J2EE Architecture & Design

Angela Fogarolli

Angela.Fogarolli@apss.tn.it

Purpose of creating an architecture

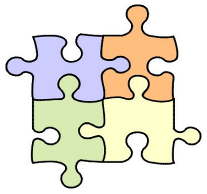
To support the service-level requirement of a system.

Without service-level requirements, your systems cannot meet customer demand for availability, reliability and scalability.

Architecture definition

Architecture is a set of structuring principles that enables a system to be comprised of a set of simpler systems each with its own local context that is independent of but not inconsistent with the context of the larger system as a whole.

[Sun Microsystems, Inc.]



In pratice…

Architecture describes the structure of the system to be build and how that structure supports the business end service-level requirements

…Example…if the system has a service level requirement that states no user response time will be greater than three seconds then the software infrastructure (persistence, communication, security…) you create must ensure that the system can meet this requirement.

Architecture and Design

Architecture defines what is going to be built and design outlines how you will build it.

Architecture is controlled by one or a few individuals who focus on how to archive the big picture. An architect creates an architecture, the desing team can use it to make the system archive its overall goals. The level of details of the architecture depends on the designers expertise.

The designer is concerned with what happends when a user presses a button and the architect is concerned with what happens when ten thousand users press a button.

Service-level Requirements (QoS)

- Performance
- Scalability
- Reliability
- Availability
- Extensibility
- Maintainability
- Manegeability
- Security

Service-level Requirements (QoS)

- Performance

  The architecture must allow the designers and developers to complete the system without considering the performance mesurament.

- Scalability

  The architecture must allow vertical(additional processors, memory…) and horizontal scaling(additional machines).

Service-level Requirements (QoS)

- Reliability

  Reliability ensures the integrity and consistency of the application and all its transactions.

- Availability

  Availability ensures that a service/resource is always accessible . (Use of redundancy).

Service-level Requirements (QoS)

- Extensibility

  Extensibility is the ability to add additional functionality without impacting existing system functionality.

- Maintainability

  Maintainability ensures to correct flaws in existing functionality without impacting other components of the system. (low coupling, modularity and documentation).

Service-level Requirements (QoS)

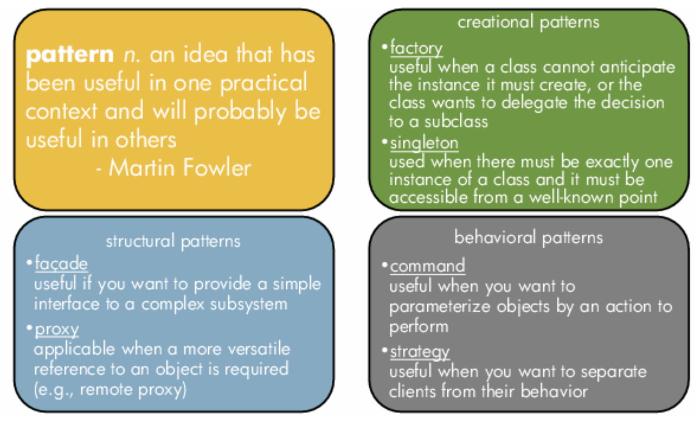
- **Manageability**

  Manageability deals with system monitoring of the QoS requirements and the ability to change the system configuration to improve the QoS without changing the system.

- **Security**

  Security is related to confidentiality, integrity and DoS attacks. An architecture that is separated into functional components makes it easier to secure the system.

Architectural Patterns



pattern *n.* an idea that has been useful in one practical context and will probably be useful in others
- Martin Fowler

creational patterns

- factory
  useful when a class cannot anticipate the instance it must create, or the class wants to delegate the decision to a subclass
- singleton
  used when there must be exactly one instance of a class and it must be accessible from a well-known point

structural patterns

- façade
  useful if you want to provide a simple interface to a complex subsystem
- proxy
  applicable when a more versatile reference to an object is required (e.g., remote proxy)

behavioral patterns

- command
  useful when you want to parameterize objects by an action to perform
- strategy
  useful when you want to separate clients from their behavior

What is a J2EE pattern?

- A J2EE design pattern is essentially any pattern that utilizes J2EE technology to solve a recurring problem.
- Patterns are typically classified by logical tier:
  - presentation tier
  - business tier
  - integration (EIS) tier

Approach to Using Patterns

" A successful J2EE implementation makes use of required patterns and best practices "

There are many decisions that must be made at different levels what architecture will I use and how will I communicate among components?

should I use EJBs? what kind and when?

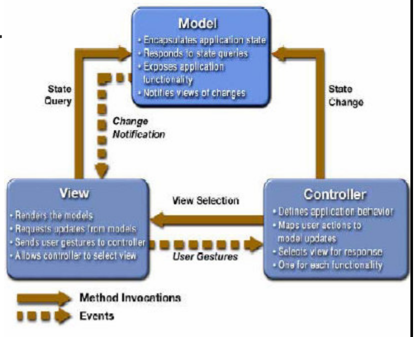
how can I develop J2EE applications efficiently?

how do I make sure my application runs well when deployed?

Patterns are simply a tool to address these issues

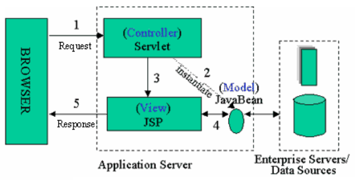
Architecture decisions Use MVC architecture

- Clear separation of design concerns
- Easier maintenance
  - decreased code duplication
  - better isolation and less interdependence
- Easy to add new client types or adjust to different client requirements
- A common approach in J2EE
  - **Model** = EJBs
  - **View** = JSPs
  - **Controller** = servlet


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Controller (type of) Example

```
ublic class dispatcher extends HttpServlet {

      public void doGet(HttpServletRequest
equest,HttpServletResponse response)
       throws ServletException, IOException
    {
        doPost(request, response);
    }


  public void doPost(HttpServletRequest request, HttpServletResponse
esponse) {
    HttpSession session = request.getSession();
    String selectedScreen = request.getServletPath();
    request.setAttribute("selectedScreen", selectedScreen);

      Integer userInt=(Integer)getServletContext().getAttribute
("userID");
      int userId = 0;
      if(userInt != null){
       userId= userInt.intValue();
      }
      System.out.println("userId");
      System.out.println(userId);
      if (userId == 0){
            try {
            request.getRequestDispatcher("/index.jsp").forward
(request, response);
      } catch (Exception e) {
        // Not possible
      }
            }
```



Dispatcher

Controller Example

```
if (selectedScreen.equals("/index")) {

            try {
            if (request.getParameter("login") != null){

            request.getRequestDispatcher
("/LoginController.java").forward(request, response);
            }else
            {
                  getServletContext().removeAttribute
("userID");
                  int usr=0;
                  Integer retInt= new Integer(usr);
                  getServletContext().setAttribute
("userID",retInt);
                  request.getRequestDispatcher
("/index.jsp").forward(request, response);
            }
      } catch (Exception e) {
        System.out.println(e);
      }
    }
     if (selectedScreen.equals("/stocks")) {

      try {
            request.getRequestDispatcher("/stocks.jsp").forward
(request, response);
      } catch (Exception e) {

      }
    }
       if (selectedScreen.equals("/main")) {

      try {
            request.getRequestDispatcher("/main.jsp").forward
(request, response);
      } catch (Exception e) {

      }
```

Dispatcher

calling View

Design Strategies

- Local vs. Remote calls
- Persistence strategies
- 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)

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

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

Persistence Strategies: Issues

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

Persistence Strategies Choices

- CMP 2.0
  – Entity beans
- BMP
- Session beans with JDBC or other strategies like JDO, Hibernate…

Advantages of CMP 2.0

- 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™ Query Language (EJB QL)
- Truly portable code

Advantages of CMP 2.0

- 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
- Complicated operations beyond scope of the EJB 2.0 specification
  – Bulk updates
  – Multi object selects
  – Aggregates (like sorting)

Recommendations

- Use CMP 2 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 now!
- If you have to build BMP entity bean, subclass CMP 2 bean
  – Easy migration to CMP later on

Transaction Strategies: Issues

- Which transaction style to use?
  - Declarative (Container-managed)
  - Programmatic (Bean-managed)
  - Client-controlled

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

Passing data between tiers

- Avoid fine grained data access calls network latencies often incurred
- Use value object pattern when
  - passing data between EJBs
  - passing data between EJB and Controller in MVC
- Use XML when
  - passing content to the “view” component of the architecture
  - communicating with heterogeneous systems

Value Object Pattern (DTO)

- Entity beans typically expose their attributes through accessor methods

  - every attribute retrieval is potentially a remote call
- The value object pattern is used to encapsulate the business data a single method call is required to retrieve all the data (reduces traffic)
- Caching can introduce stale objects.
- Domain DTO vs. Custom DTO

Example Domain DTO Implementation

```
public class StockTrader implements java.io.Serializable {

    private String id;
    private String description;
    private float value;
   private int min;

    public StockTrader(String id, String description, float value, int min) {
        this.id = id;
        this.description = description;
        this.value = value;
        this.min=min;
    }

public StockTrader() {}
    public String getId() { return id; }
    public float getValue() { return value; }
    public String getDescription() { return description; }
    public int getMin() { return min; }

    public String toString() {
        return "{ id=" + id + "; description="
            + description + "; value=" + value + "; min=" + min + "}";
    }
}
```

Example Custom DTO Implementation

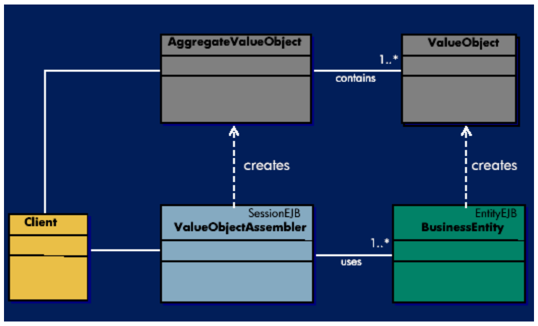
```
public class UserDetails  implements java.io.Serializable {
                        private String username;
                        private String firstname;
                        private String lastname;
                        private String email;
                        private Integer organization;
                        private Integer userId;
                        private Integer externalId;
                        private Integer groupRoleId;
                        private String groupRole;
                        private String foreignKey;
                        private HashMap groupList;
                        private java.sql.Date dateOfBirth;

    public UserDetails (String firstname, String lastname, String email, Integer organization,java.sql.Date DateOfBirth) {
                                                this.firstname = firstname;
                                                this.lastname = lastname;
                                                this.email = email;
                                                this.organization = organization;
                                                 this.dateOfBirth=DateOfBirth;
}

    /**
     * Class constructor with no arguments, used by the web tier.
     */
    public UserDetails () {}
```

Example Custom DTO Implementation

```
    public HashMap getGroupList()
     {
        return groupList;
     }

    public void setGroupList(HashMap list)
     {
        this.groupList = list;
     }

    public String getForeignKey() { return this.foreignKey; }
    public String getUsername() { return this.username; }
    public String getFirstname() { return this.firstname; }
    public String getLastname() { return this.lastname; }


    public void setDateOfBirth(java.sql.Date fk) { this.dateOfBirth = fk; }
    public void setForeignKey(String fk) { this.foreignKey = fk; }
    public void setUsername(String username) { this.username = username; }
    public void setFirstname(String firstname) { this.firstname = firstname; }
    public void setLastname(String lastname) { this.lastname = lastname; }
    public void setGroupRoleId(Integer groupRoleId) { this.groupRoleId = groupRoleId; }
    public void setGroupRole(String groupRole) { this.groupRole = groupRole; }
}
```

Assembling multiple Value Objects

- A J2EE application may need data from multiple entity beans on a single user interface screen
- The value object assembler pattern can be used to aggregate value objects



Storing Session State

Presentation tier or business tier?

- performance is generally the same
- requirements will drive the selection

Presentation tier

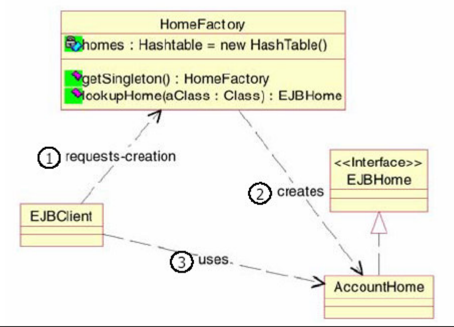
- use for web only clients
- determine when to expire a session

Business tier

- use for non-web clients
- use when transaction support is required

The Home Factory Pattern

- Insulates clients from the naming service caches lookup for better performance
- Can be used for EJB home and JNDI lookups
- Uses the singleton pattern to ensure only one instance of the factory class is created


HomeFactory
homes : Hashtable = new HashTable()
getSingleton() : HomeFactory
lookupHome(aClass : Class) : EJBHome
1 requests-creation
2 creates
3 uses
<<Interface>> EJBHome
EJBClient
AccountHome

Home Factory Example

```
import javax.ejb.*;
import java.rmi.*;
import javax.rmi.*;
import java.util.*;
import javax.naming.*;
public class EJBHomeFactory
{

     private Map ejbHomes;
     private static EJBHomeFactory aFactorySingleton;

     Context ctx;
     private EJBHomeFactory() throws NamingException
     {
          ctx = new InitialContext();
          this.ejbHomes = Collections.synchronizedMap(new HashMap());
     }
     public static EJBHomeFactory getFactory() throws
HomeFactoryException
     {

        try
        {        if ( EJBHomeFactory.aFactorySingleton == null )
                 {
                     EJBHomeFactory.aFactorySingleton = new EJBHomeFactory();
                 }

             } catch (NamingException e)
             {
                 throw new HomeFactoryException(e);
             }

             return EJBHomeFactory.aFactorySingleton;
          }
```

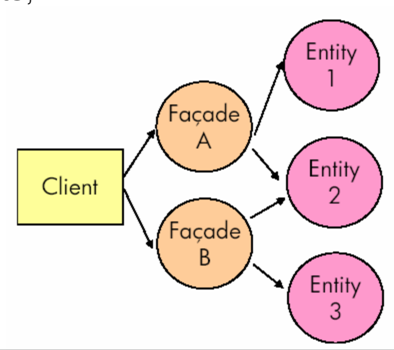
Singleton

Home Factory Example

```
        public EJBHome lookUpHome(Class homeClass)
        throws HomeFactoryException
        {

             EJBHome anEJBHome;
             anEJBHome = (EJBHome) this.ejbHomes.get(homeClass);

           try
           {
               if(anEJBHome == null)
               {
                   anEJBHome = (EJBHome) PortableRemoteObject.narrow   <-- LookUp EJB Home
                                (ctx.lookup (homeClass.getName()),
   homeClass);
                   this.ejbHomes.put(homeClass, anEJBHome);   <-- Caching home object
               }
           }
           catch (ClassCastException e)
           {
               throw new HomeFactoryException(e);
           }
           catch (NamingException e)
           {
               throw new HomeFactoryException(e);
           }

           return anEJBHome;
```

The session facade Pattern

- Uses a session bean to encapsulate the complexity of interactions between the business objects participating in a workflow.
- Manages the business objects, and provides a uniform coarse-grained service access layer to clients



Facade Example

```
public class StockManagerBean implements SessionBean
{
    public List getPortfolio(int idUser)       {
    List items = new ArrayList();
    try {
            LoginManagerHome loginManagerHome =EJBGetter.getLoginManagerHome();
            LoginManager logManObj = (LoginManager) loginManagerHome.create();
            UserDetails usr=logManObj.getUser(idUser);
            TransactionHome transactionHome =EJBGetter.getTransactionHome();
            Transaction transactionObj = (Transaction) transactionHome.create();
            List portfolioList =transactionObj.getPortfolio(usr.getCcdefault());
            ListIterator iter= portfolioList.listIterator();
            TraderHome traderHome =EJBGetter.getTraderHome();
            Trader traderObj = (Trader) traderHome.create();
            while (iter.hasNext()){
                        PortfolioStocks por= (PortfolioStocks)iter.next();
                        String a = por.getStockID();
                        StocksTrader stk=traderObj.getStock(por.getStockID());
                        float value = stk.getValue();
                        String id= stk.getId();
                        String stock= stk.getDescription();
                        float totalActualValue= por.getQuantity() * value;
                        float difference= totalActualValue - por.getTotalCost();
                        items.add(new PortfolioDetails(id,stock,por.getQuantity(),
                        por.getTotalCost(),por.getAveragePrice(),value,totalActualValue,
                        difference));
        }
                return items;
                } catch (NamingException re)
                {
                        re.printStackTrace();
                }catch (Exception exception)
                {
                        exception.printStackTrace();
                } finally{
                        return items;
                }
     } }
```

session bean call

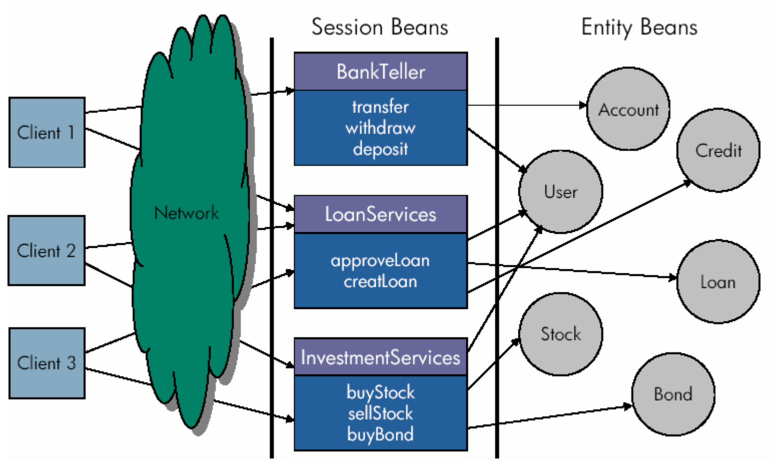
entity bean call

entity bean call

Business Logic

custom DTO creation

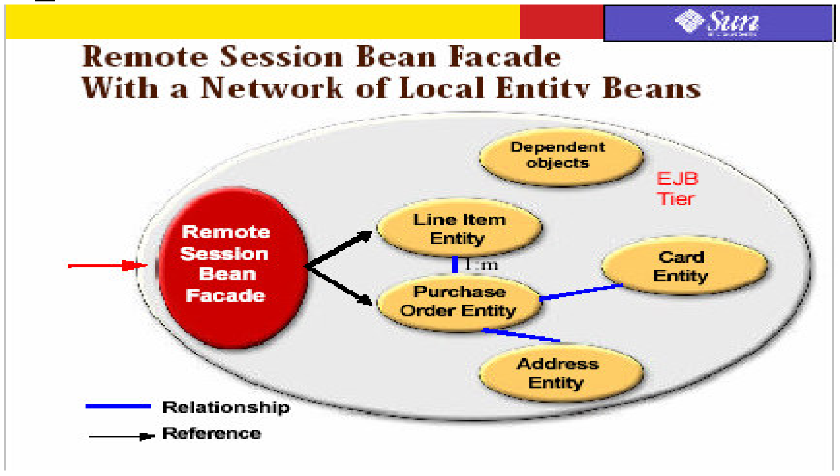
Example: Session Facade



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

Example


Remote Session Bean Facade
With a Network of Local Entitv Beans
Dependent objects
EJB Tier
Remote Session Bean Facade
Line Item Entity
1:m
Purchase Order Entity
Card Entity
Address Entity
Relationship
Reference