Software Engineering  
G22.2440-001  

Session 8 – Main Topic  
From Analysis and Design to Software Architectures  
(Part III)  

Dr. Jean-Claude Franchitti  

New York University  
Computer Science Department  
Courant Institute of Mathematical Sciences  

Agenda  

• Enterprise Architectural Patterns  
• Sample Middleware Reference Architectures  
• Architectural Capabilities  
• Object-Oriented Design Guidelines  
• Summary  
  • Individual Assignment #5  
  • Project (Part 3)
Summary of Previous Session

- Review
- OOAD Using UML
  - Use Cases Review
  - Use Case Design
  - GRASP Patterns
  - System Design
  - UML and OOAD Summarized
  - UML to Java Mapping
  - Sample OCL Problem
- Introduction to Design and Architectural Patterns
- Micro/Macro Architecture
- Design and Architectural Patterns
- Summary
  - Individual Assignment #4 - ongoing
  - Project (Part 2) - ongoing

Part I

Enterprise Architectural Patterns

(See Sub-Topic 1 Presentation on Design Patterns, Architectural Patterns)
Part II

Sample Middleware Reference Architecture ala J2EE

(See Sub-Topic 2 presentation on Middleware J2EE Reference Architecture)

Learning Objectives

- Examine tactical solutions for building scalable and maintainable J2EE systems
- Introduce the following J2EE patterns
  - Business Delegate
  - Service Locator
  - Data Access Object
  - Session Façade
  - Data Transfer Object
  - Fast Lane Reader
  - Value List Handler
  - Intercepting Filter
- While these are notionally “J2EE Core Patterns”, the majority of these solutions can be applied in both a J2EE, .NET or any other component technology context
Business Delegate / Service Locator

- Lookup of server-side business components can be a complex undertaking
  - e.g. JNDI lookup, UDDI for Web Services Discovery
- Business Delegate decouples the client from the necessary server lookup code
- Provides a standard interface to server-side logic, and encapsulates the programmatic interface

Service Locator

- The dynamic discovery of services between tiers is typically facilitated by a naming and lookup service (e.g. JNDI/LDAP/UDDI)
- This may introduce the following problems:
  - Lookup code is replicated at multiple points in the client tier
  - Lookup code is performance intensive
  - Lookup code may often include vendor-specific API’s, this makes it difficult to change discovery service providers without extensive code change
Solution!

- Encapsulate the lookup logic in a service locator object
- This will typically implement a single strategy (e.g. JNDI)
- It may (optionally) provide caching capabilities for service references, thus avoiding expensive lookup overheads for the same service

Diagram
Data Access Object

- Persistence code can often be divided across the business and database application tiers
- This makes it difficult to change the underlying persistence mechanism without impacting the application code
- Common scenario: JDBC. While the vendor details are abstracted away to some extent coupling is increased by…
  - a. tying the code to a relational model through connection management, statements and result sets
  - b. using “hardwired” SQL in the code. This often ties you to a specific vendor implementation of SQL-92 (e.g. the use of functions)

Solution

- Provide an object/interface that encapsulates the persistence logic
- A common interface facilitates:
  - Minimal coupling with the persistence layer
  - The capability to change persistence implementation without requiring client changes
Example Implementation

// From the J2EE PetStore implementation

/* * Copyright 2002 Sun Microsystems, Inc. All rights reserved. */

public class CatalogDAO {
    public Category getCategory(String categoryID, Locale l) throws CatalogDAOException {
    }
    public Page getCategories(int start, int count, Locale l) throws CatalogDAOException {
    }
    public Product getProduct(String productID, Locale l) throws CatalogDAOException {
    }
    public Page getProducts(String categoryID, int start, int count, Locale l) throws CatalogDAOException {
    }
    public Item getItem(String itemID, Locale l) throws CatalogDAOException;
    public Page getItems(String productID, int start, int size, Locale l) throws CatalogDAOException {
    }
    public Page searchItems(String query, int start, int size, Locale l) throws CatalogDAOException {
    }
}
Session Façade

- Many business processes require complex workflow and manipulate numerous business objects
- This can lead to tight coupling between a client and the respective interfaces of server side objects
- This has an adverse impact on maintainability
  - Decreases design understandability
  - Forces the client to understand and coordinate complex business object interaction
- This has an adverse impact on performance
  - Increased coordination between client and server == increased “time on the wire”
- This has an adverse effect on fault tolerance
  - Transactional systems have to “stretch” the transaction across client-server boundaries

Session Façade Solution

- A Session Façade provides a “Large Grain” coordinating object that manages business object interactions on the server-side
- The client makes a single call to the façade interface
  - The façade unpacks the parameters and executes the workflow as a logical extension of the client
  - From the client perspective complexity (and coupling) are reduced to a single workflow unit
  - Reduces “time on the wire” as all parameters are passed at once
  - Facilitates a single server-side transaction
- Typically a session façade is used to model a given use case scenario (that isn’t a hard and fast rule folks!)
Implementation

- A core pattern in the application of J2EE technology
- A Session Façade is typically implemented using a stateless session bean
- It will be used to execute the workflow across one or more entity beans or POJOs
- It will also represent the beginning of a server-side transaction (specified using the requires or requiresnew transaction attributes in the deployment descriptor)
Data Transfer Objects (DTO’s)

• Addresses a key issue in server-side (and particularly EJB development)
  – “If I hide all my business objects behind facades how do I access their encapsulated data in a fine-grained fashion?”

• For EJB, accessing entity beans directly from the client side is considered bad practice
  – Introduces all the problems a session façade is designed to resolve (e.g. maintainability, performance…)

A Solution!

• Use intermediate objects to encapsulate and transport data
• These should encapsulate the following:
  – A set (or subset) of the data from an entity bean
  – Getters and Setters to access the data
  – A version id to facilitate safe updates
  – The capability to transform themselves into a format suitable for distributed communication (e.g. Serialization)
DTO Diagram

Versioning

- A problem with the DTO pattern is that the server-side data and the DTO may become unsynchronized
  - e.g. “stale” data in a DTO may overwrite “fresh” data encapsulated in the entity bean
- Solution: provide a versioning number in the entity bean and the DTO
  - Only version numbers that match can be used to update the entity bean using the DTO.
Fast Lane Reader

• Designed to efficiently lookup “read-only” tabular data, without the overhead of object creation in EJB or POJO frameworks
• An OO-relational scheme typically assigns an object to a given row of data extracted from a RDBMS
  – In cases of large-scale lookups (e.g. a SELECT on 10000 Member rows), this can lead to excessive object creation overhead
  – In EJB this overhead is increased by the need to support numerous container managed services for each object

The Solution!

• Bypass the object-relational mapping, go straight to the database via a DAO pattern
• In Java/J2EE this will typically be accomplished by a DAO implemented using JDBC
• A Session bean will utilize the DAO and process the tabular data
Value List Handler

- Multi-tier web applications typically require the use of large lists of objects (e.g. for display)
- Such lists have the following characteristics:
  - Typically only a few items in the list are required
  - The list represents a large (and inefficient) use of client memory
  - There is a significant network overhead in transmitting the list from server to client tier.
Solution

- The server tier exports an *iterator*.
- This is used to by the client tier pull back additional objects on request.
- Objects “pulled back” to the client tier are cached in a local list for the purposes of reverse scrolling and random access.
- Advantages:
  - Allows the client to minimize memory usage, it’s no longer an “all or nothing” proposition
  - The client tier only needs to iterate to find the objects it requires, thus reducing network latency.

Diagram
Intercepting Filter

• Many applications have infrastructure requirements such as logging or authentication that have to be uniformly applied for all client requests
• This becomes a “cross cutting” concern, and will have to be done by all “Handlers”
• This can lead to code replication, with the obvious maintainability implications

The Solution:

• Place one or more Filter objects into the path of communication
• Each filter will implement a specific type of request (e.g. logging)
• An incoming request will pass through the filter on its way to the server-side object
• In a synchronous communication mode, control will pass back through the filter after server-side processing
Example Filter Implementation

```java
public class EncodingFilter implements Filter {
    private FilterConfig config = null;
    // default to ASCII private String targetEncoding = "ASCII";

    public void init(FilterConfig config) throws ServletException {
        this.config = config;
        this.targetEncoding = config.getInitParameter("encoding");
    }

    public void destroy() {
        config = null;
        targetEncoding = null;
    }

    public void doFilter(ServletRequest srequest, ServletResponse sresponse, FilterChain chain) throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest)srequest;
        request.setCharacterEncoding(targetEncoding);
    // move on to the next
        chain.doFilter(srequest, sresponse);
    }
}
```
Part III

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

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 attacker's 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 IV

Object-Oriented Design Guidelines

(See Sub-Topic 3 presentation on Middleware J2EE Reference Architecture)
Part V

Conclusion

Course Assignments

• Individual Assignments
  • Problems and reports based on case studies or exercises

• Project-Related Assignments
  • All assignments (other than the individual assessments) will correspond to milestones in the team project.
  • As the course progresses, students will be applying various methodologies to a project of their choice. The project and related software system should relate to a real-world scenario chosen by each team. The project will consists inter-related deliverables which are due on a (bi-) weekly basis.
  • There will be only one submission per team per deliverable and all teams must demonstrate their projects to the course instructor.
  • A sample project description and additional details will be available under handouts on the course Web site.
Course Project

- Project Logistics
  - Teams will pick their own projects, within certain constraints: for instance, all projects should involve multiple distributed subsystems (e.g., web-based electronic services projects including client, application server, and database tiers). Students will need to come up to speed on whatever programming languages and/or software technologies they choose for their projects - which will not necessarily be covered in class.
  - Students will be required to form themselves into "pairs" of exactly two (2) members each; if there is an odd number of students in the class, then one (1) team of three (3) members will be permitted. There may not be any "pairs" of only one member! The instructor and TA(s) will then assist the pairs in forming "teams", ideally each consisting of two (2) "pairs", possibly three (3) pairs if necessary due to enrollment, but students are encouraged to form their own 2-pair teams in advance. If some students drop the course, any remaining pair or team members may be arbitrarily reassigned to other pairs/teams at the discretion of the instructor (but are strongly encouraged to reform pairs/teams on their own). Students will develop and test their project code together with the other member of their programming pair.

Sample Project Methodology

Very eXtreme Programming (VXP)

- After teams formed, 1/2 week to Project Concept
- 1/2 week to Revised Project Concept
- 2 to 3 iterations
- For each iteration:
  - 1/2 week to plan
  - 1 week to iteration report and demo
Sample Project Methodology
Very eXtreme Programming (VXP)
(continued)

• Requirements: Your project focuses on two application services
• Planning: User stories and work breakdown
• Doing: Pair programming, write test cases before coding, automate testing
• Demoing: 5 minute presentation plus 15 minute demo
• Reporting: What got done, what didn’t, what tests show
• 1st iteration: Any
• 2nd iteration: Use some component model framework
• 3rd iteration: Refactoring, do it right this time

Revised Project Concept (Tips)

1. Cover page (max 1 page)
2. Basic concept (max 3 pages): Briefly describe the system your team proposes to build. Write this description in the form of either user stories or use cases (your choice). Illustrations do not count towards page limits.
3. Controversies (max 1 page)
First Iteration Plan (Tips)

- Requirements (max 2 pages):
  - Select user stories or use cases to implement in your first iteration, to produce a demo by the last week of class
  - Assign priorities and points to each unit - A point should correspond to the amount of work you expect one pair to be able to accomplish within one week
  - You may optionally include additional medium priority points to do “if you have time”
  - It is acceptable to include fewer, more or different use cases or user stories than actually appeared in your Revised Project Concept

First Iteration Plan (Tips)

- Work Breakdown (max 3 pages):
  - Refine as engineering tasks and assign to pairs
  - Describe specifically what will need to be coded in order to complete each task
  - Also describe what unit and integration tests will be implemented and performed
  - You may need additional engineering tasks that do not match one-to-one with your user stories/use cases
  - Map out a schedule for the next weeks
  - Be realistic – demo has to been shown before the end of the semester
2\textsuperscript{nd} Iteration Plan (Tips):
Requirements

- Max 3 pages
- Redesign/reengineer your system to use a component framework (e.g., COM+, EJB, CCM, .NET or Web Services)
- Select the user stories to include in the new system
  - Could be identical to those completed for your 1\textsuperscript{st} Iteration
  - Could be brand new (but explain how they fit)
- Aim to maintain project velocity from 1\textsuperscript{st} iteration
- Consider what will require new coding vs. major rework vs. minor rework vs. can be reused “as is”

2\textsuperscript{nd} Iteration Plan (Tips):
Breakdown

- Max 4 pages
- Define engineering tasks, again try to maintain project velocity
- Describe new unit and integration testing
- Describe regression testing
  - Can you reuse tests from 1\textsuperscript{st} iteration?
  - If not, how will you know you didn’t break something that previously worked?
- 2\textsuperscript{nd} iteration report and demo to be presented before the end of the semester
2\textsuperscript{nd} Iteration Report (Tips): Requirements

- Max 2 pages
- For each engineering task from your 2\textsuperscript{nd} Iteration Plan, indicate whether it succeeded, partially succeeded (and to what extent), failed (and how so?), or was not attempted
- Estimate how many user story points were actually completed (these might be fractional)
- Discuss specifically your success, or lack thereof, in porting to or reengineering for your chosen component model framework(s)

2\textsuperscript{nd} Iteration Report (Tips): Testing

- Max 3 pages
- Describe the general strategy you followed for unit testing, integration testing and regression testing
- Were you able to reuse unit and/or integration tests, with little or no change, from your 1\textsuperscript{st} Iteration as regression tests?
- What was most difficult to test?
- Did using a component model framework help or hinder your testing?
Project Presentation and Demo

• All Iterations Due
• Presentation slides (optional)

Readings

• Readings
  • Slides and Handouts posted on the course web site
  • Documentation provided with business and application modeling tools
  • Pattern Documentation (www.martinfowler.com)
  • SE Textbook: Chapters 8-12 (Part 2), 18-19 (Part 3), 30 (Part 5)
• Project Frameworks Setup (ongoing)
  • As per references provided on the course Web site
• Individual Assignment
  • See Session 8 Handout: “Assignment #5”
• Team Assignment
  • See Session 6 Handout: “Team Project” (Part 2)
Next Session
Software Engineering Tools Primer

- Build Tools (e.g., Ant)
- Continuous Build Process Frameworks (e.g., CruiseControl)
- Unit Testing Frameworks (e.g., jUnit)
- Refactoring Browsers (e.g., IntelliJ IDEA)
- Selecting Appropriate Tools
- Individual Assignment #5
- Project (Part 2) - Ongoing
- Readings