IMPORTANT NOTICE TO STUDENTS

These slides are NOT to be used as a replacement for student notes. These slides are sometimes vague and incomplete on purpose to spark class discussions.

Service Layer Design
“Facade Vs. Command”

CS 446/646 ECE452
[DATE]
Introduction

Client Tier
- thick
- thin

Presentation Tier
- controller
- controller
- controller

Business Tier
- Service layer
  - service
  - service
  - service
- Data access layer
  - DAO
  - DAO
  - DAO

Persistence Tier
Introduction

Client Tier

Presentation Tier
- HTTP Servlet
- web service

Business tier
- Service layer
  - login
  - time of day
- Data access layer
  - User DAO

Persistence Tier
Facade

Intent

- “provide a unified interface to a set of interfaces in a subsystem”

Considerations
- inter component coupling
- system evolution
- fault-tolerance

creational? structural? behavioural?
Facade

Intent

- “provide a unified interface to a set of interfaces in a subsystem”
Facade pattern: http://en.wikipedia.org/wiki/Facade_pattern
Facade Pattern

Advantages

• looser coupling
• lower network communication
  – in enterprise application each method call incurs communication latency
• provides an extension point (how?)
  – security, logging
• promotes reusability
  – unit of (business) work
• simple to understand & implement
Facade Pattern

Disadvantages

• Evolution
  – facade methods are written in stone (*why?*)

• Scalability
  – addition of new methods
  – deprecation of old methods
  – facade becomes complicated itself
    • error reporting/handling
    • does not grow organically
Facade Pattern

Disadvantages

- **Re-usability**
  - change in execution environment
  - aggregation of facade methods
Facade based Service Layer

Class Activity

- identify facade
  - login
  - change password
Login Example

:UserFacade
:ServiceLookup
:UserService
:AuthenticationService
:AuditService

login(...) → lookup(serviceIds) → loaduser() → authenticate(context) → audit(....)
Command Pattern

Intent

• “encapsulate the request as an object...”

So what

• how does the execution change?
  - can we serialize objects?
  - can we aggregate requests (commands)?

• separation of concerns
  - caller object from the execution object

• dynamic in nature
  - commands can be replaced at runtime (why?)
where is the functional logic implemented here?
A switch can be associated with any other component by simply injecting the appropriate commands for flip up and flip down.
Command Service Layer

- Invokes:
  - Handler
    - executeCommands(...)
  - Command
    - execute()
  - Commands
    - getbyld(id):Command

- Dependency:
  - LoginCommand
    - execute()
  - ChangePasswordCommand
    - execute()
Command Service Layer

Observation

- if commands represent business functionality then how come they are exposed to the client?
- in tiered applications, how do we deal with the marshalling and demarshalling of commands objects?
  - expensive to move 'heavy duty' objects
Command Service Layer

- Client
  - creates
  - «bean» Context
    - id
    - status
    - requestParams: Object[]
    - response: Object
    - executionFlags

- «invoker» Handler
  - executeCommands(...)

- «interface» Command
  - execute()

- Commands
  - getById(id): Command
  - LoginCommand
    - execute()
  - ChangePasswordCommand
    - execute()
Command Service Layer

Evolution

- defining new commands is trivial
- deprecating commands is easy
  - only need to retain the command identifier

Unit of work

- each unit of work is a command
- what was the unit of work in facade?
Scalability

- as the system grows we only add new concrete implementations
- more control over execution environment
- *can I merge two or more commands into a single execution unit – composite command?*
Command as Service Layer

Re-usability

- commands are simple & hence can be used in many different ways
  - single command
  - command chains (aggregation)
  - composite command

Testing

- easy to test
  - due to the separation of concerns