Software Design and Implementation

J2EE Design Patterns
Enterprise Software

- Enterprise is an organization of individuals or entities, working together to achieve some goals.
- They have common needs for collaboration: information sharing, resource planning, customer management, etc.
- Enterprise software has to provide support for all of those needs.
- B2B and B2C
J2EE Patterns

- Same idea
- Application of GoF patterns
- Patterns are not isolated island
- J2EE specifics
  - Existing infrastructure
  - Enterprise environment
  - More complex – need (in depth) understanding of J2EE
- In the course we’ll consider only few patterns
J2EE Patterns

- Pattern name
- Problem
- Forces (Motivation)
- Solution
  - Structure
  - Strategies
- Consequences
- Sample code
- Related Patterns
Terminology

- EJB – Enterprise Java Bean
- POJO – Plain Old Java Object
- BMP – Bean Managed Persistence
- CMP – Container Managed Persistence
- DAO – Data Access Object
J2EE and UML

- (Heavy) use of stereotypes
  - EJB
  - SessionEJB
  - EntityEJB
  - View
  - JSP
  - ServletException
  - Session
  - Singleton
  - Custom Tag
  - POJO
Core J2EE Patterns

http://java.sun.com/blueprints/corej2eepatterns/Patterns/
# J2EE Core Patterns

<table>
<thead>
<tr>
<th>Presentation Tiers</th>
<th>Business Tier</th>
<th>Integration Tier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercepting Filter</td>
<td>Business Delegate</td>
<td>Data Access Object</td>
</tr>
<tr>
<td>Front Controller</td>
<td>Service Locator</td>
<td>Service Activator</td>
</tr>
<tr>
<td>Context Object</td>
<td>Session Façade</td>
<td>Domain Store</td>
</tr>
<tr>
<td>Application Controller</td>
<td>Application Service</td>
<td>Web Service Broker</td>
</tr>
<tr>
<td>View Helper</td>
<td>Business Object</td>
<td></td>
</tr>
<tr>
<td>Composite View</td>
<td>Composite Entity</td>
<td></td>
</tr>
<tr>
<td>Service to Worker</td>
<td>Transfer Object</td>
<td></td>
</tr>
<tr>
<td>Dispatcher View</td>
<td>Transfer Object Assembler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Value List Handler</td>
<td></td>
</tr>
</tbody>
</table>

| Integration Tier            |                                 |                             |
| Data Access Object          |                                 |                             |
| Service Activator           |                                 |                             |
| Domain Store                |                                 |                             |
| Web Service Broker          |                                 |                             |
J2EE Core Patterns
Problem BLOG

• Design J2EE application for simple blogging
• Web interface
• Enterprise Beans
• Persistence
Software Design and Implementation

J2EE Design Patterns
Problem

- You want to intercept and manipulate a request and a response before and after the request is processed
Forces

- Centralized, common processing across request
- Pre and postprocessing components loosely coupled
- Pre and postprocessing components independence
Intercepting Filter (144)

- Add pluggable filter to pre and postprocess request and responses
Filter strategies

- Standard filter strategy
- Custom strategy with use of GoF patterns
- Base Filter Strategy
- Template Filter Strategy
- …
Consequences

- Centralizes control with loosely coupled handler
- Improves reusability
- Declarative and flexible configuration
- Information sharing is inefficient
Related patterns

- Front controller
- Decorator (GoF)
- Template Method (GoF)
- Interceptor
- Pipes and Filters
Problem

• You want to hide clients from complexity of remote communication with business service components
• Changes if business service layer API should not result in modifying clients
• Remove infrastructure code from client such as JNDI, network failures
Forces

- Access to business-tier components from presentation-tier
- Minimize coupling between clients and business-tier
- Avoid unnecessary invocation to remote services
- Translate network exceptions into application exceptions
- Hide details of service creation, reconfiguration and invocation
Business Delegate (302)

- Encapsulates access to a business service
- Acts as a client-side business abstraction
Collaborations
Strategies

- Delegate Proxy Strategy
  - Validation
  - Caching

- Delegate Adapter Strategy
  - B2B integration
  - Server not necessarily J2EE
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
Related Patterns

- Service Locator
- Session Facade
- Proxy (GoF)
- Adapter (GoF)
- Broker
Problem

- You want to expose business components and services to remote client
- Reduce coupling between business components, services and remote clients
- Give access to local, not remotely accessible services
Forces

- Avoid giving clients direct access to business-tier components
- Remote access layer to business tier components
- Centralize and aggregate all business logic that needs to be exposed to remote clients
- Hide complex interactions and interdependencies between business components and services
- Reduce coupling
Session Façade (341)

- Implement session façade to encapsulate business-tier components to prevent tight coupling
Session Façade (341)

- Implement session façade to encapsulate business-tier components to prevent tight coupling
- Clients use session façade instead directly using business-tier components
- Expose only required interfaces used by clients
- No business logic – use Application Service design pattern
- One use case one session façade?
Collaborations
Session Façade Strategies

- Stateless session façade
  - Business method needs only one method call

- Stateful session façade
  - Multiple method calls – conversational business process
Consequences

- Introduces a layer that provides services to remote clients
- Exposes a uniform coarse-grained interface
- Reduces coupling between the tiers
- Promotes layering, increases flexibility and maintainability
Consequences (2)

- Reduces complexity
- Improves performance, reduces fine-grained remote methods
- Centralizes security management
- Centralizes transaction control
- Exposes fewer remote interfaces to clients
Related patterns

- Business Delegate
- Business Object
- Application Service
- Data Access Object
- Service Locator
- Façade (GOF)
- Broker
Problem

- Conceptual domain model with business logic and relationship
- Reusability is reduced and business logic code gets duplicated
- Bloated procedure implementations that become lengthy and complex
- Poor maintainability due to duplication and because business logic is spread over different modules
Terminology

• Business model
  – business use-case model to describe the business actors and the business processes
  – business object model to describe business entities used by the business use cases

• Domain model
  – abstract model that captures the most important types of objects in the context of the system.
Forces

- Conceptual model containing structured, interrelated composite objects
- Conceptual model with sophisticated business logic, validation and business rules
- Separate the business state and related behavior from the rest of the application, improving cohesion and reusability
- Centralize business logic and state in an application
- Increase reusability of business logic and avoid duplication of code
Business Object (374)

- Separate business data and logic using an object mode
Collaboration
Identifying Candidate Business Object

- Entities that represent business data identified by nouns
- Additional entities for association between two entities
Business Object (BO) and Persistence

- BO are almost always persistent
- Map object state to data store
- Persistence logic should be not coded into BO
- Implementing persistence
  - EJB Persistence
  - Custom DAOs
  - JDO
  - Domain Store
Strategies

• POJO Business Object
  – DAO, JDO, Domain Store

• Composite Entity Business Object Strategy
  – Recommend use local entity beans
POJO or EJB

- Security
- Transaction
- Caching
- Pooling
- Concurrency
- Data synchronization
- Persistence
Consequences

- Promotes object-oriented approach to the business model implementation
- Centralizes business behavior and state, and promotes reuse
- Avoids duplication of and improves maintainability of code
- Separates persistence logic from business logic
- Promotes service-oriented architecture
- POJO implementations can induce, and are susceptible to, stale data
- Adds extra layer of indirection
- Can result in bloated objects
Related Patterns

- Composite Entity
- Application Service
- Transfer Object
- Data Access Object (DAO)
- Domain Store
Problem

- You want to transfer multiple data elements over a tier
- Return data to the (remote) client from server-side components as Session Facades and Business Objects
- Need to access component that are encapsulated in different tier
Forces

- Clients want to access data from other tiers to manipulate and update data
- Reduce remote requests
- Avoid network performance degradation
Transfer Object

- Caries multiple data elements across tier
Collaborations
Strategies

- Updatable Transfer Object
- Multiple Transfer Object
- Entity Inherits Transfer Object
Transfer object

- Data Object
- Value Object
Consequences

- Reduce network traffic
- Simplifies remote object and remote interfaces
- Transfers more data in fewer remote calls
- Reduces code duplication (Inheritance)
- Introduces stale transfer objects (propagation of changes)
- Increases complexity due to synchronization and version control
Related Patterns

• Session Facade
• Transfer Object Assembler
• Value List Handler
• Composite Entity

• Memento (GoF)
Problem

- Encapsulate data access and manipulation on separate layer
Forces

• Implement data access mechanisms to access and manipulate data in a persistent storage
• Decouple the persistent storage implementation from the rest of your application
• provide a uniform data access API for a persistent mechanism to various types of data sources (RDBMS, LDAP, OODB, XML, files)
• Organize data access logic and encapsulate proprietary features to facilitate maintainability and portability.
Data Access Object (462)
Collaborations(2)
Strategies

- Custom DAO Strategy
- DAO Factory strategies
  - Factory Method
- Transfer Object Collection Strategy
- Cached RowSet Strategy
- Read Only RowSet Strategy
- RowSet Wrapper List Strategy
Consequences

- Centralizes control with loosely coupled handlers
- Enables location transparency
- Provides object oriented view
- Enables easier migration
- Reduces code complexity of clients
- Organizes all data access to separate layer
- Add extra layer
- Introduces complexity
Related Patterns

- Transfer Object
- Transfer Object Assembler
- Value List Handler
- Factory Method (GoF)
- Abstract Factory (GoF)
- Broker
Problem

- You want to separate persistence from your object model
Forces

- Avoid putting persistence details in your Business Objects
- Don’t want to use Entity Beans
- Application might be running in web container
- Your object model uses inheritance and complex relations
Domain Store (516)
Structure
Strategies

- Custom persistence strategy
  - On demand state retrieval
- JDO Strategy
Consequence

- Creating custom persistence is complex task
- Multi-layer object-tree(graph) loading and storing requires optimization techniques
- Improves understanding of persistence frameworks
- A full-blown persistence framework might be overkill for a small object model
- Improves testability of your persistent object model
- Separates business object model from persistence logic
• Martin Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley, 2002
  – Unit of Work
  – Query Object
  – Data Mapper
  – Table Data Gateway
  – Dependent Mapping
  – Domain Model
  – Data Transfer Object
  – Identity Map
  – Lazy Load
Not to sin

• There are other solutions than Java
• Distributed intraNet Architecture (DNA)
  – 3 layers: presentation, business, data access
• Service Oriented Architecture
• Object Relational Mapping
• Interoperability
Object-relational mapping

- Class = Table
- Reference to other object(s)
- What about inheritance?
  - Concrete table inheritance
  - Single table inheritance
  - Class table inheritance
Persistence

- Use of tools
  - Java: hibernate
  - .NET: nhibernate
  - C++
  - Smalltalk
Can be also used in non-J2EE solutions
Also in non-java solutions
Important is idea of the solution
Terminology domain model and business model
DAO can be used in C#, C++, VB, ...
Use of Transfer Object to transfer data between tiers (transfer simple object)