

Applying Design Patterns in Java

Duration:	3 days
Type:	intermediate

Description

This course is designed for developers who want to extend their architectural skills using design patterns and related techniques. It is heavily interactive with delegates spending the majority of their time creating pattern based solutions to real world problems.

Each pattern is introduced in terms of its abstract structure (via UML diagrams), its benefits and drawbacks, sample implementations and practical examples of how it can be used to simplify software development.

The course makes extensive use of JSE libraries and JEE frameworks as examples of pattern based architecture. Examples are also drawn from other programming languages (C++, C#, and Ruby) where appropriate.

Prerequisites

Delegates should be experienced Java developers familiar with UML Sequence and Class diagrams.

List of Modules

Introduction to Patterns

- What is a design pattern?
- The evolution of design patterns
- Misconceptions about design patterns
- The dangers of becoming 'pattern happy'
- Distinguishing between patterns, idioms and refactorings
- Using refactorings to introduce patterns incrementally
- Using patterns to create an object oriented architecture

Proxy

- Adding services by intercepting messages
- Dynamically creating proxy classes in Java via Reflection
- How proxy objects are used in Enterprise JavaBeans
- How proxy objects are used to implement AOP in Spring
- The interception framework provided with .NET
- Interception functionality built into Ruby

Factory

- Advantages of separating clients from object creation
- Comparing the Factory Method and Abstract Factory Patterns
- How Factory Method is used with writing custom class loaders
- Using Abstract Factory to conceal XML Parsers and DB Drivers
- Using Factory and Proxy to control access to pooled resources
- Extending the Factory Pattern into Dependency Injection
- Examples of Dependency Injection Tools (Spring and Guice)

Builder

- Simplifying the creation of complex object trees
- How Groovy uses Builder to generate XML documents

Composite

- Modelling nested whole-part relationships in OO
- Examples of Composite in XML and GUI libraries

Singleton

- Why ensure a class only has a single instance?
- General problems implementing Singleton objects
- Language specific problems with Singletons (Java/C#/C++)
- Allowing singletons to be reborn (Phoenix Singleton pattern)
- Making Singletons safe and efficient in multithreaded environments
- Problems with the double checked locking idiom

Strategy

- Creating class hierarchies to represent algorithms
- Separating a class from a changing or complex algorithm
- The Strategy Pattern, functor objects and threading libraries
- Strategy and Layout Managers in Swing and SWT
- How Strategy is used in the JMock mock object generator

Command

- Similarities between Strategy and Command
- Using Command objects to simplify event dispatching
- Incrementally refactoring code to introduce Command
- Uses of the Command Pattern in JEE Web Frameworks

Template

- Using polymorphism to customize algorithms
- Similarities between Template and Factory Method
- The Template Pattern and rendering Swing components
- Template and the Struts JEE Web Framework

Decorator

- Using composition to layer extra functionality
- Applying Decorator to create specialized collections
- Decorator as the basis of the Java I/O libraries

Adapter

- Distinguishing between Adapter and Decorator
- Uses of adapter in the Java I/O libraries

Iterator

- Accessing an aggregate object without knowing its representation
- How Iterator is used in the STL and the Java and .NET collections
- Adding iterator support to your own collections

Observer

- Informing interested objects of state changes
- Benefits and dangers in implementing Observer
- The Observer Pattern and event handling in Java
- The Observer Pattern, delegates and events in C#

Visitor

- Simplifying class design by modelling operations as visitors
- Adding support for Visitor to existing collections of objects
- Using Visitor to add reporting and logging behaviour
- Code generation in Java with Visitor and the ASM library
- Using Visitor in the design of a mock objects generator

State

- Benefits of the State Pattern over subclassing
- Modelling objects with complex internal state transitions
- Different approaches to implementing state transitions
- Automatically generating state machines

Threading Patterns

- Implementing the Active Object Pattern in Java, C# and C++
- Worker thread pool implementations in Java and C#
- Introducing the IOU (Asynchronous Completion Token) Pattern
- Futures in Java and Asynchronous Delegates in C#