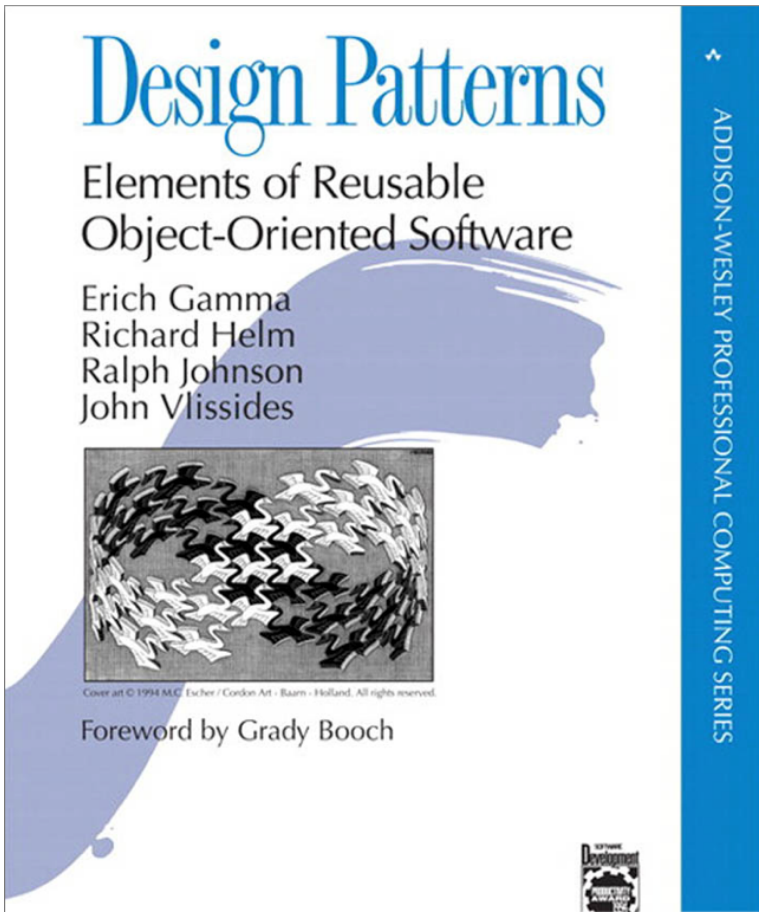
Design Patterns

Christopher Simpkins
chris.simpkins@gatech.edu

Design Patterns



A recurring object-oriented design.

- Make proven techniques more accessible to developers of new systems – don't have to study other systems.
- Helps in choosing designs that make the system more reusable.
- Facilitate documenentation and communication with other developers.

Design pattern catalog: descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

Elements of Design Patterns

- The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
- The **problem** describes when to apply the pattern.
- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The pattern provides an abstract description of a design problem and how a general arrangement of classes and objects solves it.
- The **consequences** are the results and trade-offs of applying the pattern.

How Design Patterns Solve Design Problems

- Finding appropriate objects
- Determining object granularity
- Specifying object interfaces
- Specifying object implementations

Types versus Classes

Class versus interface inheritance

- An object's class defines how the object is implemented.
  - Class inheritance defines an object's implementaion in terms of another object's implementation.
- An object's type refers to its interface – the set of methods it can respond to.
  - Interface inheritance is sybtyping. It describes when an object can be used in place of another.

Program to an interface, not an implementation.

Reuse Mechanisms

Inheritance versus composition

- Inheritance: “White box reuse” – subclass reuses details of superclass and extends with new functionality
  - Defined at compile-time.
  - Straightforward to use
  - “Inheritace breaks encapsulation” (superclass implementaiotn exposed to subclasses)
  - Reuse can be difficult in new contexts – may require rewriting superclasses or carrying baggage.
- Composition: “Black-box reuse” – new functionality obtained by composing objects of other objects
  - Defined at run-time by objects acquiring references to other objects.
  - Must program to interfaces, so interfaces must be well thought-out and stable.
  - Emphasis on interface stability encourages granular objects with single responsiblities.

Favor object composition over class inheritance.

Designing for Change

Common causes of redesign (and design patterns that address them):

- Creating an object by specifying a class explicitly. (Factory)
- Dependence on specific operations. (Command)
- Dependence on hardware and software platform. (Factory, Bridge).
- Dependence on object representations or implementations (Factory, Bridge, Proxy).
- Algorithmic dependencies. (Visitor, Iterator, Strategy, Template Method)
- Tight coupling. Design patterns: (Factory, Bridge, Command, Facade, Mediator, Observer).
- Extending functionality by subclassing. (Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy)
- Inability to alter classes conveniently. (Adapter, Decorator, Visitor)

Selecting a Design Pattern

- Consider how design patterns solve design problems.
- Scan Intent sections. Read through each pattern's intent to find one or more that sound relevant to your problem.
- Study how patterns interrelate. Studying these relationships can help direct you to the right pattern or group of patterns.
- Study patterns of like purpose.
- Examine a cause of redesign, look at the patterns that help you avoid the causes of redesign.
- Consider what should be variable in your design. This approach is the opposite of focusing on the causes of redesign. Instead of considering what might force a change to a design, consider what you want to be able to change without redesign. The focus here is on encapsulating the concept that varies, a theme of many design patterns.

Using Design Patterns (1 of 2)

- Read the pattern once through for an overview.
- Go back and study the Structure, Participants, and Collaborations sections. Make sure you understand the classes and objects in the pattern and how they relate to one another.
- Look at Sample Code to see a concrete example of the pattern in code.
- Choose names for pattern participants that are meaningful in the application context. OK to use abstract participant names from design pattern. For example, if you use the Strategy pattern for a text compositing algorithm, then you might have classes SimpleLayoutStrategy or TeXLayoutStrategy.

Using Design Patterns (2 of 2)

- Define the classes. Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify existing classes in your application that the pattern will affect, and modify them accordingly.
- Define application-specific names for operations in the pattern. Use the responsibilities and collaborations associated with each operation as a guide. Be consistent in your naming conventions. For example, you might use the "Create-" prefix consistently to denote a factory method.
- Implement the operations to carry out the responsibilities and collaborations in the pattern. The Implementation section from a pattern catalog and sample code offers hints to guide you in the implementation.

Common Design Patterns

- **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes. (GoF, 87)
- **Factory Method** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. (GoF, 107)
- **Adapter** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompat ible interfaces. (GoF, 139)
- **Composite** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. (GoF, 163)

Common Design Patterns

- **Decorator** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. (GoF, 175)
- **Observer** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. (GoF, 293)
- **Strategy** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. (GoF, 315)
- **Template Method** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. (GoF, 325)