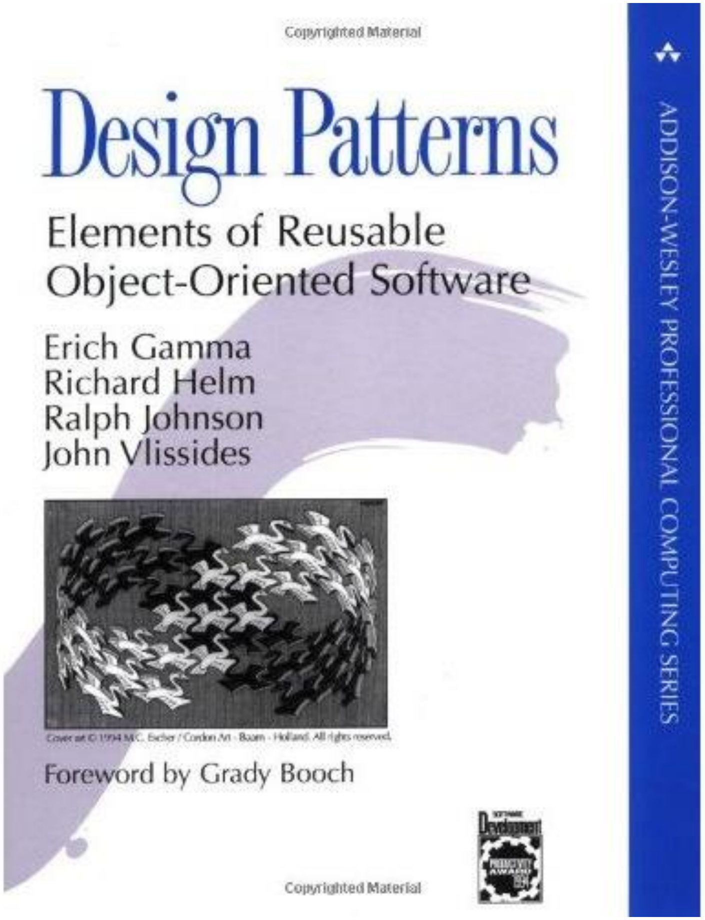
Introduction to Design Patterns

CS 345 Winter 2018

Chris Reedy

Design Patterns: Reusing Good Designs

- As we learn more we learn design “tricks”
  - ways to solve (or not to solve) certain design problems
  - answers to “how do we do that?”
- Design patterns is an organized approach to reusing design “tricks”
- Classic Reference (1994): *Design Patterns, Elements of Reusable Object-Oriented Software*
  - Still relevant



Copyrighted Material

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

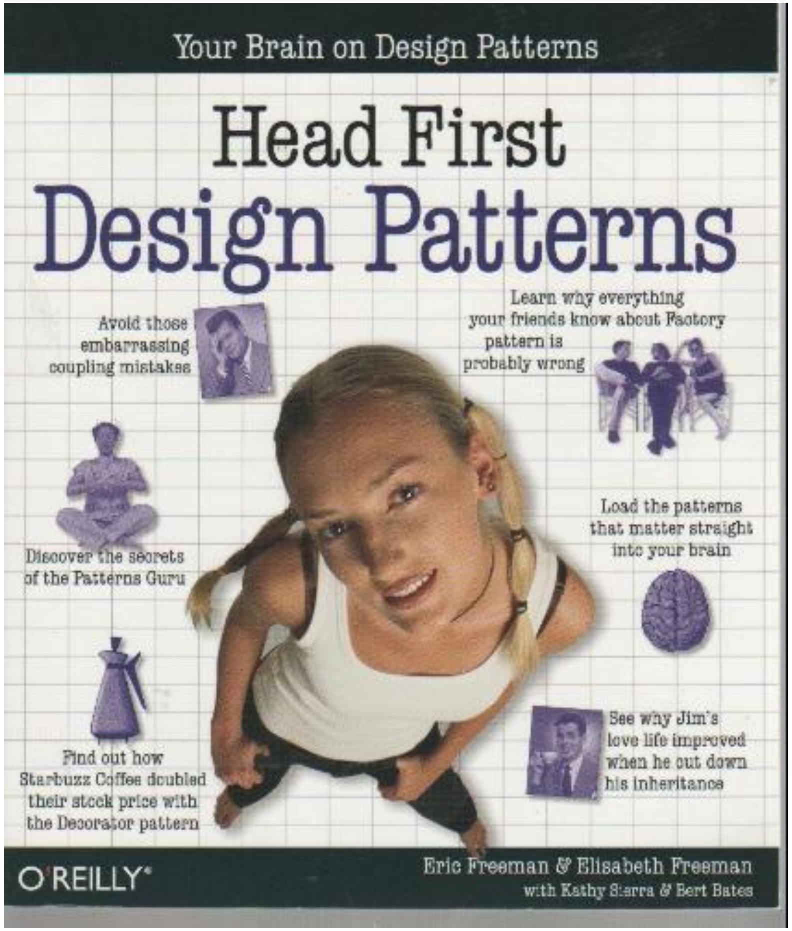
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

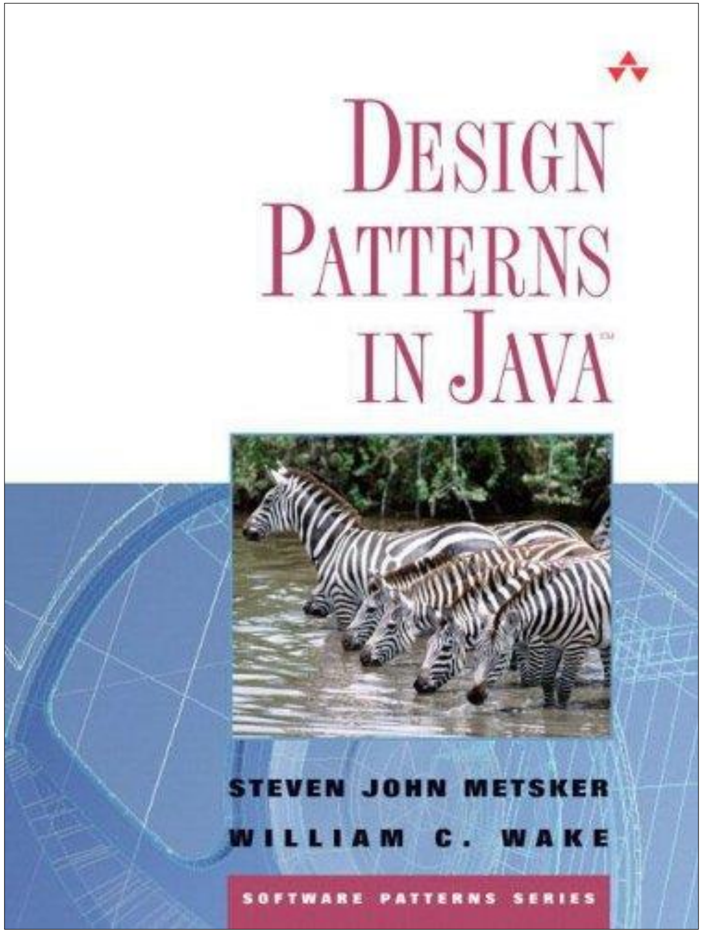
Foreword by Grady Booch

Copyrighted Material

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Additional References


Your Brain on Design Patterns
Head First
Design Patterns
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates



What is a design pattern?

"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."

- Christopher Alexander

(http://en.wikipedia.org/wiki/Christopher_Alexander)

What are the parts of a Design Pattern?

- Name
  - A good name is a handle to the solution
  - A good name lets us communicate quickly and easily about the form of the solution
  - E.g. “Use an Observer here.”
    - “Observer” is a design pattern we will discuss later
- Problem
- Solution
- Consequences

Design Pattern: The Problem

When do you use the pattern?

- What is the problem you are trying to solve?
- What is the context?
  - Situations where the pattern can be applied
  - Things which must be true before it makes sense to apply the pattern
  - Examples of poor design that the pattern addresses

Design Pattern: The Solution

- Components that make up the pattern
- How the components work
  - Relationships
  - Responsibilities
  - Collaborations
  - Interactions
  - Sequences of events
- UML Diagrams are frequently used here

Design Pattern: The Solution (2)

- A design pattern usually does not describe a particular concrete implementation
- A design pattern is like an outline or “template”
    - Not (necessarily) a programming language template
    - Describes a general arrangement of components to solve a design problem
- Some design patterns have been partially or fully implemented in libraries

Design Patterns: Consequences

- Results and trade-offs
  - Costs and benefits
  - What you gain and what you lose
- Some possible issues
  - Space and time trade-offs
  - Understandability versus performance
  - Impacts on flexibility, portability, etc.

Kinds of Design Patterns

- Patterns discussed here are specifically Object-Oriented patterns.
- Other kinds of design patterns:
  - Architectural – describe the structure of an entire system
  - User Interface
  - Information Visualization
  - Security
  - …

(Software) Design Patterns are not

- Data structures
  - Hash tables, linked lists, …
- Algorithms (usually)
  - E.g. How to sort
- However, a design pattern may make use of a data structure or algorithm

A Catalog of Object Design Patterns (from Wikipedia, “Software Design Pattern”)

- Creational
  - Abstract Factory*
  - Builder*
  - Factory Method*
  - Lazy Initialization*
  - Multiton
  - Object Pool
  - Prototype*
  - Resource Acquisition is Initialization
  - Singleton*
- Structural
  - Adapter (Wrapper, Translator)*
  - Bridge*
  - Composite*
  - Decorator*
  - Façade*
  - Flyweight*
  - Front Controller
  - Marker
  - Module
  - Proxy*
  - Twin
- Behavioral
  - Blackboard
  - Chain of Responsibility*
  - Command*
  - Interpreter*
  - Iterator*
  - Mediator*
  - Memento*
  - Null Object
  - Observer* (Publish/Subscribe)
  - Servant
  - Specification
  - State*
  - Strategy*
  - Template Method*
  - Visitor*

* Gang of Four (GOF) pattern from original Design Patterns book

Creational Patterns

- Factory Method*
  - Define an interface for creating an object, but retain control of which class to instantiate.
- Abstract Factory*
  - Create families of related or dependent objects without specifying their concrete classes.
- Singleton*
  - Ensure a class has only one instance. Provide a global point of access.
- Multiton
  - Ensure a class has only named instances, and provide a global point of access to them.
- Builder*
  - Separate construction of a complex object from its representation.

Creational Patterns (Continued)

- Prototype*
  - Specify kinds of objects to create by copying a prototypical instance.
- Lazy Initialization*
  - Delay creation of an object, calculation of a value, or some expensive process until the first time it is needed.
    - GOF: “Virtual proxy” implementation strategy for Proxy.
- Object Pool
  - Avoid acquisition and release of resources by recycling objects that are no longer in use.
- Resource Acquisition is Initialization (RAII)
  - Tie resources that need to be released to the lifespan of a suitable object.

Structural Patterns

- Adapter (Wrapper, Translator)*
  - Convert the interface of a class into another interface expected by clients.
- Bridge*
  - Decouple an interface abstraction from its implementation allowing them to vary independently.
- Composite*
  - Compose objects into tree structures representing part-whole hierarchies. Treat individuals and compositions uniformly.
- Decorator*
  - Dynamically attach additional responsibilities to an object while keeping the same interface.
- Façade*
  - Provide a unified interface to a collection of interfaces for a subsystem.

Structural Patterns (continued)

- Flyweight*
  - Use sharing to support large numbers of fine grained objects.
- Front Controller
  - Provide a centralized entry point for handling requests.
- Marker
  - Associate metadata with a class
- Module
  - Group related elements (classes, singletons, methods, etc.) into a single conceptual entity.
- Proxy*
  - Provide a surrogate or placeholder for another object.
- Twin
  - Model multiple inheritance in languages that don’t support this.

Behavioral Patterns

- Chain of Responsibility*
  - Chain receivers of a request and pass request along until it is handled. Decouples sender of a request from its receiver.
- Command*
  - Encapsulate a request as an object.
- Interpreter*
  - Given a “language”, build an interpreter for that language.
- Iterator*
  - Access elements of an aggregate (a collection) sequentially without exposing the underlying implementation.
- Mediator*
  - Define an object that encapsulates how objects interact. Decouples objects and allows dynamic control of interactions.

Behavioral Patterns (continued)

- Memento*
  - Externalize an object’s internal state so that the object can be restored to that state at a later point in time.
- Null Object
  - Avoid null references by providing a default “null” object.
- Observer (Publish/Subscribe)*
  - A state change in an object causes all dependents to be automatically notified and updated.
- Blackboard
  - A generalized Observer allowing multiple readers and writers that communicate information system-wide.
- Servant
  - Define common functionality for a group of classes.

Behavioral Patterns (continued)

- Specification
  - Provide combinable boolean business logic.
- State*
  - Allow an object to change behavior when internal state changes. The object appears to change class.
- Strategy*
  - Define an interchangeable family of algorithms. Decouples the implementation of the algorithm from clients that use it.
- Template Method*
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Subclasses can provide/redefine certain steps in an algorithm without changing the overall algorithm.
- Visitor*
  - Represent an operation to be performed on the elements of an object structure. Define a new algorithm without having to change the implementation of the classes on which it operates.

Some Criticisms of Design Patterns

- The use of design patterns is a sign of missing features of a programming language.
  - In Java, you are forced to distinguish between constructors and functions that return “new” objects.
  - In Python, the distinction doesn’t exists.
    - The class object is callable as a function returning a new object.
  - The Visitor pattern is a solution to the fact that you can’t add new methods to an existing class.
- Design patterns force the developer to be a compiler.
- Inappropriate use of design patterns can easily lead to increased complexity and poor performance.

Some Web References

- The Portland Pattern Repository
  - http://c2.com/ppr/
- Wikipedia, start here:
  - http://en.wikipedia.org/wiki/Software_design_pattern