Patterns for Building Enterprise Solutions in .NET

Michael Stal
Siemens AG, Corporate Technology

Agenda

- Fundamentals
  - Patterns and Pattern Systems
  - Important .NET/C# features
  - Pattern-Oriented Software Engineering
  - Pattern Styles
- Patterns and .NET
  - Idioms
  - Design Patterns
  - Architectural Patterns
  - Best Practice Patterns (some of them presented with permission by David Trowbridge, Microsoft)
- Summary
What the hell is a Pattern?

- A Pattern
  - presents a solution for a recurring design problem that arises in a specific context;
  - documents proven design experience;
  - specifies a spatial configuration of elements and the behavior that happens in this configuration;
  - provides a common vocabulary and concept understanding;
  - addresses additional quality properties of the problem's solution.

Pattern Catalogs and Encyclopedias

A pattern catalog is a collection of stand-alone, self-contained pattern descriptions with no or just few connections between the patterns being described.

The most prominent pattern catalog is the Gang of Four book.

A pattern encyclopedia is an index of patterns.

The Pattern Almanac is a pattern encyclopedia that indexes all documented patterns.
Pattern Systems

A **Pattern System** is a collection of self-contained pattern descriptions where each pattern refers to (all) other patterns that can help with its realization.

To support a quick access to a particular pattern, all patterns in the system are categorized according to certain criteria, such as problem area and granularity.

Most prominent pattern systems are the POSA books.

<table>
<thead>
<tr>
<th>Architectural Pattern</th>
<th>Design Pattern</th>
<th>Idiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker</td>
<td>Command Processor</td>
<td>...</td>
</tr>
<tr>
<td>Microkernel</td>
<td>View Handler</td>
<td>...</td>
</tr>
<tr>
<td>Distributor</td>
<td>Reflection</td>
<td>...</td>
</tr>
<tr>
<td>Interaction</td>
<td>MVC</td>
<td>...</td>
</tr>
<tr>
<td>Adaptation</td>
<td>PAC</td>
<td>...</td>
</tr>
</tbody>
</table>

A **Pattern Language** is a network of tightly integrated patterns that together describe how to develop a particular type of, or part of an, application. No pattern exists in isolation.

One or more patterns define the ‘entry point’ of a pattern language and address the most important challenge to be resolved when developing the application (part) that is subject of the language. Each entry point pattern specifies which other patterns should be used to resolve sub-problems of the original problem, as well as the order in which to apply these other patterns. The referenced patterns thus complete the ‘larger’ patterns, which in turn specify useful combinations of the ‘smaller’ patterns in the presence of a particular problem.

The **Server-Component Patterns** and the **Smalltalk Best Practice Patterns** are prominent pattern languages.
Pattern-relevant
.NET/ C# Features

Patterns and Platforms

- Most patterns are independent of specific platforms, languages, paradigms, ...
- Reason: Patterns are only blueprints.
- However, the platform used has an impact on pattern implementations.
- Object-Orientatation as example:
  - On one hand, patterns are not restricted to OO environments.
  - On the other hand, OO features are really helpful for implementing patterns:
    - encapsulation, polymorphism, inheritance, interfaces
.NET/ C# Features

- Runtime System: Garbage Collection is really useful. Mind its non-determinism!
- Language Interoperability: E.g. Strategy: you may provide different strategies in different languages.
- Uniform Type System: Helps to build containers and ease implementations.
- Multithreading Support: Important for patterns such as Leaders/ Followers.

.NET/ C# Features (cont’d)

- Reflection is important for dynamic reconfiguration. Example: Component Configurator, (Dynamic) Proxy.
- Delegates and Events: Many patterns use Observer as sub-pattern.
- v2 Generics: E.g., parametrization with strategies.
- v2 Partial Types: Separate concerns on the class level.
Pattern-Oriented Software Engineering

Forward Engineering

- Patterns today are mostly applied to build software applications
Patterns - Upside Down

- There are many cases when you need to know architectural details of a given application:
  - you are going to change, adapt, extend, or integrate the application
  - the application is a framework with which your own applications have to integrate efficiently
  - you are planning to build an application or framework in the same domain
  - you want to re-use parts of the application
- But what if you don't have the application's design documents available?
- Architects usually do forward engineering, but sometimes need reverse engineering as well.
- Patterns are an ideal means in both directions.

The Birth of Patterns

- Patterns are found by reverse engineering best practices from existing applications
Back and Forth

- Thus, we get a feedback loop:

![Diagram showing the feedback loop between Reverse Engineering and Forward Engineering]

Platforms and Patterns

- If quality is an issue don't rely on transparency:
  - Use general purpose patterns for application development.
  - But give a tribute to the underlying VP by using best practice patterns specifically tailored to your VP. Otherwise you will have a lousy time as developer.
  - Mind the transparency gap: Transparency is nice to have to keep development easy but considered harmful when quality issues come into play.
  - That's the reason why for example game developers directly access the hardware.
  - Testing and Debugging is another example where transparency may blow up your head!
  - Consequence: An application developer must know the architecture of the underlying VP. As we saw, this knowledge can be best expressed with pattern systems and languages.
Pattern Variations

- Depending on architectural granularity and context we can differentiate the following styles:
  - Idioms
  - Design Patterns
  - Architectural Patterns
  - Best Practice Patterns
- There are even more styles but we won’t cover other flavors in this talk.

Idioms

- Idioms represent patterns applicable to a specific paradigm, language, or system architecture.
- Thus, idioms are less focused on application domains.
- Often, idioms are only useful in one concrete context.
- Examples: Explicit Termination Method, Object Resurrection in .NET.
Explicit Termination Method

- **Problem:** Assure that resources are freed.
- **Forces:**
  - .NET Garbage Collection is non-deterministic with respect to finalization.
  - Resources denote limited entities that should be only acquired for a minimum time.
- **Solution idea:** Make resource release explicit.

ETM Idiom

```csharp
class ResourceHolder : System.IDisposable {
    ResourceHandle h; // a limited resource
    // ... further parts
    public void Dispose() {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    protected void Dispose(bool isDisposing) {
        // free resource
        // depending on isDisposing
    }
    ~ResourceHolder () {
        Dispose(false);
    }
}
```
Client Code

```csharp
try {
    r = new ResourceHolder();
    // use the resource here ...
} finally {
    r.Dispose();
}
```

```
// Optimization:
using (r = new ResourceHolder()) {
    // using resource
}
```

Object Resurrection

- **Problem:** Resurrection of large objects.
- **Forces:**
  - Large structures are expensive to create and to keep in memory.
  - Reallocation and deletion of these structures is non-deterministic.
- **Solution:** Use weak references to free objects and recreate them when necessary.
Object Resurrection Idiom

```java
// Resource Allocation:
LargeObject large = new LargeObject(/* params */);
...
// Introduce weak reference:
WeakReference weak = new WeakReference(large);
...
// When object is not used anymore deallocate it:
large = null;

// Later on object is re-used. Try if it still exists:
large = wr.Target;
if (null == large) large = new LargeObject(/* params */);
// ... otherwise object is resurrected
```

Design Patterns

- Design Patterns are applicable to solve recurring problems in possibly specific application domains.
- They typically influence sub-systems, but not the application architecture as a whole.
- Examples: Master-Slave, Composite
Example: Master Slave

- **Problem**: Supporting fault-tolerance and parallel computation.
- **Idea**: *Divide et Impera* - partition tasks into subtasks and let components compute subtasks in parallel.

### Master

<table>
<thead>
<tr>
<th>mySlaves</th>
</tr>
</thead>
<tbody>
<tr>
<td>splitWork</td>
</tr>
<tr>
<td>callSlaves</td>
</tr>
<tr>
<td>combineResults</td>
</tr>
<tr>
<td>service</td>
</tr>
</tbody>
</table>

### Slave

- Delegate computation of subtasks to slaves

### Slaves

- Slaves calculate sub arrays. They are supposed to be executed within threads:

```java
class Slave {
    private double m_result;
    private double[] m_dList;
    private int m_start;
    private int m_end;
    public Slave(double[] dList, int start, int end) {
        m_start = start; m_end = end; m_dList = dList;
    }
    public double Result { get { return m_result; }}
    public void DoIt() {
        m_result = 0.0;
        for (int i = m_start; i <= m_end; i++)
            m_result += m_dList[i];
    }
}
```
Master

- The master uses slaves to calculate sub-arrays:

```java
class Master {
    public double CalculateSum(double[] dList, int start, int end) {
        if (start > end) throw new ArgumentException();
        if (start == end) return dList[start];
        int mid = (end - start) / 2;
        Slave s1 = new Slave(dList, start, mid);
        Slave s2 = new Slave(dList, mid+1, end);
        Thread t1 = new Thread(new ThreadStart(s1.DoIt));
        Thread t2 = new Thread(new ThreadStart(s2.DoIt));
        t1.Start(); // start first slave
        t2.Start(); // start second slave
        t1.Join();  // wait for first slave
        t2.Join();  // wait for second slave
        return s1.Result + s2.Result; // combine results
    }
}
```

Putting it together

- The manager class illustrates the configuration of the participants at runtime:

```java
class Manager {
    static void Main(string[] args) {
        double[] d = {1,2,3,4,5,6,7,8,9,10};
        Console.WriteLine(new Master().CalculateSum(d, 0, 9));
    }
}
```
**Composite**

- **Intent**
  - Enable to assemble complex objects out of primitive objects, recursive composition

- **Applicability**
  - Need to assemble objects out of primitive objects
  - Represent part-whole hierarchies

- **Structure**

  ![Composite Diagram](image)

**Composite (cont’d)**

- **Consequences**
  - It is easy to add new primitive objects that can be assembled into composites
  - Plurality is hidden from interfaces
  - Black-box reuse

- **Implementation**
  - Navigating and finding children in a composite
  - Back pointers to parent?

- **Known Uses**
  - GUI Frameworks
  - Portfolios
  - Syntax trees
Composite in C#

- Composite used to calculate resistor configurations.
- In this example you should additionally use the Visitor pattern (left out for brevity).
- Abstract base class:

```csharp
abstract class ResistorComponent {
    public abstract void AddComponent(ResistorComponent child);
    public abstract void RemoveComponent(ResistorComponent child);
    public abstract double Eval();
}
```

Composite in C# (cont’d)

- Normal resistor (leaf component):

```csharp
class Resistor : ResistorComponent {
    private double m_Value;
    public Resistor(double value) {
        if (value <= 0) throw new ArgumentException();
        m_Value = value;
    }
    public override void AddComponent(ResistorComponent child) {}
    public override void RemoveComponent(ResistorComponent child) {}
    public override double Eval() { return m_Value;}
}
```
Composite in C# (cont’d)

- **One of the Composite Components:**

```csharp
class SerialResistorComposite : ResistorComponent {
    private ArrayList children = new ArrayList(13);
    public override void AddComponent(ResistorComponent child)
    { children.Add(child); }
    public override void RemoveComponent(ResistorComponent child)
    { children.Remove(child); }
    public override double Eval() {
        double val = 0;
        foreach (ResistorComponent c in children)
            val += c.Eval();
        return val;
    }
}
```

Architectural Patterns

- Architectural Patterns are applicable to solve recurring problems in possibly specific application domains.
- They typically influence the whole application architecture.
- Example: Broker, Layers, Microkernel (we will see some of these later).
- Often architectural patterns are implemented in the infrastructure itself. Hence, to know them means to understand these infrastructures.
Best Practice Patterns

- **Best Practice Patterns address two aspects:**
  - They represent “idioms” applicable to specific platforms.
  - They help to solve recurring problems in the application domain.
- **Basically, they show how to efficiently combine a software & system architecture.**
- **Often, they are grouped to whole pattern systems/languages.**
- **Microsoft’s PAG (Patterns and Practices Group) has provided a pattern system for building enterprise systems.**
- **This patterns can be refined using the Core J2EE Patterns (this is not an error: many J2EE patterns also apply to .NET!).**

---

No Pattern is an Island

![Diagram showing interconnected patterns]
The Pattern Frame Provides Structure

Current Area Of Focus at Microsoft

Root Constraints

- OLTP
  - No embedded systems, data warehouse
- Object-Oriented Application
  - No procedural or AOP
- Layered Application
  - No monoliths
- Tiered Distribution
  - No stand-alone desktop apps

Clusters

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Presentation</td>
<td>How do you create dynamic Web applications?</td>
</tr>
<tr>
<td>Deployment</td>
<td>How do you divide an application into layers and then deploy them onto a multi-tiered hardware infrastructure?</td>
</tr>
<tr>
<td>Distributed Systems</td>
<td>How do you communicate with objects that reside in different processes or different computers?</td>
</tr>
<tr>
<td>Performance and Reliability</td>
<td>How do you create a systems infrastructure that can meet critical operational requirements?</td>
</tr>
</tbody>
</table>
How do you modularize the user interface functionality of a Web application so that you can easily modify the individual parts?

- User Interface changes more frequently
- Same data displayed different ways
- Different skill sets required for UI design and App dev

Separate the modeling of the domain, the presentation, and the actions based on user input into separate classes:

- Model - manages the behavior and data of the application domain
- View - display of information
- Controller - interprets the mouse and keyboard inputs
Implementing MVC In ASP.NET

**Controller**

```csharp
public class Solution : System.Web.UI.Page
{
    private void Page_Load(...) {...}
    void SubmitBtn_Click(Object sender, EventArgs e) {
        DataSet ds = DatabaseGateway.GetTracks(...)
        MyDataGrid.DataSource = ds;
        MyDataGrid.DataBind
    }
}
```

**Model**

```csharp
public class DatabaseGateway
{
    public static DataSet GetRecordings() {...}
    public static DataSet GetTracks(...) {...}
}
```

**View**

```html
 <%@ Page language="c#" Codebehind="Solution.aspx.cs" AutoEventWireup="false" inherits="Solution" %>
 <html>
 <body>
 <form id="Solution" method="post" runat="server">
    <asp:dropdownlist id="recordingSelect"... /> 
    <asp:button id="submit" ... /> 
    <asp:datagrid id="MyDataGrid" ... />
 </form>
 </body>
</html>
```

---

**Page Controller**

- How do you best structure the controller for moderately complex Web applications so that you can achieve reuse and flexibility while avoiding code duplication?
  - MVC focuses primarily on the separation between the model and the view
  - Many dynamic Web pages involve similar steps: verifying user authentication, extracting query string parameters etc.
  - Testing user interface code is time-consuming

- Use the Page Controller pattern to accept input from the page request, invoke the requested actions on the model, and determine the correct view to use for the resulting page.
Implementing Page Controller in ASP.NET

```csharp
public class BasePage : Page
{
    virtual protected void PageLoadEvent(object sender, System.EventArgs e) {}
    protected void Page_Load(...)
    {
        string name = Context.User.Identity.Name;
        eMail.Text = DBGateway.RetrieveAddress(name);
        siteName.Text = "Micro-site";
        PageLoadEvent(sender, e);
    }
}
```

Benefits
- Simplicity
- Leverages Framework features
- Increased reuse

Liabilities
- One controller per page
- Deep inheritance trees

Front Controller

- How do you best structure the controller for very complex Web applications so that you can achieve reuse and flexibility while avoiding code duplication?
  - Page controller can lead to overburdened base classes
  - Inheritance hierarchy is static
  - Might need to coordinate processes across pages
  - Might deal with dynamic navigation paths, e.g. a 'Wizard' that includes optional pages

- Front Controller solves the decentralization problem present in Page Controller by channeling all requests through a single controller
Implementing Front Controller In ASP.NET Using HTTP Handler

**Benefits**

- Flexibility
- Simplified Views
- Open for extension, but closed to modification (open-closed principle)
- Supports URL mapping
- Thread-safety

**Liabilities**

- Performance Implications (object creation)
- Additional complexity
Services Patterns

- **Key Points**
  - Service vs. Instance interfaces
  - Far Links
  - Web Services are about interoperability

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Interface</td>
<td>How do you make pieces of your application's functionality available to other applications, while ensuring that the interface mechanics are decoupled from the application logic?</td>
</tr>
<tr>
<td>Implementing Service Interface using .NET</td>
<td>How do you implement Service Interface in .Net?</td>
</tr>
<tr>
<td>Service Gateway</td>
<td>How do you decouple the details of fulfilling the contract responsibilities defined by the service from the rest of your application?</td>
</tr>
<tr>
<td>Implementing Service Gateway with .NET</td>
<td>How do you implement Service Gateway in .Net</td>
</tr>
</tbody>
</table>

Service Interface

**Problem**

- How do you expose your application’s functionality over a network so it can be consumed by disparate applications?

**Forces**

- Separation of concerns – business logic and service contract issues
- Consuming applications may require different channels for optimization

**Solution**

- Design your application as a collection of software services, each with a service interface through which consumers of the application may interact with the service.
Service Interface

- **Discussion Points**
  - Service vs. instance
  - Specialized kind of Remote Façade
  - May want to aggregate in to Service Layer

- **Benefits**
  - Application flexibility
  - Optimized performance
  - Interoperable

- **Liabilities**
  - Adds complexity

Implementing Service Interface using .NET

```csharp
using System.Web.Services;

namespace ServiceInterface
{
    public class RecordingCatalog : ServiceCatalog, IServiceInterface
    {
        private RecordingGateway gateway;

        public RecordingCatalog()
        {
            gateway = new RecordingGateway();
            InitializeComponent();
        }

        #region Component Designer generated code
        // ...
        #endregion

        public void RecordingCatalogLogging() { ... }
    }
}
```
Service Gateway

- **Context**
  - You are designing an application that consumes services provided by another application. The use of this service is governed by a contract indicating communication protocols, msg formats, etc.

- **Problem**
  - How do you decouple the contract implementation responsibility from the rest of your application?

- **Forces**
  - Communication and message details often change at a different rate than business logic
  - Your application may use a different data format than an external service, therefore the data may need transformation
  - If the contract changes, you want to localize the impact of the change within your application

- **Solution**
  - Encapsulate contract implementation code into a Service Gateway component

Implementing Service Gateway with .NET

- **Key Points**
  - RecordingCatalog is the Gateway
  - Recording and Track are Data Transfer Objects
Implementing Service Gateway in .NET

```csharp
public class RecordingCatalog:

    /// <remarks/>
    public RecordingCatalog() {
        this.Uri = "http://localhost/ServiceInterface/RecordingCatalog.svc";
    }

    /// <remarks/>
    public Recording Getsong(int id) {
        object[] results = this.Invoke("Get", new object[] {
            id});
        return (Recording)results[0];
    }

    }
}
```

Distributed Systems Cluster

- **Key Points**
  - Distributed Computing Challenges
  - Layered Applications
  - Coarse Grained Interfaces
  - Client versus Server Activation

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker</td>
<td>How can you structure a distributed system so that application developers don't have to concern themselves with the details of remote communication?</td>
</tr>
<tr>
<td>Implementing Broker with .Net Remoting (Server Activated)</td>
<td>How do you Implement Broker in .Net?</td>
</tr>
<tr>
<td>Singleton</td>
<td>How do you make an instance of an object globally available and guarantee that only one instance of the class is created?</td>
</tr>
<tr>
<td>Implementing Singleton in .Net</td>
<td>How do you implement Singleton in .Net</td>
</tr>
</tbody>
</table>
Broker

**Problem**
- How can you structure a distributed system so that application developers don't have to concern themselves with the details of remote communication?

**Forces**
- Distributed communication complexity
- Deployment flexibility
- Specialized skills

**Solution**
- Use the Broker pattern to hide the implementation details of remote service invocation by encapsulating them into a layer other than the business component itself

**Discussion Points**
- Location Transparency
- Server Lookup
- Broker as Intermediary
- Security Concerns

**Benefits**
- Separation of Concerns
- Complexity Management
- Flexibility

**Liabilities**
- Performance
Implementing Broker With .Net (Server Activated)

public class HttpServer
{
    static void Main(string[] args)
    {
        HttpChannel channel = new HttpChannel(8100);
        ChannelServices.RegisterChannel(channel);
        RemotingConfiguration.RegisterWellKnownServiceType(typeof(RecordingsManager), "RecordingsManager.soap", WellKnownObjectMode.Singleton);
        Console.ReadLine();
    }
}

public class RecordingsManager : MarshalByRefObject, IRecordingsManager
{
    public DataSet GetRecordings();
}

public interface IRecordingsManager
{
    DataSet GetRecordings();
}

public class HttpClient
{
    [STAThread]
    static void Main(string[] args)
    {
        HttpChannel channel = new HttpChannel();
        ChannelServices.RegisterChannel(channel);
        IRecordingsManager mgr = (IRecordingsManager)Activator.GetObject(typeof(IRecordingsManager), "http://localhost:8100/RecordingsManager.soap");
        Console.WriteLine("Client.main(): Reference acquired");
        DataSet ds = mgr.GetRecordings();
        Console.WriteLine("Recordings Count: {0}", ds.Tables["recording"].Rows.Count);
    }
}

Singleton

Problem
- How do you make an instance of an object globally available and guarantee that only one instance of the class is created?

Forces
- Global visibility
- Control of instantiation process

Solution
- Singleton provides a global, single instance by
  - Making the class responsible for creating a single instance of itself
  - Allowing other objects to access this instance through a class method that returns a reference to the instance; A class method is globally accessible
  - Declaring the class constructor as private, so that no other object can create a new instance
Singleton

- Discussion Points
  - Global visibility
  - Instantiation control

- Benefits
  - Single instance
  - Flexibility

- Liabilities
  - Overhead
  - Development confusion
  - Object lifetime

Implementing Singleton with C#

Gamma Version
```csharp
public class Singleton
{
    private static Singleton instance;
    private Singleton() {}
    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }
}
```

Basic Version
```csharp
public sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();
    private Singleton() {}
    public static Singleton Instance
    {
        get
        {
            return instance;
        }
    }
}
```

Multi-Threaded Version
```csharp
public class Singleton
{
    private static volatile Singleton instance;
    private static object syncRoot = new Object();
    private Singleton() {}
    public static Singleton Instance
    {
        get
        {
            lock (syncRoot)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
            }
            return instance;
        }
    }
}
```
Deployment Cluster

Key Points
- Layers - Logical application structure
- Tiers - Physical distribution onto server infrastructure
- Deployment - Application and infrastructure teams map components to tiers

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layered Application</td>
<td>How do you structure an application to support such operational requirements as maintainability, reusability, scalability, robustness, and security?</td>
</tr>
<tr>
<td>Three-Layered Services Application</td>
<td>How do you layer a service-oriented application and then determine the components in each layer?</td>
</tr>
<tr>
<td>Tiered Distribution</td>
<td>How should you structure your servers and distribute functionality across them to efficiently meet the operational requirements of the solution?</td>
</tr>
<tr>
<td>Three-Tiered Distribution</td>
<td>How many tiers should you have, and what should be in each tier?</td>
</tr>
<tr>
<td>Deployment Plan</td>
<td>How do you determine which tier you should deploy each of your components to?</td>
</tr>
</tbody>
</table>

Layered Application

Problem
- How do you structure an application to support such operational requirements as maintainability, reusability, scalability, robustness, and security?

Forces
- Impact of change
- Separation of concerns
- Independent components
- Operational requirements

Solution
- Separate the components of your solution into layers. The components in each layer should be cohesive and at roughly the same level of abstraction; Each layer should be loosely coupled to the layers underneath.
Layered Application

**Discussion Points**
- Dependency management
- Strict versus relaxed
- Top down versus bottom up
- Custom
- Reuse existing scheme
- Testing considerations

**Benefits**
- Easier maintenance and enhancement
- Reuse
- Support for distributed dev
- Enables tiered distribution
- Testability

**Layers**
- Layer N
- ...
- Layer J
- Layer J-1
- ...
- Layer 1

**Liabilities**
- Layer overhead
- Bound data
- Complexity
- Custom
- Change propagation

Three Layered Services Application

**Problem**
- How do you layer a service-oriented application and then determine the components in each layer?

**Forces**
- Minimize impact of adding service to an application
- Differing operational requirements by component type
- Separation of concerns

**Solution**
- Base your layered architecture on three layers - presentation, business, and data
### Three Layered Services Application

**Discussion Points**
- Presentation layer
- Business layer
- Data layer
- Foundation services

**Benefits**
- Separation of concerns
- Minimal layers

**Liabilities**
- Complex business logic may require more business layers
- Complex UI apps may require additional UI layers

### Tiered Distribution

**Problem**
- How should you structure your servers and distribute functionality across them to efficiently meet the operational requirements of the solution?

**Forces**
- Resource Consumption
- Server Optimization
- Security Requirements
- Geographic and licensing constraints

**Solution**
- Structure your servers and client computers into a set of physical tiers; A tier is composed of one or more computers that share one or more of the following characteristics:
  - System resource consumption profile
  - Operational requirements
  - Design constraints
Tiered Distribution

- **Benefits**
  - Tiers optimized for specific task
  - Different security needs
  - Operational requirements
  - Admin and deployment overhead

- **Liabilities**
  - Performance
  - Admin overhead
  - Complexity

Deployment Plan

**Problem**
- How do you determine which tier you should deploy each of your components to?

**Forces**
- Layers != Tiers
- Different Skill Sets
- Impact of adding tiers
- Resource Consumption
- Constraints, security and operational requirements
- Performance Impact of distribution

**Solution**
- The application architects must meet with the system architects to create a deployment plan that describes which tier each of the application's components will be deployed to
Deployment Plan

**Discussion Points**
- Importance of testable requirements
- Define Tiers and Components
- Map Components to tiers
- Models
  - Simple Web App
  - Complex Web App
  - Extended Enterprise App
  - Smart Client App

**Benefits**
- Equal emphasis on both operational and functional requirements
- Increased communication

**CORE J2EE Patterns**
Rationale

- Core J2EE Patterns were collected by some Java experts for Enterprise apps
- Nonetheless, most of them are applicable to .NET applications
- Even the Microsoft authors of Enterprise Solution Pattern refer to these patterns
- Let us pick some of them to illustrate their usage

Network Traffic and Latency

**Network Traffic:**
- distributed component systems generate a lot of network traffic
- accessing fine-grained entity beans from remote clients is very expensive

**Latency:**
- latency is the delay between the time at which we execute a command and the time at which it completes
- with enterprise beans there is always a bit of latency due to the time it takes to communicate requests via the network
- a client that uses many beans will suffer from a high latency

**best practice:**
- clients should do most of their work with coarse-grained session beans, that manages the workflow (session facade)
- use remote component interfaces on the session facade and local component interfaces on the enterprise beans that it manages
Session Facade

- encapsulates the complexity of interactions between the business objects
- provides a uniform coarse-grained service access layer to clients
Design coarse-grained Remote Interface Methods

- Each client call to a remote object is a remote call and involves checks such as access control, transaction, and activation/passivation (COM+).
- Each call to a remote object is many times slower than a remote call and magnitude times slower than a local method call.
- Reducing the number of calls improves the performance.
- Design coarse-grained methods:
  - increase the amount of data passed back and forth in a method call
  - combine multiple methods into fewer methods.
  - do not expose all remote object attributes via getter/setter methods

Design coarse-grained Remote Interface Methods (cont’d)

Example for an interface that is not a remote preferred interface because the client needs to make multiple calls to fetch the Person’s data:

```java
public interface Person {
    public String getFirstName();
    public void setFirstName(String firstName);
    public String getLastName();
    public void setLastName(String lastName);
    public int getPersonId();
    public void setPersonId(int personId);
}
```
Design coarse-grained Remote Interface Methods (cont’d)

Improve interface by using fewer coarse-grained methods, but those methods pass more data:

```java
public interface Person {
    public PersonData getPersonData();
    public void setPersonData(PersonData personData);
}

[Serializable]
public class PersonData {
    private String firstName;
    private String lastName;
    private int personId;

    PersonData(String firstName, String lastName, int personId) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.personId = personId;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getPersonId() { return personId; }
}
```

PersonData is called Transfer Object.

Transfer Object / Value Object
Data Access Object

- most enterprise applications use relational database management
- however data can reside in other types of repositories like:
  - mainframe systems
  - Lightweight Directory Access Protocol (LDAP) repositories
  - flat files
  - object-oriented databases
  - business to business systems
- Remote objects or web services often access the database directly

Data Access Object (cont’d)

- Use data access objects to encapsulate access to the data source and to achieve portability:
Data Access Object (cont’d)

Remote Client iterating over a large result list

- many enterprise applications have clients that perform searches
- you can fulfill search in various ways:
  - „finder methods“ provided by remote objects
  - using a data access object
- „finder methods“ have a huge performance overhead when used to perform large searches (e.g. browsing entire tables in the database)
  - they return a collection of either remote or local references
  - remote network calls are very expensive
**Remote Client iterating over a large result list (cont’d)**

- Typically, a client uses the results of a query for read-only purposes, such as displaying the result list.
- Often, the client views only the first few matching records, and then may discard the remaining records and attempt a new query.

**ValueListHandler**

Use a Value List Handler to search, cache the results, and allow the client to traverse and select items from the results.
ValueListHandler (cont’d)

Reduce Coupling between Clients and Business Services

The client should not interact directly with the business service interface due to the following reasons:

• when API changes client code has to be changed
• tight coupling decreases system flexibility
• client has to handle remote Exceptions
• the client is not transparent to naming and lookup services
**BusinessDelegate**

Use a Business Delegate to reduce coupling between clients and business services. The Business Delegate hides the underlying implementation details of the business service, such as lookup and access details of the EJB architecture.

Typically, business delegates are developed by the EJB developers and provided to the client developers.

**ServiceLocator / LookupService**

Use a Service Locator object to abstract all registry/Active Directory/UDDI usage and to hide the complexities of initial context creation.

Either the factory object can be returned to the client or the ServiceLocator can cache the factory object and gain the additional responsibility of proxying all client calls to the factory object.
Proposed Architecture

ServiceActivator

- the .NET Remoting and/or Web Service clients do not offer sufficient asynchronous invocation support
  - When a client needs to access a remote service, it has to wait until the method returns,
  - or to use a delegate
ServiceActivator (cont’d)

Use a Service Activator to receive asynchronous client requests and messages. On receiving a message, the Service Activator locates and invokes the necessary business methods on the business service components to fulfill the request asynchronously.

Transfer / Value Object Assembler

- Application clients need to obtain business data from the business tier
- An application model represents business data encapsulated by business components
- A special application of the Session Facade called Transfer Object Assembler can be used
Transfer / Value Object Assembler (cont’d)
Summary

- Patterns are useful to synthesize and to analyze systems.
- Different kinds of patterns exist: Idioms, Design Patterns, Architecture Patterns, Best Practice Patterns, ...
- Even for platform agnostic patterns infrastructure properties are important - at least for implementation aspects.
- Use patterns whenever possible AND where useful.
- This talk could only illustrate some examples. Read the patterns literature to get a more complete picture!

Suggested Reading

- Visit the Microsoft Patterns Repository: www.msdn.microsoft.com/architecture/patterns
- Gamma, Helm, Johnson, Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.
- Alur, Crupi, Malik: Core J2EE Patterns, Prentice Hall PTR, June 2003
Noch Fragen?

Development for Professionals!