C++ Design Patterns: Creational

Preliminaries

Dmitri Nesteruk

@dnesteruk dn@activemesa.com http://activemesa.com

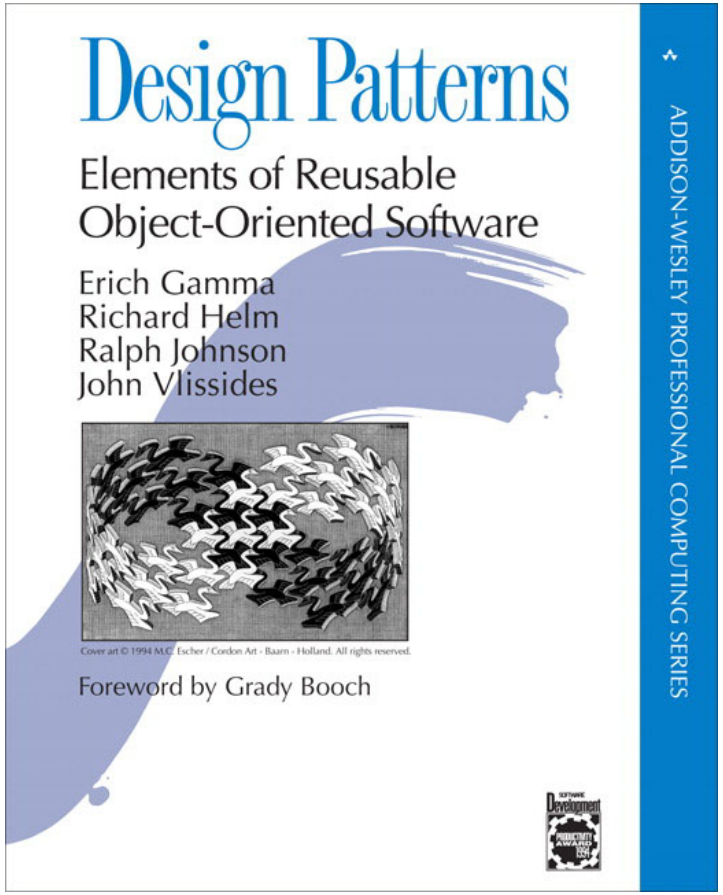
Introduction

Design patterns are common architectural approaches

Popularized by the Gang of Four book (1994)
Smalltalk & old C++

Translated to many OOP languages
C#, Java, …

Universally relevant
Internalized in some other languages
Language extensions (non-standard C++)
Libraries



Course Overview

- First in a series of courses on C++ Design Patterns
  - Covers creational patterns
- Covers every pattern from GoF book
  - Motivation
  - Classic implementation
  - Pattern variations
  - Library implementations
  - Pattern interactions
  - Important considerations (e.g., testability)
- Patterns demonstrated via live coding!

Demonstrations

- Uses modern C++ (C++11/14/17)
- Demos use Microsoft Visual Studio 2015, MSVC, ReSharper C++
- Some simplifications:
  - Classes are often defined inline (no .h/.cpp separation)
  - Pass by value everywhere
  - Liberal import of namespaces (e.g., `std::`) and headers

Course Structure

Preliminaries SOLID, DI, testing, libraries…

Builder

Factories Factory Method, Abstract Factory

Prototype

Singleton

Preliminaries

Dmitri Nesteruk

@dnesteruk dn@activemesa.com http://activemesa.com

Module Overview

SOLID Design Principles

Dependency Injection

Monads Maybe

SOLID Design Principles

- **S**ingle Responsibility Principle (SRP)
  - A class should only have a single responsibility
- **O**pen-Closed Principle (OCP)
  - Entities should be open for extension but closed for modification
- **L**iskov Substitution Principle (LSP)
  - Objects should be replaceable with instances of their subtypes without altering program correctness
- **I**nterface Segregation Principle (ISP)
  - Many client-specific interfaces better than one general-purpose interface
- **D**ependency Inversion Principle (DIP)
  - Dependencies should be abstract rather than concrete

Dependency Inversion Principle

- Summary
  1. **High-level modules should not depend on low-level modules. Both should depend on abstractions.**
     Example: reporting component should depend on a ConsoleLogger, but can depend on an ILogger.
  2. **Abstractions should not depend upon details. Details should depend upon abstractions.**
     In other words, dependencies on interfaces and supertypes is better than dependencies on concrete types.
- **Inversion of Control (IoC)** – the actual process of creating abstractions and getting them to replace dependencies.
- **Dependency Injection** – use of software frameworks to ensure that a component's dependencies are satisfied.

Monads

- Monads are design patterns in *functional* programming
- Much more usable in functional languages due to
  - Better treatment of functional objects
  - Useful auxiliary constructs (e.g., pattern matching)
- Somewhat implementable in OOP languages

Summary

Observe SOLID principles

DI simplifies expressing dependencies

The world is not only OOP!