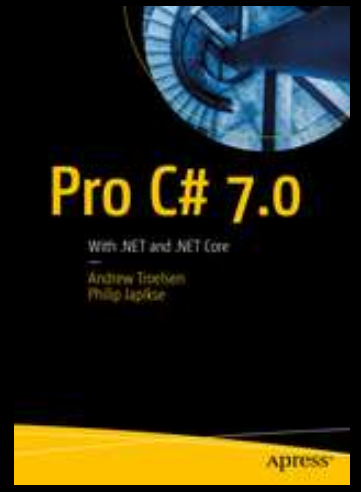
DESIGN PATTERNS FOR MERE MORTALS

Philip Japikse (@skimedic)
skimedic@outlook.com
www.skimedic.com/blog
Microsoft MVP, ASPInsider, MCSD, MCDBA, CSM, CSP
Consultant, Teacher, Writer

Phil.About()

- Consultant, Coach, Author, Teacher
  - Lynda.com (http://bit.ly/skimediclyndacourses)
  - Apress.com (http://bit.ly/apressbooks)
- Microsoft MVP, ASPInsider, MCSD, MCDBA, CSM, CSP
- Founder, Agile Conferences, Inc.
  - http://www.dayofagile.org
- President, Cincinnati .NET User's Group



“The goal is not to bend developers to the will of some specific patterns, but to get them to think about their work and what they are doing”

--Phil Haack

WHAT ARE DESIGN PATTERNS?

- General Reusable Solutions To A Common Problem
- Conceptual
- Defined by Purpose and Structure
- Method of Communication
- Support SOLID development
- NOT CODE!

TYPES OF DESIGN PATTERNS

- Creational
  - Deal with instantiation of objects (Singleton, Factories)
- Structural
  - Deal with Composition and Relations (Adapter, Decorator, Façade)
- Behavioral
  - Deal with responsibilities and communication between objects (Command, Strategy)

BEHAVIORAL

THE STRATEGY PATTERN

- The **strategy pattern** (also known as the **policy pattern**) is a behavioral software design pattern that enables an algorithm's behavior to be selected at runtime. The strategy pattern:
  - defines a family of algorithms,
  - encapsulates each algorithm, and
  - makes the algorithms interchangeable within that family.

- Promotes the Open/Closed principle by using Composition over Inheritance

- Examples:
  - Sorting (with custom comparer)
  - Log4Net Fallback Appender

https://en.wikipedia.org/wiki/Strategy_pattern

THE OBSERVER PATTERN

- The **observer pattern** is a pattern in which an object, called the **subject**, maintains a list of its dependents, called **observers**, and notifies them automatically of any state changes, usually by calling one of their methods.
- Can cause memory leaks due to strong references
  - This is mitigated with weak references

https://en.wikipedia.org/wiki/Observer_pattern

THE PUBLISH-SUBSCRIBE PATTERN

- In software architecture, **publish–subscribe** is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead characterize published messages into classes without knowledge of which subscribers, if any, there may be.
- Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

https://en.wikipedia.org/wiki/Publish–subscribe_pattern

OBSERVER VS PUB-SUB

Observer

- The recipient knows the sender
- The sender knows the recipient
- Send and receive one at a time
- Direct communication

Pub-Sub

- Sender and recipients unknown to each other
- Send once, every subject receives.
- Intermediary handles filtering and routing

THE COMMAND PATTERN

- The **command pattern** is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time.
- Can optionally add Undo
  - Tracked by either the command or the controller
- Used in conjunction with many other patterns:
  - Factory method, Template method

https://en.wikipedia.org/wiki/Command_pattern

THE MEMENTO PATTERN

- The **memento pattern** provides the ability to restore an object to its previous state (undo via rollback).
- The memento pattern is implemented with three objects: the *originator*, a *caretaker* and a *memento*.
  - The originator is some object that has an internal state.
  - The caretaker is going to do something to the originator, but wants to be able to undo the change. It asks for a memento object.
  - Then it does whatever operation (or sequence of operations) it is going to do.
  - To roll back to the state before the operations, it returns the memento object to the originator.

https://en.wikipedia.org/wiki/Memento_pattern

THE TEMPLATE METHOD PATTERN

- The **template method pattern** defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses.
  - It lets one redefine certain steps of an algorithm without changing the algorithm's structure.
- In the template **method**, one or more algorithm steps can be overridden by subclasses to allow differing behaviors while ensuring that the overarching algorithm is still followed.

https://en.wikipedia.org/wiki/Template_method_pattern

CREATIONAL

THE SINGLETON PATTERN

- The **singleton pattern** is a software design pattern that restricts the instantiation of a class to one instance and provides global access to that instance.
- Commonly used to implement many other patterns:
  - Factories (abstract and simple), façade, state, builder, and prototype
- Many DI frameworks provide singleton support
  - You need to understand how the DI f/w works to prevent issues

https://en.wikipedia.org/wiki/Singleton_pattern

THE SIMPLE FACTORY ~~PATTERN~~

- Not a “true” pattern
- Encapsulates object creation in one place
  - Should be the only part of the application that refers to concrete classes
- Reduces duplicate code by enforcing DRY

THE FACTORY METHOD PATTERN

- The *factory method pattern* uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.
- This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

https://en.wikipedia.org/wiki/Factory_method_pattern

THE ABSTRACT FACTORY PATTERN

- The **abstract factory pattern** provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.
- In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme.
- This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.

https://en.wikipedia.org/wiki/Abstract_factory_pattern

THE PROTOTYPE PATTERN

- The **prototype pattern** is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used to:
  - avoid subclasses of an object creator in the client application, like the abstract factory pattern does.
  - avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) when it is prohibitively expensive for a given application.

https://en.wikipedia.org/wiki/Prototype_pattern

STRUCTURAL

THE ADAPTER AND FAÇADE PATTERNS

- The **adapter pattern** allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifing code.
- A **facade** is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can:
  - Make a software library easier to use, understand, and test, since the facade has convenient methods for common tasks,
  - Reduce dependencies of outside code on the inner workings of a library
  - Wrap a poorly designed collection of APIs with a single well-designed API.

https://en.wikipedia.org/wiki/Adapter_pattern
https://en.wikipedia.org/wiki/Facade_pattern

THE DECORATOR PATTERN

- The **decorator pattern** that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.
- Provides an alternative to subclassing for extending functionality.
- Supports the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.

https://en.wikipedia.org/wiki/Decorator_pattern

RESOURCES

- “Design Patterns: Elements of Reusable Object Oriented Design”
  - Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides
- “Head First Design Patterns”
- www.dofactory.com

DEMO

Patterns in C#

Contact Me

skimedic@outlook.com
www.skimedic.com/blog
www.twitter.com/skimedic

http://bit.ly/skimediclyndacourses
http://bit.ly/apressbooks

www.hallwayconversations.com

Thank You!

https://github.com/skimedic/presentations