

Architectural Design Patterns

Project

- Sign up your group on Canvas
- Anyone not part of a group will be notified and asked to form groups.

Outline

- Introduction to Design Patterns
- Introduction to Some Common Design Patterns
 - Layering Pattern
 - MVC Pattern
 - Session State Patterns
 - Domain Logic Patterns
 - Messaging Channel Patterns
- Further Readings
- Lab & Project

Further Reading

"Patterns of Enterprise Application Architecture" -Martin Fowler *et al*

"Enterprise Integration Patterns" – Gregor Hohpe and Bobby Woolf

Design Patterns (reminder)

- What?
 - A general *reusable* solution to a *commonly occurring* problem within a given context in software design
- Why?
 - Common requirements
 - Proved mature solutions
 - Can be reused/repeated!

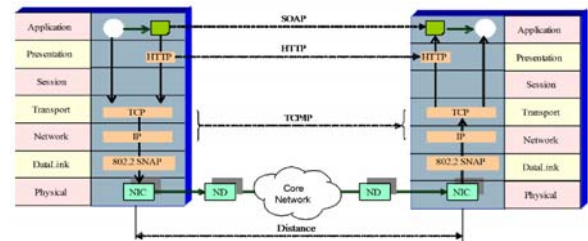
Classification of Design Patterns

- **Coding-level design patterns**, such as:
 - Creational design patterns: How to create objects? e.g., Factory, Builder, Singleton,
 - Structural design patterns: How to construct a system with objects/components? e.g., Adapter, Bridge, Decorator, Façade, Proxy, etc.
 - Behavioural patterns: How to control objects at runtime? e.g., Interpreter, Iterator, Mediator, Observer, etc.

"Design Patterns" -Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
- **Architectural-level design patterns**
 - How to construct a system?
 - How to ensure a secure session?
 - How to manage domain data?
 - How to communicate each other?

System Design Patterns

Laying Pattern (Vertical Viewpoint)



Laying Pattern

- With layering, we can:
 - Understand a single layer as a coherent whole without knowing much about the other layers.
 - Substitute layers with alternative implementations of the same basic services.
 - Minimize dependencies between layers.
 - Standardize each layer's interfaces to its upper layers
 - Use the standard interfaces for many/various higher-level services.

Laying Pattern - problems

- Cannot encapsulate all things well (e.g. field displayed on UI and in database must be added in all layers in between)
- Too many layers can harm performance
- Hard to decide what layers to have!!

Horizontal vs Vertical

- Vertical typically about business functionality (more dependent on individual application)
- Horizontal are common across many applications

Horizontal Laying Pattern



Presentation Layer

- Handle interaction between the user and the software.
 - command-line or text-based menu system
 - rich-client graphics UI or an HTML-based browser UI.
- Display information to the user and to interpret commands from the user into actions upon the domain and data source.

Domain/Business Logic Layer

- Work that the application needs to do for the domain you're working with
 - Calculations based on inputs and stored data
 - Data validation from presentation layer
 - Determining what logic to run depending on commands from presentation layer

Data Service layer

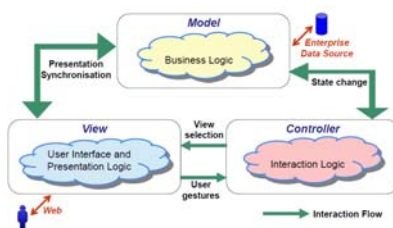
- Manage database
- Communicating with other systems that carry out tasks on behalf of the application.
 - Transaction monitors,
 - Other applications,
 - Messaging systems, and so forth.

Horizontal Layering Pattern

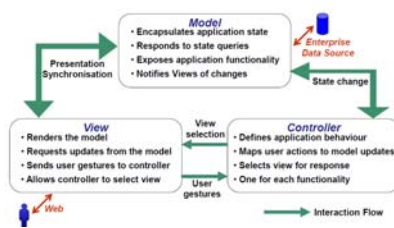


- With the multi-tiers (layers), we can:
- Separate/isolate complexities from each other
 - Maximize reuse of business components & data
 - Making it easier to manage and scale

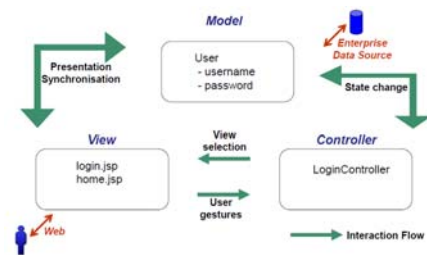
MVC Pattern



MVC Pattern



MVC Pattern Spring example



User Model

```

1 package au.edu.qut.elec5002.domain;
2
3 public class User {
4     private String username;
5     private String password;
6     public User() {
7     }
8     public User(String username, String password) {
9         this.username = username;
10        this.password = password;
11    }
12    public String getUsername() {
13        return username;
14    }
15    public void setUsername(String username) {
16        this.username = username;
17    }
18    public String getPassword() {
19        return password;
20    }
21    public void setPassword(String password) {
22        this.password = password;
23    }
24 }

```

Login view

```

1 <!-- login.jsp -->
2 <!-- login.jsp -->
3 <!-- login.jsp -->
4 <!-- login.jsp -->
5 <!-- login.jsp -->
6 <!-- login.jsp -->
7 <!-- login.jsp -->
8 <!-- login.jsp -->
9 <!-- login.jsp -->
10 <!-- login.jsp -->
11 <!-- login.jsp -->
12 <!-- login.jsp -->
13 <!-- login.jsp -->
14 <!-- login.jsp -->
15 <!-- login.jsp -->
16 <!-- login.jsp -->
17 <!-- login.jsp -->
18 <!-- login.jsp -->
19 <!-- login.jsp -->
20 <!-- login.jsp -->
21 <!-- login.jsp -->
22 <!-- login.jsp -->
23 <!-- login.jsp -->
24 <!-- login.jsp -->
25 <!-- login.jsp -->
26 <!-- login.jsp -->
27 <!-- login.jsp -->
28 <!-- login.jsp -->
29 <!-- login.jsp -->
30 <!-- login.jsp -->
31 <!-- login.jsp -->
32 <!-- login.jsp -->
33 <!-- login.jsp -->
34 <!-- login.jsp -->
35 <!-- login.jsp -->
36 <!-- login.jsp -->
37 <!-- login.jsp -->
38 <!-- login.jsp -->
39 <!-- login.jsp -->
40 <!-- login.jsp -->
41 <!-- login.jsp -->
42 <!-- login.jsp -->
43 <!-- login.jsp -->
44 <!-- login.jsp -->
45 <!-- login.jsp -->
46 <!-- login.jsp -->
47 <!-- login.jsp -->
48 <!-- login.jsp -->
49 <!-- login.jsp -->
50 <!-- login.jsp -->
51 <!-- login.jsp -->
52 <!-- login.jsp -->
53 <!-- login.jsp -->
54 <!-- login.jsp -->
55 <!-- login.jsp -->
56 <!-- login.jsp -->
57 <!-- login.jsp -->
58 <!-- login.jsp -->
59 <!-- login.jsp -->
60 <!-- login.jsp -->
61 <!-- login.jsp -->
62 <!-- login.jsp -->
63 <!-- login.jsp -->
64 <!-- login.jsp -->
65 <!-- login.jsp -->
66 <!-- login.jsp -->
67 <!-- login.jsp -->
68 <!-- login.jsp -->
69 <!-- login.jsp -->
70 <!-- login.jsp -->
71 <!-- login.jsp -->
72 <!-- login.jsp -->
73 <!-- login.jsp -->
74 <!-- login.jsp -->
75 <!-- login.jsp -->
76 <!-- login.jsp -->
77 <!-- login.jsp -->
78 <!-- login.jsp -->
79 <!-- login.jsp -->
80 <!-- login.jsp -->
81 <!-- login.jsp -->
82 <!-- login.jsp -->
83 <!-- login.jsp -->
84 <!-- login.jsp -->
85 <!-- login.jsp -->
86 <!-- login.jsp -->
87 <!-- login.jsp -->
88 <!-- login.jsp -->
89 <!-- login.jsp -->
90 <!-- login.jsp -->
91 <!-- login.jsp -->
92 <!-- login.jsp -->
93 <!-- login.jsp -->
94 <!-- login.jsp -->
95 <!-- login.jsp -->
96 <!-- login.jsp -->
97 <!-- login.jsp -->
98 <!-- login.jsp -->
99 <!-- login.jsp -->
100 <!-- login.jsp -->

```

Login Controller

```

1 package au.edu.qut.elec5002.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.ui.Model;
6 import org.springframework.ui.ModelMap;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RequestMethod;
9
10 import au.edu.qut.elec5002.service.UserService;
11
12 @Controller
13 public class LoginController {
14     @Autowired
15     private UserService userService;
16
17     @RequestMapping(value = "/login", method = RequestMethod.GET)
18     public String login() {
19         return "login";
20     }
21
22     @RequestMapping(value = "/login", method = RequestMethod.POST)
23     public ModelAndView login() {
24         ModelAndView modelAndView = new ModelAndView();
25         modelAndView.addObject("username", "");
26         modelAndView.addObject("password", "");
27         modelAndView.addObject("message", "");
28         if (!userService.login(username, password)) {
29             modelAndView.addObject("message", "Invalid username or password.");
30         }
31         return modelAndView;
32     }
33 }

```

Why separate view from model?

- Different concerns
 - UI
 - Business policies/database interactions
- Users want to see same information in different ways
 - Web browser/mobile/command line
- People programming model should not know/care how it is presented
 - Presentation changes can be made freely

Separation of view from controller

- In general applications, not always critical
 - Exception: Editable vs non-editable behaviour
- Common for web applications

Secure sessions

Session State Patterns

- Session
 - Semi-permanent interactive information interchange, also known as a conversation, between two or more devices
- Session State
 - information required to maintain the unique conversation, such as session id, your shop cart, your emails, etc.

Why Session State

- Security: allow a server to check if the request is from a valid login-ed user;
- Personalized web page: allow a server to provide personalized services, such as your shopping cart,

Session State

- Options:
 - Client Session state – Data on client e.g. URL for a web presentation/cookies/hidden field on web form/store on rich client
 - Server Session State – Data in server memory between requests
 - Database Session State – Data in tables and fields in database

Client Session State

- Improve server performance
 - Stateless servers
 - Distribute across compute servers
 - Easy failover recovery
- Communication overhead
 - Programming overhead
 - Not Secure
- When to use
 - Almost always needed for session ID
 - Small amount of session state data
 - Not serious applications

Server Session State

- Advantages
 - Existing software can help (less programming)
 - Can handle complex session objects
 - Secure
- Hard to use clusters/recover from failure
- Memory Cost
- When to use:
 - Secure systems
 - Complex session data

Database Session State

- Uses Session ID
- Can handle large session state objects
- Can handle multiple types of session state data
- Clustering/Failover recovery easier
 - Use Database Redundancy
- Secure
- Have to separate session data from norma data
 - Programming/performance overhead
 - Need to manage transactions
- When to use:
 - Large/complex session data with multiple types
 - Secure systems
 - Useful server clustering

Managing domain data

- M in MVC
- Domain logic patterns
 - Transaction script - scripts
 - Domain Model – Use objects
 - Table Model – One class per table in database

Transaction Script

- Simple, fast
- Can get complex as logic grows
 - Hard to maintain/reuse
 - Creates duplicate code

Domain Model

- Easy re-use
- Can handle complicated business
 - E.g. everchanging business rules involving validation, calculations, and derivations (chances are that you'll want an object model to handle them)
- Expensive at creation
- Use for large/complex systems

Example

- Transaction script

<http://lorenzo-dee.blogspot.com/2014/06/quantifying-domain-model-vs-transaction-script.html>

```

1 public interface MoneyTransferService {
2     BankingTransaction transfer(
3         String fromaccountId, String toaccountId, double amount);
4 }
5
6 public class MoneyTransferServiceTransactionScriptImpl {
7     implements MoneyTransferService {
8         private AccountDb accountDb;
9         private BankingTransactionRepository bankingTransactionRepository;
10    }
11    @Override
12    public BankingTransaction transfer(
13        String fromaccountId, String toaccountId, double amount) {
14        Account fromAccount = accountDb.findById(fromaccountId);
15        Account toAccount = accountDb.findById(toaccountId);
16        double newBalance = fromAccount.getBalance() - amount;
17        switch (fromAccount.getOverdraftPolicy()) {
18            case NONE:
19                if (newBalance < 0) {
20                    throw new DebitException("Insufficient funds");
21                }
22                break;
23            case ALLOWED:
24                if (newBalance < -limit) {
25                    throw new DebitException(
26                        "Overdraft limit of " + limit + " exceeded: " + newBalance);
27                }
28                break;
29        }
30        fromAccount.setBalance(newBalance);
31        toAccount.setBalance(toAccount.getBalance() + amount);
32        BankingTransaction moneyTransferTransaction =
33            new MoneyTransferTransaction(fromaccountId, toaccountId, amount);
34        bankingTransactionRepository.addTransaction(moneyTransferTransaction);
35        return moneyTransferTransaction;
36    }
37 }

```

Example domain model

```

1 public class MoneyTransferServiceDomainModelImpl {
2     implements MoneyTransferService {
3         private AccountRepository accountRepository;
4         private BankingTransactionRepository bankingTransactionRepository;
5     }
6     @Override
7     public BankingTransaction transfer(
8         String fromaccountId, String toaccountId, double amount) {
9         Account fromAccount = accountRepository.findById(fromaccountId);
10        Account toAccount = accountRepository.findById(toaccountId);
11        fromAccount.debit(amount);
12        toAccount.credit(amount);
13        BankingTransaction moneyTransferTransaction =
14            new MoneyTransferTransaction(fromaccountId, toaccountId, amount);
15        bankingTransactionRepository.addTransaction(moneyTransferTransaction);
16        return moneyTransferTransaction;
17    }
18 }

```

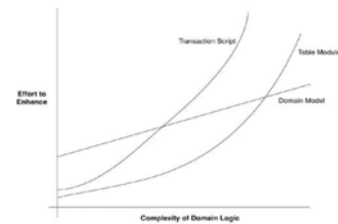
```

1 // Entity
2 public class Account {
3     // ID
4     private String id;
5     private double balance;
6     private OverdraftPolicy overdraftPolicy;
7 }
8
9 public double balance() { return balance; }
10 public void debit(double amount) {
11     this.balance -= amount;
12     this.overdraftPolicy.postDebit(this, amount);
13 }
14
15 public void credit(double amount) {
16     this.balance += amount;
17 }

```

Table Model

- Easy to understand/some re-use
- Hard for complicated business
- Poor performance
- Use for simple systems with limited change where want some re-use

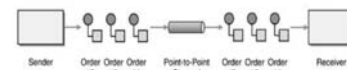


Messaging Channel Patterns

- Point-to-Point Channel Pattern
 - One-to-One
- Publish-Subscribe Channel Pattern
 - One-to-Many
 - Many-to-One
 - Many-to-Many

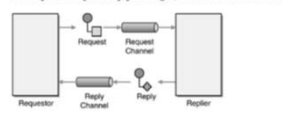
Point-to-Point (P2P) Messaging

Send the message on a *Point-to-Point Channel*, which ensures that only one receiver will receive a particular message.

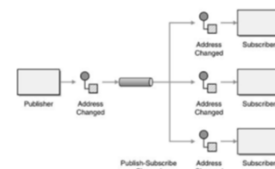


Request-Reply Using P2P

Send a pair of *Request-Reply* messages, each on its own channel.



Publish-Subscribe (Pub/Sub) Messaging



Lab

- After 4 labs, you should be able to:
 - Set up your development environment, including Maven, Spring, Hibernate, MySQL, Eclipse, etc.
 - Develop simple web applications
 - Create database tables
 - Insert and query your data in the database tables using Hibernate.