# Design Patterns

# Notes from – ‘Head First Design Patterns’

## Why?

Design patterns tell us how to structure classes and objects to solve certain problems and it is our job to adapt those designs to fit our application.

Patterns show how to build systems with good OO design qualities.

Patterns are proven object-oriented experience.

Patterns don’t give you code, they give you general solutions to design problems.

Patterns provide a shared language that can maximize the value of your communication with other developers.

## Defining Design Patterns

A Pattern is a solution to a problem in a context.

**Context –** Situation in which the pattern applies. This should be a recurring situation.

**Problem –** Refers to the goal you are trying to achieve in this context, but also refers to any constraints that occur in the context.

**Solution –** End result. A general design that anyone can apply which resolves the goal given the context.

# Organising Design Patterns

There are 3 distinct categories for Design Patterns: Creational, Behavioural, Structural.

**Creational Patterns** involve object instantiation, and all provide a way to decouple a client from the objects it needs to instantiate.

**Behavioural Patterns** are concerned with how classes and objects interact and distribute responsibility.

**Structural Patterns** let you compose classes or objects into larger structures.

## Thinking in Design Patterns

**KISS** – Solve things the simplest way possible. You goal should be simplicity, not ‘How do I apply a pattern?’.

**Not a ‘Magic Bullet’** – Cannot simply choose a pattern, write some classes, compile & go for lunch! To use patterns successfully, also consider the consequences for the rest of the design.

**Know when a pattern is needed** – Only introduce patterns when you are *sure* it addresses a problem in the design. If a simpler solution might work, give that consideration too.

**Refactoring** – The goal of refactoring is to improve structure, not change the behaviour therefore it is a good time to overview the system and consider patterns.

**Removing Patterns is OK** – When the system is complex and the planned for flexibility isn’t needed, time to see if it’s simpler without patterns.

**If you don’t need it now, don’t do it now** – If the reason behind adding a pattern is hypothetical, don’t add it now!

# Design Principles

* Identify the aspects of the application that vary and separate them from what stays the same.
* Program to an interface, not an implementation.
* Favour composition over inheritance.
* Strive for loosely coupled designs between objects that interact.
* Classes should be open for extension but closed for modification.
* Depend on abstractions. Do not depend on concrete classes.
* Talk only to friends (Law of Demeter / Principle of Least Knowledge)
* Don’t call us, we’ll call you!
* A class should only have one reason to change.

# Dependency Inversion Principle

High-level components should not depend on low-level components - they should *both* depend on abstractions.

* **No variable should hold a reference to a concrete class** - If **new** is used, there will be a reference to a concrete class. Use a factory to get around that!
* **No class should derive from a concrete class** - If an object is derived from a concrete class, there is a dependency to a concrete class. Derive from an abstraction, like an interface or an abstract class.
* **No method should override an implemented method of any of its base classes** - If an implemented method is overridden, then the base class wasn’t really an abstraction to start with. Those methods implemented in the base class are meant to be shared by all subclasses.

# Law of Demeter / Principle of Least Knowledge

Reduce interactions between objects to just a few close *friends*.

When designing a system, for any object, be careful of the number of classes it interacts with and how it comes to interact with those classes.

This principle prevents designs that have many classes coupled together so that changes in one part of the system cascade to other parts. When there are a lot of dependencies between many classes, the system being built can be fragile and that will be costly to maintain and complex for others to understand.

The principle states that for any given object methods should only be invoked that belong to:

* The object itself
* Objects passed in as a parameter to the method
* Any method the object creates or instantiates
* Any components of the method

public class Car {

Component of the class. Can call its methods

private readonly Engine \_engine;

public Car(Engine engine) { \_engine = engine; }

public void Start(Key key) {

Can call a method on an object passed as a parameter

Doors doors = new Doors();

bool authorised = key.Turns();

if (authorised) {

Can call a method on a component of an object 

\_engine.Start();

updateDashboardDisplay();

doors.Lock();

}

Can call a local method within the object

Can call a method of an object created or instantiated

}

private void updateDashboardDisplay() { // update display }

}

# Patterns

## **Strategy Pattern**

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**Usage examples:** Used to provide users a way to change the behaviour of a class without extending it.

**Identification:** Strategy pattern can be recognized by a method that lets a nested object do the actual work, as well as the setter that allows replacing that object with a different one.

**Example:** <https://refactoring.guru/design-patterns/strategy/csharp/example>

## **Observer Pattern**

The Observer Patten defines a one-to-many dependency between objects so that when the state of one object changes, all its dependants are notified and updated automatically.

**Usage examples:** It provides a way to react to events happening in other objects without coupling to their classes – particularly found in the GUI components.

**Identification:** The pattern can be recognized by subscription methods that store objects in a list and by calls to the update method issued to objects in that list.

**Example:** <https://refactoring.guru/design-patterns/observer/csharp/example> or <https://codewithshadman.com/observer-pattern-csharp/>

## **Decorator Pattern**

The Decorator Patten allows the ability to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**Usage examples:** The Decorator is standard in C# code, especially in code related to streams.

**Identification:**  Decorator can be recognized by creation methods or constructor that accept objects of the same class or interface as a current class.

**Example:** <https://refactoring.guru/design-patterns/decorator/csharp/example>

## **Factory Pattern**

The Factory Pattern lets a class defer instantiation to the subclasses.

The Abstract Factory Patten provides an interface for families of related or dependant objects without specifying their concrete classes.

**Usage examples:** Both Factory and Abstract Factory patterns are very common in C# code. They are very useful when used to provide a high level of flexibility to code and as such many frameworks and libraries use them to provide a way to extend and customize their standard components.

**Identification:** Factory methods can be recognized by creation methods, which create objects from concrete classes, but return them as objects of abstract type or interface.

**Example:** <https://refactoring.guru/design-patterns/factory-method/csharp/example> and <https://refactoring.guru/design-patterns/abstract-factory/csharp/example>

## **Singleton Pattern**

Ensure a class only has one instance and provide a global point of access to it.

**Usage examples:** Can be considered an antipattern. Usage is on the decline in C# code.

**Identification:** Can be recognised by a static creation method, returning the same cached object.

**Example:** <https://refactoring.guru/design-patterns/singleton/csharp/example>

## **Command Pattern**

Encapsulates a request as an object, thereby letting you parameterise other objects with different requests, queue or log requests, and support undoable operations.

**Usage examples:** Most often used as an alternative for call-backs to parameterising UI elements with actions. It’s also used for queueing tasks, tracking operations history, etc.

**Identification:** The Command pattern is recognizable by behavioural methods in an abstract/interface type (sender) which invokes a method in an implementation of a different abstract/interface type (receiver) which has been encapsulated by the command implementation during its creation. Command classes are usually limited to specific actions.

**Example:** <https://refactoring.guru/design-patterns/command/csharp/example>

## **Adapter Pattern**

Converts the interface of a class into another interface that another client expects. Allows classes to work together that otherwise couldn’t due to incompatible interfaces.

**Usage examples:** Very often used in systems based on some legacy code. In such cases, Adapters make legacy code work with modern classes.

**Identification:** Adapter is recognizable by a constructor which takes an instance of a different abstract/interface type.

**Example:** <https://refactoring.guru/design-patterns/adapter/csharp/example>

## **Façade Pattern**

Provides a unified interface to a set of interfaces in a sub-system. Defines a higher-level interface that makes the sub-system easier to use.

**Usage examples:** It is especially handy when working with complex libraries and APIs.

**Identification:** Can be recognised in a class that has a simple interface, but delegates most of the work to other classes.

**Example:** <https://refactoring.guru/design-patterns/facade/csharp/example>

## **Template Method Pattern**

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.

**Usage examples:** Developers often use it to provide framework users with a simple means of extending standard functionality using inheritance.

**Identification:** Can be recognized by behavioural methods that already have a “default” behaviour defined by the base class.

**Example:** <https://refactoring.guru/design-patterns/template-method/csharp/example>

## **Iterator Pattern**

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Usage Examples:** Many frameworks and libraries use it to provide a standard way for traversing their collections.

**Identification:** Iterator is easy to recognize by the navigation methods (such as *next*, *previous* and others). Client code that uses iterators might not have direct access to the collection being traversed.

**Example:** <https://refactoring.guru/design-patterns/iterator/csharp/example>

## **Composite Pattern**

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

**Usage Examples:** It’s often used to represent hierarchies of user interface components or the code that works with graphs.

**Identification:** If you have an object tree, and each object of a tree is a part of the same class hierarchy, this is most likely a composite. If methods of these classes delegate the work to child objects of the tree and do it via the base class/interface of the hierarchy, this is definitely a composite.

**Example:** <https://refactoring.guru/design-patterns/composite/csharp/example>

## **State Pattern**

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

**Usage Examples:** The State pattern is commonly used in C# to convert massive switch-base state machines into the objects.

**Identification:** State pattern can be recognized by methods that change their behaviour depending on the objects’ state, controlled externally.

**Example:** <https://refactoring.guru/design-patterns/state/csharp/example>

## **Proxy Pattern**

Provide a surrogate or placeholder for another object to control access to it.

**Usage Examples:** It’s irreplaceable when you want to add some additional behaviours to an object of some existing class without changing the client code.

**Identification:** Proxies delegate all the real work to some other object. Each proxy method should, in the end, refer to a service object unless the proxy is a subclass of a service.

**Example:** <https://refactoring.guru/design-patterns/proxy/csharp/example>

## **Compound Patterns**

A Compound Pattern combines two or more patterns into a solution that solves a recurring or general problem. An example of a Compound Pattern is MVC, which combines Observer (Model), Composite (View) and Strategy (Controller) patterns.

**Bridge Pattern**

Divides business logic into separate class hierarchies that can be developed independently.

**Usage Examples:** Especially useful when dealing with cross-platform apps, supporting multiple types of database servers, or working with several API providers of a certain type.

**Identification:** Recognised by a clear distinction between some controlling entity and several different platforms it relies upon.

**Example:** <https://refactoring.guru/design-patterns/bridge/csharp/example>

**Builder Pattern**

Creational design pattern, which allows constructing complex objects step-by-step. Unlike other creational patterns, doesn’t require products to have a common interface.

**Usage Examples:** Especially useful when you need to create an object with lots of possible configuration options.

**Identification:** Can be recognised in a class which has a simple creation method and several methods to configure the resulting object.

**Example:** <https://refactoring.guru/design-patterns/builder/csharp/example>

## **Chain of Responsibility**

Allows multiple objects to handle the request without coupling sender class to the concrete classes of the receivers. The chain can be composed dynamically at runtime with any handler that follows a standard handler interface.

**Usage Examples:** Not a common pattern only relevant when code operates with chains of objects.

**Identification:** recognizable by behavioural methods of one group of objects that indirectly call the same methods in other objects, while all the objects follow the common interface.

**Example:** <https://refactoring.guru/design-patterns/chain-of-responsibility/csharp/example>

## **Flyweight**

Allows programs to support vast quantities of objects by keeping their memory consumption low.

**Usage Examples:** The Flyweight pattern has a single purpose: minimizing memory intake.

**Identification:** Recognised by a creation method that returns cached objects instead of creating new.

**Example:** <https://refactoring.guru/design-patterns/flyweight/csharp/example>

## **Mediator**

The Mediator makes it easy to modify, extend and reuse individual components because they’re no longer dependent on the dozens of other classes.

**Usage Examples:** The most popular usage of the Mediator pattern in C# code is facilitating communications between GUI components of an app.

**Example:** <https://refactoring.guru/design-patterns/mediator/csharp/example>

## **Memento**

Allows making snapshots of an object’s state and restoring it in future. The Memento doesn’t compromise the internal structure of the object it works with, as well as data kept inside the snapshots.

**Usage Examples:** The Memento’s principle can be achieved using serialization.

**Example:** <https://refactoring.guru/design-patterns/memento/csharp/example>

## **Prototype**

Creational design pattern that allows cloning objects, even complex ones, without coupling to their specific classes.

**Usage Examples:** Prototype pattern is available out-of-the-box with a ICloneable interface.

**Identification:** Easily recognisable with *clone* or *copy* methods.

**Example:** <https://refactoring.guru/design-patterns/prototype/csharp/example>

## **Visitor**

Allows adding new behaviours to existing class hierarchy without altering any existing code.

**Usage Examples:** Visitor isn’t a very common pattern because of its complexity and narrow applicability.

**Example:** <https://refactoring.guru/design-patterns/visitor/csharp/example>