Part I
Architectural Capabilities
## Learning Objectives

- Examine Tactics for architecting for non-functional requirements

## Availability Tactics

- A system becomes “unavailable” when it fails to deliver behavior consistent with its specification
- This failure is observable to users / clients
- Failure is caused by one or more faults
- Architectural tactics are required that prevent faults becoming failures.
Fault Detection

• Exceptions
  – a fault raises an exception that propagates to a handler (typically placed as close as possible to the fault)
  – The handler detects, logs and handles the fault

• “Ping!”
  – A component “pings” another component and expects a synchronous “echo” back
  – Failure to receive the echo indicates the presence of a fault
  – This tactic also useful for monitoring performance of communication paths

Fault Detection (2)

• Heartbeat
  – A component periodically emits a “heartbeat” message
  – Failure to receive a heartbeat by a listener, invokes a fault handling component
Fault Recovery

• Checkpointing and Rollback
  – Periodic creation of checkpoints that capture a consistent (and correct) state
  – In the presence of faults, the system “rolls back” to a given checkpoint

• Shadowing
  – A redundant component executes in parallel with the “real” component, and takes over if it fails
  – This is typically implemented in a different way
    - why?

Fault Recovery (2)

• Voting
  – Shadowed components “vote” on a specific outcome, and run on individual processors
  – Take the same inputs and process in parallel
  – This could be majority based
  – Can detect component or processor faults
  – Again, each voter is implemented differently
Fault Recovery (3)

• Active Redundancy
  – Input goes to a number of parallel components
  – Output is usually taken from the first to complete
  – Presence of a fault “switches out” one component and the others take up the slack

Fault Recovery (4)

• Passive Redundancy
  – One component is the primary, but it will have a number of redundant components
  – The primary informs the other components of state changes and thus keeps them synchronized
  – In the event of a fault, input should be switched to the backup
  – Also known as dual or triple modular redundancy (TMR)
Fault Recovery (5)

• Provide a spare!
  – A new system or subsystem to take over
  – Will need to be booted and set with the last known good state

How to Avoid Faults

• 1. Temporarily take the faulty component out of service, do recovery, reboot, reset
• 2. Use Transactions - in the presence of a fault, rollback to the last good state
• 3. Recreate - take a process offline, and re-instantiate it, setting it back to a previous good state
• 4.
Performance Architecture

• What architectural guidelines should be used to ensure that an event arrives within a given temporal “window”?

Resource Consumption

• What CPU, storage, memory is being used - can it be upgraded? Replaced? Enhanced?
• How are access to software resources managed?
### Blocking and Resources

- The time a process is “blocked” waiting for access to a contended resource
- The greater the contention, the longer the potential blocking time
- Choose an access algorithm that prioritizes access according to a specific performance goal e.g.
  - **Deadline monotonic**: processes with the earliest deadlines have the highest priority access to the resource
  - **Rate Monotonic**: processes with the earliest rate of iteration (period) have the highest priority
  - Many “main stream” contention algorithms are non-deterministic

### Resource Availability

- What is the availability of an individual resource?
- May be the result of downtime or failure
- Architect needs to identify these possibilities, as they can effect overall performance latency
Efficiency and Overhead

- 1. How can a given algorithm be refined or replaced to provide higher latency?
  - Improving the algorithm at key “bottlenecks” can have major performance implications
- 2. How can the overhead of computation be reduced?
  - Example tradeoffs
    - Undertaking an object-to-object call inside the same address space rather than over a distributed (and slower) protocol (performance over flexibility)
    - Using “hardwired” logic instead of reflection and polymorphism (performance over maintainability)

Event Management

- Event Rate - analyze the rate at which events are produced by the system
  - Can this rate be bounded? Decreased? To increase latency?
- Polling - analyze the sampling rate, where the architecture “polls” for data
  - Can this rate be decreased?
Bounding Resources

• Bounded Execution Times
  – Assign a worse-case execution time to operations, violating these halts processing
  – Remove (and bound) uncontrolled iteration

• Bounded Queue Sizes
  – Limits the input, and thus the resources required for processing

Performance through Concurrency

• “Many threads make light work”
• Divide the work into units of concurrency to maximize parallelism (and thus throughput)
• Make sure that the resource contention problem (threads to processors) is managed by a scheduling algorithm that meets performance needs
  – Deadline Monotonic
  – Rate Monotonic
  – FIFO, Round Robin
  – Semantic (where priority is driven by domain requirements)
Reducing Contention

• Replicate the resources (processors, data stores, devices) causing excessive blocking
• Cache software resources
  – Requires synchronization with the “real” source
• Bigger, faster resources - bigger CPU’s, more memory, greater bandwidth

Security Architecture

• Three primary requirements:
  – Resisting Attacks
  – Detecting Attacks
  – Recovering from Attacks
Resistance

- Authentication - is the user who they purports to be?
  - Solutions: passwords, digital certificates
- Access Control Patterns - ensuring that a user has rights to read/write/modify data
  - Solutions: user groups, roles, lists -> resources
- Data Confidentiality - preventing data for unauthorized access
  - Solutions: encryption on data and data transmission, key-based encryption

Resistance (2)

- Data Integrity - making sure that the data transmitted is the data received
  - Solution: MD5 checksums, hash results
- Limit Exposure - if you can’t get to it, it can’t be attacked!!
  - Solution: expose only key services to the network
- Limit Access - restrict access based on message source or destination port
  - Solution: Use of firewalls, DMZ
Detecting Attacks

• Typically the province of an “off the shelf” intrusion detection system
• Compares incoming traffic patterns with a database of known attackers
• Packets are filtered on the basis of port, address, payload size…

Recovering From Attacks

• Tie-in with fault handling - recovery to a consistent state
• Important to maintain an audit trail, to trace the attackers changes
• Remember: the audit trail can also be changed as the result of an attack!
Architecting for Modification

• Reducing the number of modules that are directly influenced by a change
  – Localization
  – Limiting Change Propagation
  – Late Binding

Localization

• Maintain a high degree of cohesion within a module
  – Localizing related functionality localizes change
  – Functionality should have strong semantic cohesiveness
• Factor out common services into separate modules
  – e.g. MVC or persistence frameworks
  – Do not replicate them on a per module basis!
• Build generically
  – Provide a broad range of functions based on input
  – Downside - it takes more time and money!
Localization (2)

- Variation Points
  - Limit the possibilities for change for a given module
  - Provide variation points where change can be introduced i.e. through subclassing or composition
  - Examples: template method pattern, strategy pattern

Limiting Change Propagation

- Information Hiding
- Maintaining Interfaces
  - Adding Interfaces
  - Provision of Adapters/Wrappers
### Limiting Change Propagation (2)

- **Reduce Coupling**
  - Limiting dependence on other modules also restricts the effect of change
  - Use of intermediaries
    - A intermediate module that arbitrates on behalf of a client module
    - Typically is an intermediate in a one-to-many scenario
    - Examples: Mediator Pattern, Strategy, Proxy…

### Late Binding

- Address modification through polymorphism (or late binding)
- Make provision for late binding to new implementation components
Architecting for Validity

- Also known as *testability*
- Architectural tactics for easier testing as software is incrementally developed
- Providing architectural support for testing can have enormous implications for savings

Record/Playback

- Provide capabilities for capturing information that crosses an interface boundary
  - This is saved in a persistent fashion
  - Used as playback as part of a test harness execution
Separating Interface & Implementation

- Separation facilitates testing of various implementations using a single interface
  - Implementation can be extended with test harness code
  - Implementation can simply be “stubbed out” or “hard wired” to facilitate unit testing of dependent modules

Specialized Test Interfaces

- Provision of “privileged routes” into an implementation
- Not exported to any client but the test harness
- Can be used to expose the state of variables for manipulation by unit test code
Part II

*Design Patterns*

Objectives

- Understand the origins and role of patterns
- Examine how patterns are commonly documented
- Analyze selected patterns drawn from the classic “Gang of Four” GoF publication
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
Definitions

• A pattern represents a design guideline or heuristic captured in as a context-independent problem-solution pair
• Derived from architectural work by Christopher Alexander
• First used in software by Erich Gamma in his doctoral thesis
• Published as: Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley 1995.

What is the Problem?

• We do “similar” rather than identical things again and again
  – Code reuse isn't necessarily the most effective solution
• Knowledge Transfer
  – How do we communicate best practices and heuristics in a standard and comprehensible fashion?
  – How can we specify common terms and concepts, so we all know we are on the same page?
Christopher Alexander

- Author of “The Timeless Way of Building”
- First applied Patterns to Building Architecture
- 1. Solves a recurring problem
- 2. Provides a proven solution
- 3. Solution is composed of generic components
  ✓ Not tied to a specific context

Access to Water

- Town and Country: Local environment

- Problem: People have a fundamental yearning for great bodies of water. But the very movement of the people toward the water can also destroy the water.
Access To Water (2)

- **Solution:** When natural bodies of water occur near human settlements, treat them with great respect. Always preserve a belt of common land, immediately beside the water. And allow dense settlements to come right down to the water only at infrequent intervals along the water's edge.

Night Life (1)

- **Town and Country:** Local centers
- **Problem:** Most of the city's activities close down at night; those which stay open won't do much for the night life of the city unless they are together.
Night Life (2)

• **Solution**: Knit together shops, amusements, and services which are open at night, along with hotels, bars, and all-night diners to form centers of night life: well-lit, safe, and lively places that increase the intensity of pedestrian activity at night by drawing all the people who are out at night to the same few spots in the town. Encourage these evening centers to distribute themselves evenly across the town.

Software Design Patterns

• Tactical solutions
  – Represent common solutions to tactical problems
• Domain Independent
  – Are not linked to a specific business domain
• Programming Language Independent
• Address both functional and non-functional issues
• Used at the architectural and detailed design level
Documenting Patterns

• Pattern Name
  – A brief, but illustrative name
• Pattern Summary
  – A brief description of what the pattern does
• Also Known As…
  – Other names that the pattern may be known by
• Illustrative Example
  – A real world example documenting the need for the pattern

Documenting Patterns (2)

• Context
  – The context in which the problem will arise
• Problem
  – A full description of the generic problem
Documenting Patterns (3)

• Solution
  - A description of how the pattern provides a solution
    ✓ Textual description
    ✓ Static Model, typically a UML class diagram
    ✓ Dynamic Model, a UML sequence or collaboration diagram

Documenting Patterns (4)

• Implementation
  – A description of how the pattern can be implemented
  – By necessity this will be context-specific
• Alternatives
  – A discussion of variants or alternative implementations
Pattern Classification

- Three types of pattern
  - Creational
  - Structural
  - Behavioral
- Address both class-centric compile time issues and object-centric runtime instantiation concerns

Creational Patterns

- Designed to abstract the complexity of object creation processes
- Creational classes are typically composed at run-time, rather than compile-time
Creational Patterns List

• Abstract Factory
• Singleton
• Builder
• Factory Method
• Prototype
Pattern Summary

- Provides an interface for creating families of related objects without specifying their concrete implementation
- Can decouple a client using these products from specific implementations e.g. commitments to a specific OS

Illustrative Examples

- XWindows (X11)
  - A cross-platform API for building event-driven GUIs
  - Has a common windowing, graphics and event API
  - Clients can connect to an XServer across a distributed network, or be compiled for a given target platform and run without code change!
Problem

• We have a client that is dependent on one or more product objects
• These products are present on a number of different platforms, they may have the same interface, but… their construction process for each platform is significantly different
• e.g. Window objects for WinXP versus OSX

Solution

• Make sure each product implements a common interface
  – This enables the client to interact with it without change
• Provide an abstract factory class to construct instances of the product interface
Solution (2)

• Define one or more concrete factories that “know” how to construct products for a specific platform
• Make sure the client only maintains references to the abstract factory and abstract products
  – Use polymorphism to associate the factory and products with concrete instances

Implementation

public interface IWindowProduct
{
    public void open ();
    public void destroy ();
}

public class OSXAquaWindowProduct implements IWindowProduct
{
    public void open () { ... }
    public void destroy () { ... }
}

public class Win32WindowProduct implements IWindowProduct
{
    public void open () { ... }
    public void destroy () { ... }
}
Implementation (2)

public interface IWindowFactory
{
    public IWindowProduct createWindow ();
}

public class OSXWindowFactory implements IWindowFactory
{
    public IWindowProduct createWindow ()
    {
        IWindowProduct aWindow = new OSXAquaWindowProduct ();
        // do environment specific setup here
        return aWindow;
    }
}

Discussion

• Why would this pattern create implementations that were difficult to maintain if there were many products?
• What would happen in the Windowing example if the interface for an OSX and Win32 window product was different? Could we use this pattern?
Singleton

Pattern Summary

• Ensures that a given class only has one specific instance at any point in time
• All clients will be given that same instance on request
Illustrative Examples

- Abstract Factories
  - typically these are singletons, why make more than one for each Factory type?
- Palm / WinCE Devices
  - PDA GUI environments may have a singleton graphics object to do all the drawing/rendering
- System Configuration Objects
  - Configuration information can be read into memory once, and made available as a singleton

Problem

- The need to simply maintain a single instance of a class, to avoid unnecessary/erroneous replication
Solution

• Provide the class with the capability to control its instance creation
• Make the class constructor private
• Provide a static getInstance() method
• Declare a private static variable of that class (within the same class!)
• Callers of the getInstance() method will receive the single static instance object

Implementation

class Singleton
{
    private static Singleton aSingleton = new Singleton();

    private Singleton()
    {
    
    }

    public static Singleton getInstance()
    {
        return aSingleton;
    }
}

Discussion

- Why is a singleton problematic within a distributed environment?
- Having identified the problem - is there a viable solution?

Structural Patterns

- Addresses composition at the class and object level
- Makes provision for using structural relationships to provide variant behavior
- Extends functionality through the use of composition rather than inheritance
Structural Patterns List

• Composite
• Adapter
• Template Method
• Proxy
• Bridge
• Façade
• Decorator

Composite
Pattern Summary

- Facilitates the construction of complex objects by recursive composition
- All objects in the composite have a common super-class or interface
- Each object is manipulated in a identical fashion

Problem

- A client needs to manage a complex object as a single object
- The complex object has an unpredictable hierarchical structure
Solution

• Facilitates the treatment of composite and primitive objects in an identical fashion
  – Whole and part are manipulated in the same fashion
• Provides functionality for manipulating composites

Solution (2)

• Define a common interface for all objects in the structure
  – The interface will provide operations common to all objects in the structure
  – The leaf object represents a “primitive” at the bottom of a given tree
  – The composite is used to encapsulate other composite or leaf objects
  – Facilitates arbitrarily large recursive structures
public interface IComponent
{
    public void sampleOperation ();
    public Composite getComposite ();
}

public class Composite implements IComponent
{
    private List children = new ArrayList ();
    public void sampleOperation () { ... }
    public Composite getComposite () { return this; }
    public void add (IComponent c) { children.add ( c ); }
    public void remove (IComponent c) { children.remove ( c ); }
    public Enumeration components () { children.enumeration(); }
}

public class Leaf implements IComponent
{
    public void sampleOperation () { ... }
    public Composite getComposite () { return null; }
}
Discussion

• Why does the getComposite() method in this method return null for the Leaf?
• Is there a way to build this pattern without placing composite operations on the Leaf Primitive?
• Why would the Component (see diagram) sometimes need to be an abstract class rather than an interface?

Adapter
**Pattern Summary**

- Addresses a mismatch between client method calls and server interfaces
- Making them interact would usually entail changing client, server or both
- The adapter is inserted between the two and isolates client from server
- Also known as a *Wrapper*

**Problem**

- The class you are expecting to reuse does not have an interface expected by the client
- You wish to construct a reusable class that will work with unforeseen classes
  - Its difficult to foresee such classes!
- Sometimes classes built separately just need to be able to work together
Solution Types

• Using Inheritance
  – Known as Class Adapters
  – Extend the Target Interface with additional functionality to perform the adaptation

• Using Composition
  – Known as Object Adapters
  – Object Adapter “wrappers” the Target and arbitrates all calls from the adaptee

Class Adapter Implementation

```java
public class Target {
  
  public void sampleOperation (TargetData data) { ... }

}

public class TargetAdapter extends Target {

  public void sampleOperation (int x, int y) {

    TargetData adaptedData = new TargetData ();
    adaptedData.setX (x);
    adaptedData.setY (y);
    // call superclass method
    this.sampleOperation (adaptedData);
  }

}
```
Object Adapter Implementation

```java
public class Target {
    public void sampleOperation (TargetData data) { ... }
}

public class TargetAdapter {
    private Target theTarget;

    public TargetAdapter (TargetAdapter t) { theTarget = t; }

    public void sampleOperation (int x, int y) {
        TargetData adaptedData = new TargetData ();
        adaptedData.setX (x);
        adaptedData.setY (y);
        // call wrappered object
        theTarget.sampleOperation (adaptedData);
    }
}
```

Discussion

• What are the advantages / disadvantages associated with each approach?
Class Adapter Issues

- Advantages
  - Easy to implement!
  - Can adapt some or all functionality
- Disadvantages
  - What if the adapter needs to inherit from two superclasses in a single inheritance language?

Object Adapter Issues

- Advantages
  - Target can be parameterized
  - Object adapter can adapt for multiple targets
- Disadvantages
  - Client knows that it is talking to an adapter
Behavioral Patterns

- Observer
- Strategy
- Iterator
- Visitor
- Interpreter
- Chain of Responsibility
- Command
- Mediator
- State
Pattern Summary

- Observer facilitates semantically related objects to remain synchronized using a publish-subscribe update mechanism
- Facilitates synchronization, but at the same time both objects remain loosely coupled

Problem

- An object has a number of disparate dependents
- They must be notified if the object state changes
- The exact number and type of these dependents is not known in advance
- Polling (for change notification) from the dependent to the object would be impractical from a performance perspective
Illustrative Examples

- Message-Oriented Middleware (MOM)
  - Provides publish-subscribe behavior
  - An Observable “publishes” the queue
  - An Observer registers interest in a specific publication topic
  - When the Observable publishes the MOM delivers this to all Observers

Solution

- Provide an observer interface that all dependents must implement
- Provide an observable (or subject) interface that the object with the key state changes that must be watched must implement
  - Observable must support methods for registering / deregistering observers
  - The Observable is responsible for propagating events back to the Observers
Implementation

public interface Observer
{
    public void inform (Data d);
}

public interface Observable
{
    public void add (Observer o);
    public void remove (Observer o);
    public void notify ();
}

Implementation (2)

public class MyObserver implements Observer
{
    public void inform (Data data) {/* do stuff with data */}
}

public class MyObservable
{
    private List observerList = new ArrayList();

    public void add (Observer o) { observerList.add (o); }
    public void remove (Observer o) {observerList.remove (o); }
    public void notify ()
    {
        ListIterator notifyIterator = observerList.listIterator();
        while (notifyIterator.hasNext())
        {
            Observer current = (Observer)
            notifyIterator.next();
            current.inform (data);
        }
    }
}
Visitor

Pattern Summary

- In a complex object structure, often made up of instances of different types common actions need to be performed
- One way to address this is to distribute that functionality through the hierarchy
- Alternatively it can be placed in a Visitor, that will “visit” each object in the structure
Problem

• Distributed functionality
  – Hard to understand
  – If distributed through an inheritance hierarchy changes at the superclass level will require subclass compilation

Solution

• Two primary abstractions
  – Element: represents an abstraction to be visited, provides a method to accept a visitor
  – Visitor: encapsulates the logic that will perform the “visit” on all elements in a given set of elements to be visited
Implementation

public interface Element
{
    public void accept (Visitor v);
}

public interface Visitor
{
    public void visitElement (Element e);
}

Implementation (2)

public class ConcreteElement implements Element
{
    public void accept (Visitor v)
    {
        v.visitElement (v);
    }

    public void sampleMethod () {  }
}

public class OtherConcreteElement implements Element
{
    public void accept (Visitor v)
    {
        v.visitElement (v);
    }

    public void yetAnotherSampleMethod () {  }
}
public class ConcreteVisitor implements Visitor {
    public void visitElement (Element e) {
        if (e instanceof ConcreteElement) {
            ConcreteElement ce = (ConcreteElement)e;
            ce.sampleMethod ();
        } else if (e instanceof OtherConcreteElement) {
            OtherConcreteElement oe = (OtherConcreteElement)e;
            oe.yetAnotherSampleMethod ();
        }
    }
}
Learning Objectives

• Review best practices for process and architectural design

• Provide a detailed overview of heuristic assessment

• Provide an introduction to the application of metrics

Layering

• Structures applications into groups of subtasks
• Each group of subtasks is defined at a particular level of abstraction
Solution

• Partition your system into large grain “layers”
• Start at the lowest level of abstraction (layer N), these will represent the foundational, or base services of your application
• Build upwards towards the “highest” level of abstraction by developing additional layers
• Each N+1 Layer will build upon the services offered by the layer below

Layering Criteria

• The basis of abstraction utilized to partition the system into layers
• Common software system partitioning
  – Presentation
  – Business Logic
  – Common Services
  – Operating System Services
  – Operating System
  – Hardware
How Many Layers?

• This depends on the chosen abstraction criteria
• Too many layers
  – Adds additional and unnecessary complexity
  – Makes the system hard to maintain
  – Inter-layer communication increases overhead
• Too few layers
  – Monolithic structure, leads to poor understandability and maintainability!

Name and Assign Layers

• Give the layer a name, and broadly assign it a set of tasks
• Each layer (layer $N$) is designed to be built upon, and in turn build upon lower layers
• Layer $N+1$ uses Layer $N$ as infrastructure to fulfill its own particular processing tasks
• Layer $N-1$ provides services that support Layer $N$
Specify Layer Services

• Specify the services of a given layer in detail
• Make sure that the layer is:
  – Cohesive: it brings together services that make sense as a consistent abstraction
    • e.g. Persistence Layer, Reporting Layer, Business Object Layer...
  – Loosely Coupled: a given layer should build upon layers at lower levels of abstraction
    • A layer should *not* know about higher levels!

Specify Layer Interfaces

• A lower level layer should represent its services to higher level layers through one or more *façades*
• A façade represents a flat interface that exposes the services that the layer offers, but “hides” the internal object structure
• Facilitates building a higher layer, where the layer below is simply a black box
• The internal structure of Layer N-1 may change, but as long as the façade(s) remain consistent layer N will require minimal or no alteration
Design Each Layer

- Behind each façade will be a package structure and object model
- These can be designed prior to layer determination and assigned by the architect to a given layer
- They can also be built for a specific layer
- There is a clear synergy between layering and the objects that implement each layer
  - Do not be afraid to refactor your emerging object model to appropriately reflect your layering scheme

Layer Intercommunication

- Decide how each layer will intercommunicate - what transmission medium and message protocols will be in place?
- Examples:
  - Method calls in the same address space (the layers are on the same machine)
  - SOAP messages to web services over HTTP
  - Publish/Subscribe XML-based communication via JMS
  - CORBA
  - Sockets passing serialized objects over UDP
Layer Coupling

- Make sure that a given layer (*Layer N*), only knows about the lower adjacent level (*Layer N - 1*).
- All calls to layers further down the stack (*Layer N -2, N - 3…*) should be arbitrated by *Layer N - 1*! Why?
- *Layer N* should not know about its users (*N+1…N+2*)
  - If this has to be done, maintain loose coupling through the use of callbacks (i.e. publish - subscribe models).

Error Strategies

- Define a common error handling strategy for your “stack” of layers
- Ideally errors should be handled (and hidden) by a given layer
- Otherwise they should be defined and propagated to higher levels by defining them as part of the layer façade.
Advantages

- Standardization - layering facilitates the definition of local and global standards
  - e.g. corporate persistence layer, CORBA
- Reuse - each layer represents a well-defined cohesive, loosely coupled abstraction, can potentially be applied in new contexts
- Localized Dependencies - coupling is only between adjacent layers, higher layers are unaware of the implementation requirements of layers that are not adjacent
  - Can be used to “hide” issues such as OS dependency

Advantages (2)

- Exchangeability
  - Higher level layers can be swapped in or out with no impact to lower level layers
  - Hiding a layer behind a façade facilitates structural / functional / dynamic change internally without impacting higher layers!
Disadvantages

• Efficiency and Replication
  – Multiple layers may impose penalties due to additional overhead (e.g. communications)
  – Processing may have to be replicated in each layer (e.g. parsing an XML document passed as data)
• Its Complicated
  – Good layering strategy comes from experience with the domain, its not easy!

Layers and Tiers

• Typically these represent interchangeable terms
• Common tier typologies
  – Two Tier - also known as “Client-Server”
  – Three Tier
  – N Tier
Two Tier Systems

- A two tier system typically partitions responsibility between an application and a database server

Advantages

- It’s simple!
- Ideal for rapidly prototyping an application with persistent storage requirements
- Suitable for script-based platforms (e.g. Python, Perl, JavaScript) where additional architectural layers are unnecessary
- Provides greater levels of performance (less “time on the wire” as business logic is located on the client)
Disadvantages

• Complex and expensive to maintain and redeploy
• Poor scalability - how do you handle an increasing user base?

Three Tier Overview
Three Tier Systems

• Partitions responsibility into three classic layers
  – User Interface
  – Business Logic
  – Persistence

Advantages

• Many of the key advantages of layering
  – Reuse
  – Maintainability: each layer can be understood and modified without needing to understand higher or lower levels
  – Effective use of technology paradigms
    • A business logic layer can represent data as objects
    • A RDMS layer can represent data as tables
    • No direct impedance mismatch by “shoehorning” one into the other!
  – Specialization - staff can localize their skills around the maintenance of specific layers
Disadvantages

• As per classic layering…
  – Performance
  – Redundancy
  – Complexity (with implications for maintainability!)

When to Use Three Tiers

• Use a three-tier architecture if any of the following conditions apply:
  – Systems with many application programs (more than 20)
  – Mix of application programs of different languages or origins
  – Two or more heterogeneous data sources (e.g., two RDBMS or a DBMS and a file system)
  – Expected life of application beyond three years
  – Many modifications and additions anticipated
  – High-volume workload (more than 10,000 transactions per day)
  – Significant inter-application communication, including inter-enterprise communication
  – Upsizing; i.e., the application may grow over time so one or more previous conditions apply

[Gartner Group]
Model-View-Controller

• Divides an interactive application into three architectural components
• *Model* - encapsulates core data and functionality
• *View* - displays information to a user, there may be multiple views
• *Controller* - provides separation and arbitration between the Model and View

What is the Problem?

• User interfaces are notoriously subject to change requests
• Many applications combine presentation and business logic
• Difficult and expensive to change “look and feel” or user experience
• Often several different versions of the application are separately maintained for different types of user and/or different customers
View-Controller Interaction

- A view is typically associated with a single controller
- This is responsible for accepting user input (from the view) and updating the model accordingly
Model-View Interaction

- Model is coupled to view and controller via a change propagation mechanism
- This is a registry of model components, where a state change will trigger an event.
- Views are observers of a given model component
- A fired change propagation event will result in a observing view having its update() method called

Advantages

- Supports multiple views of the same model - cleanly decouples business logic from presentation
- Change propagation mechanism makes sure that all views are synchronized with the model
- Easy to change look and feel
- Pluggable views and controller logic
- Its easy to build a framework for (or reuse an existing one such as Jakarta Struts)
Disadvantages

- Complexity - it's too complex for some application types
- Excessive updates - the change propagation mechanism may have to handle excessive updates, degrading performance and making debugging complex
- Tight coupling between View and Controller - difficult to reuse separately
- Change notification updates aren’t necessarily the most efficient way to get model data

Broker

- Used to decouple distributed applications by routing remote service invocations through a mediating broker
- Broker component is responsible for coordinating communications, forwarding requests, returning replies and propagating exceptions
- Also known as “Spoke and Hub”
What’s the Problem?

- Monolithic applications are significantly less flexible and scalable than applications partitioned into distributed components.
- With distribution, a component becomes coupled to communication mechanisms and server component identities.
- Services for adding, removing, exchanging and starting components are also required.
The Solution

- Introduce a broker to arbitrate communication
- Servers register themselves with the broker, and expose method interfaces
- Can be updated, added or moved from the broker registry
- Clients only have a single dependence - on the broker

Broker Participants

- Client - a component that communicates with server(s) via the broker
- Server - a component that services requests from clients via the broker
- Broker - the central point for message routing
- Client-Side Proxy - an object used for client-to-broker communication, used to hide implementation details of the protocol and transmission medium
- Server-Side Proxy - an object used for server-to-broker communication, used to hide protocol / communication details
Advantages

• Location Transparency
  – Broker isolates clients from specific knowledge of server components

• Flexibility
  – Clients and servers can be changed with minimal impact to other components

• Portability
  – Broker isolates components from other component dependencies such as communication protocols, message formats and operating systems

Disadvantages

• Testing and Debugging
  – Testing a client fully requires distributed functionality to be in place
  – Required to test broker component interoperability
  – Tedious and time-consuming (believe me!)

• Efficiency
  – Delayed binding time and communication overheads reduce performance
Steps to Implement Broker

1. Design an object model
2. Define the component interoperability model - binary, IDL, SOAP over HTTP
3. Specify the broker API for communicating with clients and servers
4. Build client and server proxies
   - Client proxy to hide details of interaction with broker
   - Server proxy to receive requests from broker and pass back results and exceptions

Steps to Implement Broker (2)

5. Design the broker component
   - Specify the “on-the-wire” protocol
   - Implement a directory service for associating local server identifiers with physical machines
   - Provide marshalling and unmarshalling capabilities for message transformation (e.g. from XML to objects)
     - 1.
Pipe and Filter

- Provides a structure for applications that do “end to end” processing on a data stream
- Data is passed through “pipes” between adjacent filters
- Flexibility is derived from combining/recombining filters into various configurations

What’s the Problem?

- Data processing task is clearly subdivided into specific processing tasks
- Each task is self-contained and decoupled from other tasks
- Order and requirements for steps are likely to change
- Data may come from different input sources
- Data may be required to go to different output sources
Solution

• Build each step as a specific *Filter*
  – A filter consumes and delivers data as a specific and individual processing step
  – Operates as a typical “black box” with data input and outputs
  – Connected to other filters via *Pipes*.
  – As long as input data is consistent with its requirements can be combined and recombined into different orders

Pipe and Filter Diagram
Advantages

- Does not require intermediate data storage facilities between steps
  - But can be added if required!
- Changing order - functionality can be swapped in and out by inserting or recombinining filters
- Reuse - filters are context independent and cohesive, they exhibit high reuse potential
- Rapid Prototyping

Disadvantages

- Hard to have a notion of shared state or global data - this has to be passed through as input
  - Particularly limiting if parts of the pipe-filter run in parallel!
- Overhead
  - Each filter will typically have to undertake some data transformation, this may be expensively replicated on a filter-by-filter basis
  - Parallel Processing of Pipe/Filter
    - Expensive context-switching overheads
    - Complex synchronization problems if different pipe-filter chains need to interact
Disadvantages (2)

• Error Handling
  – Impossible to propagate errors back to completed filters
  – The only way to support error recovery is to “undo” previous filters and restart
  – Requires complex checkpointing and recovery mechanisms

Service-Oriented Architecture

• Business application expose one or more “business services”
• These are typically described using WSDL (Web Services Description Language)
• The service typically accepts SOAP XML messages (Simple Object Access Protocol), typically over HTTP/S or JMS
• The service is called through an intermediate SOAP processing engine such as apache axis
• Services can be placed in a distributed registry for easy discovery, typically a UDDI (Universal) registry
Advantages

- XML based communication facilitates heterogeneous system interaction
  - E.g. CICS to .NET, .NET to J2EE
- HTTP based communication standard, secure (over HTTP/S) and firewall-friendly
Disadvantages

• “Find, bind, invoke” paradigm (UDDI, WSDL, SOAP) couples clients to a specific web service - change in service may require code change on the client side
• Web service standards (WS-*) such as WS-Reliability, WS-Security, Ws-Addressing… often a moving target, incomplete or overlapping
• HTTP does not provide a rich error topology for debugging in the event of communication failure (e.g. 500, 404)

Designing Web Services

• What is the purpose of the service?
  – Is it cohesive?
  – Is it loosely coupled from other supported services?
• What granularity should the service be?
  – Small Grain, a set of smaller services that have a higher reuse potential, and can be combined/recombined to collectively support variable behavior
  – Large Grain, less reuse potential, but easier to use
• Web Service Design - many of the issues associated with function, object design
Implementing Web Services

• Define a “service point” the actual code unit that will implement the service
  – EJB Session Bean, C# Object…
• Generate or write the WSDL specification associated with that service point
• Client generated typically using tools such as wsdl2java, building and sending SOAP messages consistent with WSDL specification

Enterprise Service Bus (ESB)

• Based on the “hardware bus” design metaphor
• Services are connected to the bus as abstract “end points”
• Clients place messages directly onto the bus
• These are transformed into a “canonical” format (typically XML) and routed by the bus to the connected services
• The result of bus processing can be returned to the client, or to a number of designated recipients
**ESB Layers**

- **Service-Oriented Architecture**
  -- Services represent abstract endpoints connected to a communication bus
- **Intelligent Routing**
  -- Services called by a "workflow layer" using content-based or itinerary-based techniques
- **Message-Oriented Middleware**
  -- A communication bus based on asynchronous, reliable pub-sub or point-to-point messaging
- **Transformation Services**
  -- Any-to-any transformation services (e.g. EDI to XML)
- **Protocol/Application Adapters**
  -- Adapters to connect a rich variety of clients to the bus

**ESB in a Nutshell**

![Diagram of ESB layers](image_url)
Advantages

1. “One stop shop” for internal or extended enterprise integration efforts
2. Empowers developers to be integrators
3. Ideal basis for hosting an emerging service-oriented architecture
   • ESB provides loose-coupling as opposed to the “find, bind, invoke” client-server WS paradigm
   • Services can be composed to form “large grain” workflows
4. “Layer and Leave” over existing EAI infrastructure, gradually migrating services and routing capabilities across to the bus

Disadvantages

• Complicated - many interrelated facets
• Immature - still an emerging technology, early adopters are on the “bleeding edge”
• Expensive - vendor products range from 200k upwards!