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
  - Laying 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!

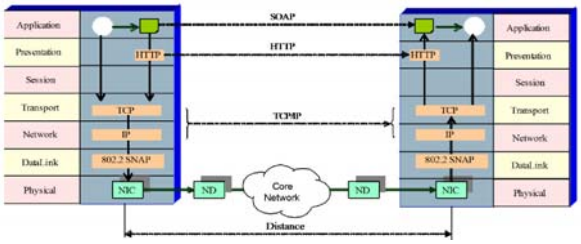
Classification of Design Patterns

- **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.

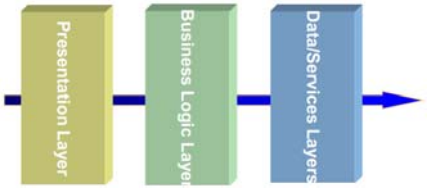
Laying Pattern - problems

- 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.

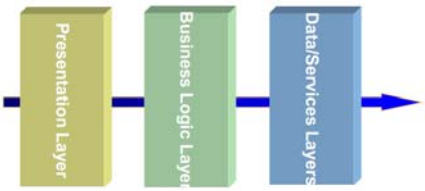
Domain/Business Logic Layer

- 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

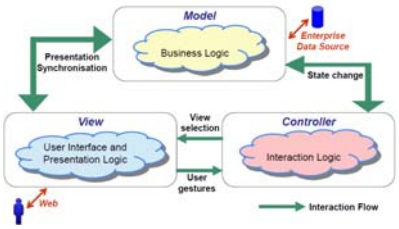
- Manage database
- Communicating with other systems that carry out tasks on behalf of the application.
  - Transaction monitors,
  - Other applications,
  - Messaging systems, and so forth.

Horizontal Laying Pattern

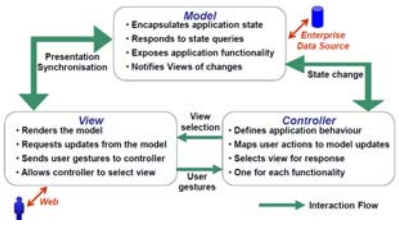


With the multi-tiers (layers), we can:
- Separate/isolate complexities from each other
- Maximize reuse of business components & data
- Making it easier to manage and scale

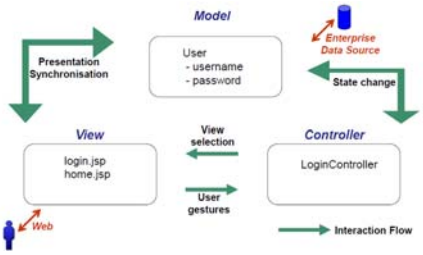
MVC Pattern



MVC Pattern



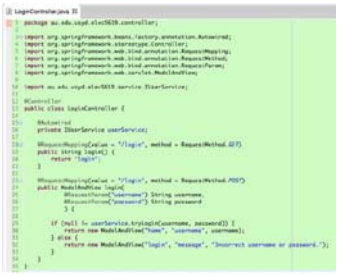
MVC Pattern Spring example



User Model

Login view

Login Controller



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

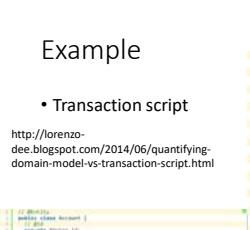
Domain Model

- 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

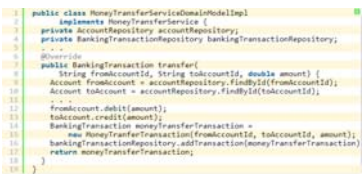
http://lorenzo-dee.blogspot.com/2014/06/quantifying-domain-model-vs-transaction-script.html



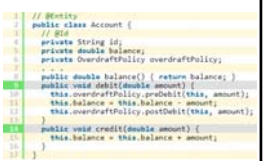
```
public interface MoneyTransferService {
    BankingTransaction transfer(
        String fromAccountId, String toAccountId, double amount);
}
```

```
public class MoneyTransferServiceTransactionScriptImpl
        implements MoneyTransferService {
    private AccountDao accountDao;
    private BankingTransactionRepository bankingTransactionRepository;
    . . .
    @Override
    public BankingTransaction transfer(
        String fromAccountId, String toAccountId, double amount) {
      Account fromAccount = accountDao.findById(fromAccountId);
      Account toAccount = accountDao.findById(toAccountId);
      . . .
      double newBalance = fromAccount.getBalance() - amount;
      switch (fromAccount.getOverdraftPolicy()) {
      case NEVER:
        if (newBalance < 0) {
          throw new DebitException("Insufficient funds");
        }
        break;
      case ALLOWED:
        if (newBalance < -limit) {
          throw new DebitException(
              "Overdraft limit (of " + limit + ") exceeded: " + newBalance);
        }
        break;
      }
      fromAccount.setBalance(newBalance);
      toAccount.setBalance(toAccount.getBalance() + amount);
      BankingTransaction moneyTransferTransaction =
          new MoneyTranferTransaction(fromAccountId, toAccountId, amount);
      bankingTransactionRepository.addTransaction(moneyTransferTransaction)
      return moneyTransferTransaction;
    }
}
```

Example domain model



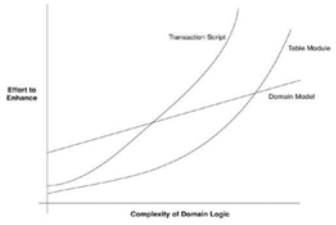
```
public class MoneyTransferServiceDomainModelImpl
        implements MoneyTransferService {
    private AccountRepository accountRepository;
    private BankingTransactionRepository bankingTransactionRepository;
    . . .
    @Override
    public BankingTransaction transfer(
        String fromAccountId, String toAccountId, double amount) {
      Account fromAccount = accountRepository.findById(fromAccountId);
      Account toAccount = accountRepository.findById(toAccountId);
      . . .
      fromAccount.debit(amount);
      toAccount.credit(amount);
      BankingTransaction moneyTransferTransaction =
          new MoneyTranferTransaction(fromAccountId, toAccountId, amount);
      bankingTransactionRepository.addTransaction(moneyTransferTransaction)
      return moneyTransferTransaction;
    }
}
```



```
// @Entity
public class Account {
  // @Id
  private String id;
  private double balance;
  private OverdraftPolicy overdraftPolicy;
  . . .
  public double balance() { return balance; }
  public void debit(double amount) {
    this.overdraftPolicy.preDebit(this, amount);
    this.balance = this.balance - amount;
    this.overdraftPolicy.postDebit(this, amount);
  }
  public void credit(double amount) {
    this.balance = this.balance + amount;
  }
}
```

Table Model

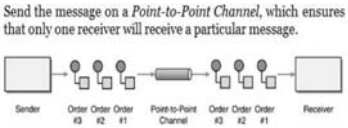
- Easy to understand/some re-use
- Hard for complicated business
- Poor performance
- Use for simple systems with limited change where want some re-use



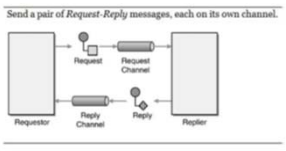
Messaging Channel Patterns

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

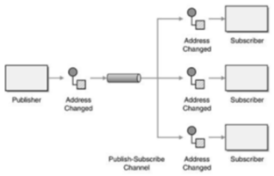
Point-to-Point (P2P) Messaging



Request-Reply Using P2P



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.