CS525 Advanced Software Development

Lesson 1 – Introduction

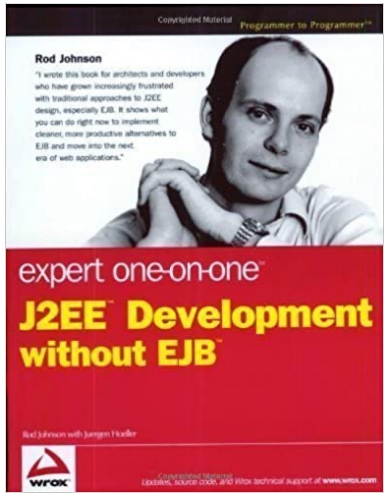
Design Patterns
Elements of Reusable Object-Oriented Software

Payman Salek, M.S.
March 2022

Motivation

Expert One-on-One:

J2EE™ Development without EJBs



Rod Johnson and Juergen Hoeller

The creators of the Spring Framework

Motivation – Simplicity is Good!

Developers tend to think that because complex technologies like EJB exist, they ought to use them whether it's really necessary or not. Good engineering practice doesn't mean using the most complex approach. The more complex an application is the longer it takes to develop, the harder it is to maintain and the worse it's likely to perform. - Rod Johnson

Motivation – Design Patterns

“My advice to developers is: read the Gang of Four patterns book from cover to cover at least once a year. This contains 90% of what you need to know about patterns and is guaranteed to make you a better OO programmer every time you read it.”

- Rod Johnson

Introduction

Designing object-oriented software is hard and designing reusable object-oriented software is even harder.

You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them.

Introduction

Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it.

Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Object Oriented Foundations

- Abstraction
- Inheritance
- LSP – Liskov Substitution Principle
- Polymorphism

Power of Abstraction

Fundamental theorem of Computer Science:

"We can solve any problem by introducing an extra level/layer of indirection [abstraction]."

David Wheeler
British Computer Scientist

Designing software is hard, and designing reusable software is even harder

You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it.

Design Patterns: Elements of Reusable Object-Oriented Software

Design Re-Use

We all know the value of design experience. How many times have you had design déjàvu—that feeling that you've solved a problem before but not knowing exactly where or how? If you could remember the details of the previous problem and how you solved it, then you could reuse the experience instead of rediscovering it.

Design Patterns: Elements of Reusable Object-Oriented Software

What is a Design Pattern?

Christopher Alexander says, “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

Design Patterns: Elements of Reusable Object-Oriented Software

What is a Design Pattern?

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Inheritance

Class inheritance is basically just a mechanism for extending an application's functionality by reusing functionality in parent classes. It lets you define a new kind of object rapidly in terms of an old one. It lets you get new implementations almost for free, inheriting most of what you need from existing classes."

Design Patterns: Elements of Reusable Object-Oriented Software

Interfaces

Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces. There is no way to know anything about an object or to ask it to do anything without going through its interface. An object's interface says nothing about its implementation—different objects are free to implement requests differently. That means two objects having completely different implementations can have identical interfaces.

Design Patterns: Elements of Reusable Object-Oriented Software

Program to Interface, not Implementation

When inheritance is used carefully (some will say properly), all classes derived from an abstract class will share its interface. This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class. All subclasses can then respond to the requests in the interface of this abstract class, making them all subtypes of the abstract class.

Design Patterns: Elements of Reusable Object-Oriented Software

Dynamic Binding

When a request is sent to an object, the particular operation that's performed depends on both the request and the receiving object. Different objects that support identical requests may have different implementations of the operations that fulfill these requests. The run-time association of a request to an object and one of its operations is known as dynamic binding.

Design Patterns: Elements of Reusable Object-Oriented Software

Polymorphism

Dynamic binding lets you substitute objects that have identical interfaces for each other at run-time. This substitutability is known as Liskov Substitution Principle and results in polymorphism, and it's a key concept in object-oriented systems.

Design Patterns: Elements of Reusable Object-Oriented Software

Polymorphism - Definition

Polymorphism is dynamic message to method binding in the presence and with the aid of LSP and sub-type abstraction.

Favor composition over inheritance

First, you can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time. Second, and generally worse, parent classes often define at least part of their subclasses' physical representation. Because inheritance exposes a subclass to details of its parent's implementation, it's often said that "inheritance breaks encapsulation" [Sny86]. The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change.

Design Patterns: Elements of Reusable Object-Oriented Software

Composition + Delegation

Delegation is a way of making composition as powerful for reuse as inheritance.

The main advantage of delegation is that it makes it easy to compose behaviors at run-time and to change the way they're composed.

Design Patterns: Elements of Reusable Object-Oriented Software

When to use Delegation

Delegation is a good design choice only when it simplifies more than it complicates. It isn't easy to give rules that tell you exactly when to use delegation, because how effective it will be depends on the context and on how much experience you have with it. Delegation works best when it's used in highly stylized ways—that is, in standard patterns.

Design Patterns: Elements of Reusable Object-Oriented Software

Parametrized Types (AKA Generics)

Another (not strictly object-oriented) technique for reusing functionality is through parameterized types, also known as generics. This technique lets you define a type without specifying all the other types it uses.

Parameterized types give us a third way (in addition to class inheritance and object composition) to compose behavior in object-oriented systems.

Design Patterns: Elements of Reusable Object-Oriented Software

Inheritance vs. Composition vs. Generics

Object composition lets you change the behavior being composed at run-time, but it also requires indirection and can be less efficient. Inheritance lets you provide default implementations for operations and lets subclasses override them. Parameterized types let you change the types that a class can use. But neither inheritance nor parameterized types can change at run-time. Which approach is best depends on your design and implementation constraints.

Design Patterns: Elements of Reusable Object-Oriented Software

Toolkits/Libraries

A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. An example of a toolkit is the Java Collections API which is basically a set of collection classes for lists, sets, and the like.

Design Patterns: Elements of Reusable Object-Oriented Software

Toolkits/Libraries

Toolkits don’t impose a particular design on your application; they just provide functionality that can help your application do its job. They let you as an implementer avoid recoding common functionality. Toolkits emphasize code reuse.

Design Patterns: Elements of Reusable Object-Oriented Software

Frameworks

A framework is a set of cooperating classes that make up a reusable design for a specific class of software. You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework.

Design Patterns: Elements of Reusable Object-Oriented Software

Frameworks

The framework dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses you can put to work immediately.

Design Patterns: Elements of Reusable Object-Oriented Software