a View may be composed of multiple subcomponents to create its template.

Composite View suggests composing a View from numerous atomic pieces. Multiple smaller views, both static and dynamic, are pieced together to create a single template. The Service to Worker and Dispatcher View patterns represent a common combination of other patterns from the catalog. The two patterns share a common structure, consisting of a controller working with a Dispatcher, Views, and Helpers. Service to Worker and Dispatcher View have similar participant roles, but differ in the division of labor among those roles. Unlike Service to Worker, Dispatcher View defers business processing until view processing has been performed.

Business Delegate reduces coupling between remote tiers and provides an entry point for accessing remote services in the business tier. A Business Delegate might also cache data as necessary to improve performance. A Business Delegate encapsulates a Session Façade and maintains a one-to-one relationship with that Session Façade. An Application Service uses a Business Delegate to invoke a Session Façade.

Service Locator encapsulates the implementation mechanisms for looking up business service components. A Business Delegate uses a Service Locator to connect to a Session Façade. Other clients that need to locate and connect to Session Façade, other business-tier services, and web services can use a Service Locator.
Session Façade provides coarse-grained services to the clients by hiding the complexities of the business service interactions. A Session Façade might invoke several Application Service implementations or Business Objects. A Session Façade can also encapsulate a Value List Handler.

Application Service centralizes and aggregates behavior to provide a uniform service layer to the business tier services. An Application Service might interact with other services or Business Objects. An Application Service can invoke other Application Services and thus create a layer of services in your application.

Business Object implements your conceptual domain model using an object model. Business Objects separate business data and logic into a separate layer in your application. Business Objects typically represent persistent objects and can be transparently persisted using Domain Store.

Composite Entity implements a Business Object using local entity beans and POJOs. When implemented with bean-managed persistence, a Composite Entity uses Data Access Objects to facilitate persistence.

The Transfer Object pattern provides the best techniques and strategies to exchange data across tiers (that is, across system boundaries) to reduce the network overhead by minimizing the number of calls to get data from another tier.

The Transfer Object Assembler constructs a composite Transfer Object from various sources. These sources could be EJB components, Data Access Objects, or other arbitrary Java objects. This pattern is most useful when the client needs to obtain data for the application model or part of the model.

The Value List Handler uses the GoF iterator pattern to provide query execution and processing services. The Value List Handler caches the results of the query execution and return subsets of the result to the clients as requested. By using this pattern, it is possible to avoid overheads associated with finding large numbers of entity beans. The Value List Handler uses a Data Access Object to execute a query and fetch the results from a persistent store.

Data Access Object enables loose coupling between the business and resource tiers. Data Access Object encapsulates all the data access logic to create, retrieve, delete, and update data from a persistent store. Data Access Object uses Transfer Object to send and receive data.

Service Activator enables asynchronous processing in your enterprise applications using JMS. A Service Activator can invoke Application Service, Session Façade or Business Objects. You can also use several Service Activators to provide parallel asynchronous processing for long running tasks.

Domain Store provides a powerful mechanism to implement transparent persistence for your object model. It combines and links several other patterns including Data Access Objects.

Web Service Broker exposes and brokers one or more services in your application to external clients as a web service using XML and standard web protocols. A Web Service Broker can interact with Application Service and Session Façade. A Web Service Broker uses one or more Service Activators to perform asynchronous processing of a request.
Presentation Tier Patterns

- **Intercepting Filter**
- **Front Controller**
- **Context Object**
- **Application Controller**
- **View Helper**
- **Composite View**
- **Service To Worker**
- **Dispatcher View**

**Facilitates pre-processing and post processing of a request.**

**Useful for security checks, auditing, caching, compression, etc.**

**A chain of pluggable independent filters**
Presentation Tier Patterns

- Intercepting Filter
- **Front Controller**
- Context Object
- Application Controller
- View Helper
- Composite View
- Service To Worker
- Dispatcher View

---

**Provides a centralized controller for managing the handling of a request.**

**The entry point for the system. Should not be too fat (delegates work to Application Controller)**

**Processing a request:**
1. Protocol handling and context transformation
2. Navigation and routing
3. Core processing
4. Dispatch
Presentation Tier Patterns

- Intercepting Filter
- Front Controller
- Context Object
- Application Controller
- View Helper
- Composite View
- Service To Worker
- Dispatcher View

Encapsulate state in a protocol-independent way to be shared within your application.

Application components should not have to know HTTP. Instead, they should call getXXX() on a context object.
Presentation Tier Patterns

- Intercepting Filter
- Front Controller
- Context Object
- Application Controller
- View Helper
- Composite View
- Service To Worker
- Dispatcher View

Centralizes and modularizes action and view management.

Who should service this request, and who should display the results?

Processing a request:
1. Protocol handling and context transformation
2. Navigation and routing
3. Core processing
4. Dispatch

Struts
Presentation Tier Patterns

- Intercepting Filter
- Front Controller
- Context Object
- Application Controller
- **View Helper**
- Composite View
- Service To Worker
- Dispatcher View

Encapsulates logic that is not related to presentation formatting into Helper components.

No programming logic in the views!
Presentation Tier Patterns

- Intercepting Filter
- Front Controller
- Context Object
- Application Controller
- View Helper
- Composite View
- Service To Worker
- Dispatcher View

Presentation Tier Patterns

- View Helper
- Composite View
- Service To Worker
- Dispatcher View

Front Controller + Application Controller + View Helper

Processing a request:
1. Protocol handling and context transformation
2. Navigation and routing
3. Core processing
4. Dispatch

Combines a dispatcher component with the Front Controller and View Helper patterns.
Presentation Tier Patterns

- Intercepting Filter
- Front Controller
- Context Object
- Application Controller
- View Helper
- Composite View
- Service To Worker
- Dispatcher View

Combines a dispatcher component with the Front Controller and View Helper patterns, deferring many activities to View processing.
Presentation Tier Patterns

- Intercepting Filter
- Front Controller
- Context Object
- Application Controller
- View Helper
- Composite View
- Service To Worker
- Dispatcher View
Intercepting Filter (1)

• **Problem**
  – You want to intercept and manipulate a request and a response before and after the request is processed.

• **Forces**
  – common processing across requests should be centralized (auditing, encoding, compression)
  – pre- and postprocessing components should be loosely coupled with request handling services
  – pre- and postprocessing components should be independent of each other and self contained
Intercepting Filter (2)

• Solution
  – Use an Intercepting Filter as a pluggable filter. A filter manager combines loosely coupled filters in a chain. You can add, remove and combine filters without changing code.

Client: sends a request to the FilterManager
FilterManager: manages filter processing
FilterChain: ordered collection of independent filters
Filter: does pre- postprocessing task
Target: resource requested by the client
Intercepting Filter (3)
Intercepting Filter (4)

• Strategies
  – *Standard Filter Strategy (Servlet 2.3)*
  – Custom Filter Strategy
  – Base Filter Strategy
  – Template Filter Strategy
Intercepting Filter (5)

```java
public class BaseEncodeFilter implements javax.servlet.Filter {
    private javax.servlet.FilterConfig filterConfig;
    public void doFilter(
        javax.servlet.ServletRequest servletRequest,
        javax.servlet.ServletResponse servletResponse,
        javax.servlet.FilterChain filterChain)
            throws java.io.IOException,
            javax.servlet.ServletException {
            filterChain.doFilter(servletRequest, servletResponse);
        }
    protected javax.servlet.FilterConfig getFilterConfig() {
        return filterConfig;
    }
    public void destroy() {
    }
    public void init(javax.servlet.FilterConfig filterConfig) throws
    javax.servlet.ServletException {
        this.filterConfig = filterConfig;
    }
}
```

```xml
<filter>
    <filter-name>StandardEncodeFilter</filter-name>
    <display-name>StandardEncodeFilter</display-name>
    <description></description>
    <filter-class>corepatterns.filters.encodefilter.StandardEncodeFilter</filter-class>
</filter>
<filter>
    <filter-name>MultipartEncodeFilter</filter-name>
    <display-name>MultipartEncodeFilter</display-name>
    <description></description>
    <filter-class>corepatterns.filters.encodefilter.MultipartEncodeFilter</filter-class>
    <init-param>
        <param-name>UploadFolder</param-name>
        <param-value>/home/files</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>StandardEncodeFilter</filter-name>
    <url-pattern>/EncodeTestServlet</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>MultipartEncodeFilter</filter-name>
    <url-pattern>/EncodeTestServlet</url-pattern>
</filter-mapping>
```
Intercepting Filter (6)

• Consequences
  – Centralizes control with loosely coupled handlers
  – improves reusability
  – declarative and flexible configuration
  – information sharing is inefficient

• Related patterns
  – Front Controller
  – Decorator [GoF], Pipes and Filters [POSA1]
Service to Worker (1)

• Problem
  – You want to perform core request handling and invoke business logic before control is passed to the view.

• Forces
  – To handle a request, business logic
  – View selection may depend from business logic
  – You may have to use a framework (e.g. Struts)
Service to Worker (2)

- **Solution**
  
  *Service to Worker* is composed of several other patterns: *Front Controller*, *Application Controller* and *View Helper*.
Service to Worker

• Consequences
  – Centralizes control and improves modularity, reusability, and maintainability
  – Improves role separation
Business Tier Patterns

- **Business Delegate**
- **Service Locator**
- **Session Façade**
- **Application Service**
- **Business Object**
- **Composite Entity**
- **Transfer Object**
- **Transfer Object Assembler**
- **Value List Handler**

Encapsulates access to a business service.

A logical extension of the business tier that lives in the presentation tier.
Business Tier Patterns

- Business Delegate
- Service Locator
- Session Façade
- Application Service
- Business Object
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler

Encapsulates service and components lookups.

Hide the implementation details (lookup, exceptions) from client.

Increase performance through caching (lookup)
Business Tier Patterns

- Business Delegate
- Service Locator
- Session Façade
- Application Service
- Business Object
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler

1 façade = 1 set of related use cases

Encapsulates business-tier components and exposes a coarse-grained service to remote clients.

Hide complexity.
Reduce network traffic.
Reduce coupling between tiers.
Centralize security and transaction control.
Business Tier Patterns

- Business Delegate
- Service Locator
- Session Façade
- Application Service
- Business Object
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler

*Provide the background infrastructure for Session Façades, which become simpler (no bus. logic).*

*Centralizes and aggregates behavior to provide a uniform service layer.*
Business Tier Patterns

- Business Delegate
- Service Locator
- Session Façade
- Application Service
- **Business Object**
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler

**BO** encapsulates and manages business data, behavior, and persistence.

**OO vs. procedural**

**Isolates business data and logic in an object model.**

**Intrinsic vs. extrinsic business logic (app. service)**
Business Tier Patterns

- Business Delegate
- Service Locator
- Session Façade
- Application Service
- Business Object
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler

Parent vs. dependent objects
EJB 1.1 vs. EJB 2.0

Implements persistent Business Objects using local entity beans and POJOs
**Business Tier Patterns**

- Business Delegate
- Service Locator
- Session Façade
- Application Service
- Business Object
- Composite Entity
- **Transfer Object**
- Transfer Object Assembler
- Value List Handler

*aka Value Object*

*You want to transfer multiple data elements.*

*Reduce network traffic*

*Carries data across a tier*
Business Tier Patterns

- Business Delegate
- Service Locator
- Session Façade
- Application Service
- Business Object
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler

```java
public class ProjectDetailsData {
    public ProjectTO projectData;
    public ProjectManagerTO projectManagerData;
    public Collection listOfTasks;
}
```

You want to obtain an application model.

Assembles a composite transfer object from multiple data sources.
Business Tier Patterns

- Business Delegate
- Service Locator
- Session Façade
- Application Service
- Business Object
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler

You have a remote client that wants to iterate over a large results list.

Handles the search, caches the results, and provides the ability to traverse and select items from the results.
Business Tier Patterns

- Business Delegate
- Service Locator
- Session Façade
- Application Service
- Business Object
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler
Session Façade (1)

• Problem
  – You want to expose business components and services to remote clients.

• Forces
  – Coupling between tiers should be reduced
  – Network traffic should be reduced
  – Complex interactions between business components should be hidden from the client
Session Façade (2)
Session Façade (3)
Session Façade (4)
Session Façade (5)

• Consequences
  – Introduces a layer that provides services to remote clients
  – Exposes a uniform coarse-grained interface
  – Reduces coupling between the tiers
  – Increases flexibility and maintainability
  – Reduces complexity
  – Improves performance (network)
  – Centralizes security management
  – Centralizes transaction control
  – Exposes fewer remote interfaces to clients
Business Delegate (1)

- Problem
  - You want to hide clients from the complexity of remote communication with business service components.
Business Delegate (2)
Business Delegate (3)
Business Delegate (4)

• Forces: You want to...
  – access the business-tier components from your presentation-tier.
  – minimize coupling between clients and the business services.
  – avoid unnecessary invocation of remote services.
  – translate network exceptions into application or user exceptions.
  – hide the details of service creation, reconfig., and invocation retries from the clients.
Business Delegate (5)
Business Delegate (6)

- Consequences
  - Reduces coupling, improves maintainability
  - Translates business service exceptions
  - Improves availability
  - Exposes a simpler, uniform interface to the business tier
  - Improves performance
  - Introduces an additional layer
  - Hides remoteness
Mini CyberCoach
Mini CyberCoach

Do we really need a distributed architecture?
• fat clients?
• complexity of the service layer?
• performance?
• EJB 2.0 remote vs. local interfaces (colocated containers)
Integration Tier Patterns

- Data Access Object
- Service Activator
- Domain Store
- Web Service Broker

**Possible Clients:**
- Business Object
- Session Façade
- Application Service
- Value List Handler
- Transfer Object Assembler

**Abstracts and encapsulates access to persistent store.**
**Isolates the persistent storage implementation**

**Encapsulate the persistent store technology (RDBMS, LDAP, ...)**
Integration Tier Patterns

- Data Access Object
- **Service Activator**
- Domain Store
- Web Service Broker

Receives messages and invokes processing asynchronously.

You want to invoke services asynchronously (pub/sub, point-to-point).

How to send a response:
- client polls database
- send an email
- send a JMS message

Implemented as a JMS listener that receives client requests.
Integration Tier Patterns

- Data Access Object
- Service Activator
- Domain Store
- Web Service Broker

Provides a transparent persistent mechanism for business objects.

You want to separate persistence from the object model (!= EJB)

Write your own persistence framework OR use a COST product
Integration Tier Patterns

- Data Access Object
- Service Activator
- Domain Store
- Web Service Broker

Exposes one or more services using XML and web protocols.
Integration Tier Patterns

- **Data Access Object**
- Service Activator
- Domain Store
- Web Service Broker
Data Access Object (1)

• Problem
  – You want to encapsulate data access and manipulation in a separate layer

• Forces
  – isolate persistence from the application
  – provide uniform API to different data sources (RDBMS, LDAP, XML, etc.)
  – encapsulate proprietary features to facilitate maintainability and portability
Data Access Object (2)
Data Access Object (3)
Data Access Object (4)

• Consequences
  – Centralizes control with loosely coupled handlers
  – Enables transparency
  – Provides object-oriented view and encapsulates database schemas
  – Enables easier migration
  – Reduces code complexity in clients
  – Organizes all data access code into a separate layer
  – Adds extra layer
  – Needs class hierarchy design (Factory Method Strategies)
  – Introduces complexity to enable object-oriented design (RowSet Wrapper List Strategy)
Best practices in J2EE projects

- Project management
- Documentation (requirements, architecture, design, code)
- Coding standards
- Testing (unit, load, stress, acceptance)
- Release management (dev/stag/prod)
- Teams (horizontal vs. vertical)
- Risks (identify early and attack)
- Methodology: RUP, XP, etc.
Infrastructure and tools

- J2EE Server
- Database
- IDE, Modeling tool
- Code repository:  
  - CVS
- Build management:  
  - Ant
- Unit Testing:  
  - JUnit, Cactus
- Logging:  
  - JDK 1.4, Log4j
- Web frameworks:  
  - Struts, Sun ONE Application Framework, etc.
- O/R Mapping:  
  - JDO, Hibernate, Castor, TopLink, Cocobase, etc.
- Caching:  
  - OpenSymphony OSCache
Conclusion

• The design of enterprise applications
  – is a challenging task
  – requires experience
  – is not an exact science

• Design patterns make it possible to
  – share and reuse experience
  – avoid common mistakes

• The Core J2EE design patterns
  – make it easier to design J2EE systems
  – are applicable to other distributed platforms