Design Patterns are Not Dead

The case for caring about and learning some of them, and how they exist in Dotnet

OOP
Object Oriented Programming

Breaking down concepts and operations into objects

I regret the term ‘object’ being so prevalent, as the real point is around the messaging, how information goes from place to place and is stored.

- Alan Kay

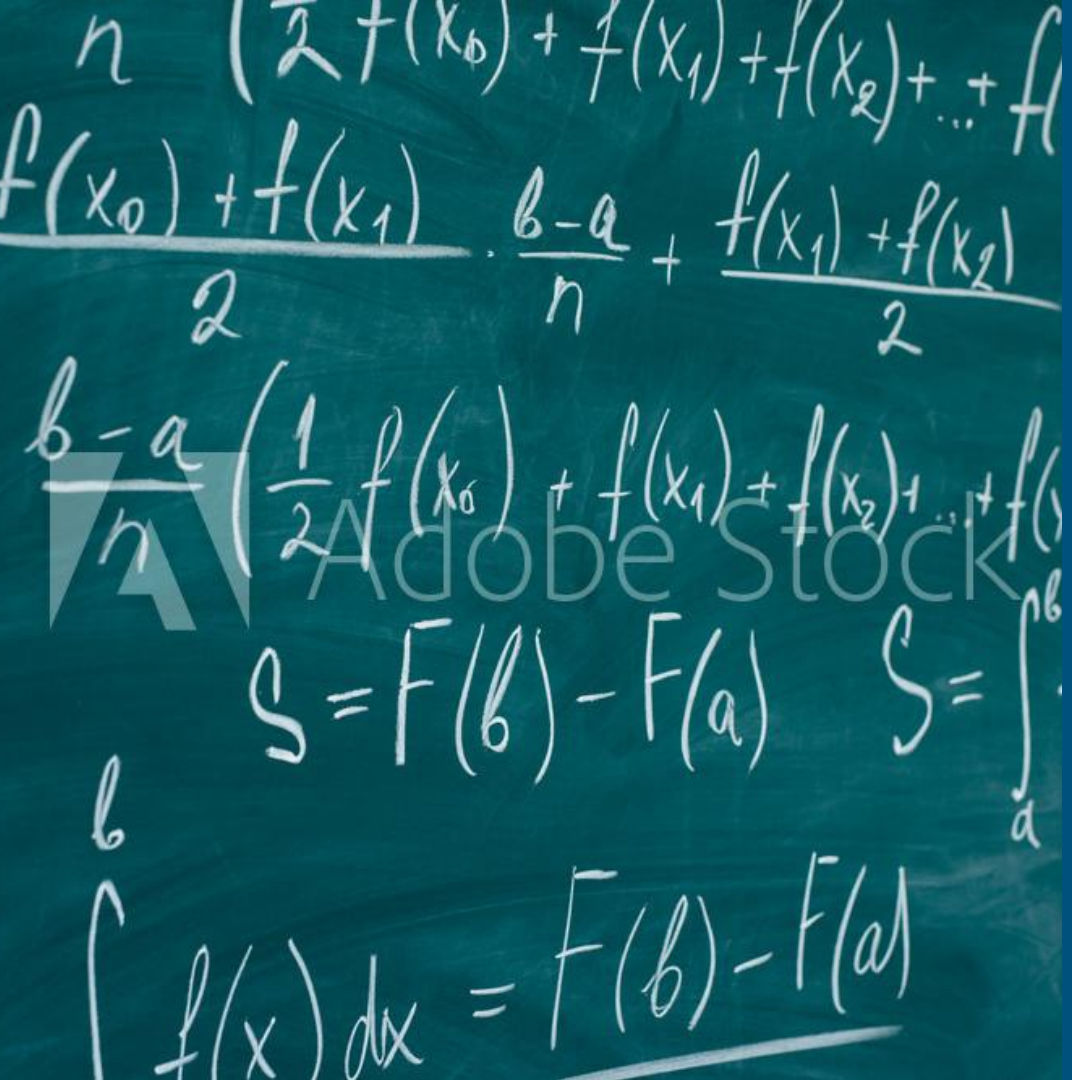
What isn’t OOP

Procedural Programming

List of Instructions

Scripting

C is a procedural programming language



Functional Programming

Actions over objects

Immutability

Different Design Patterns

Pretend OOP

Util class

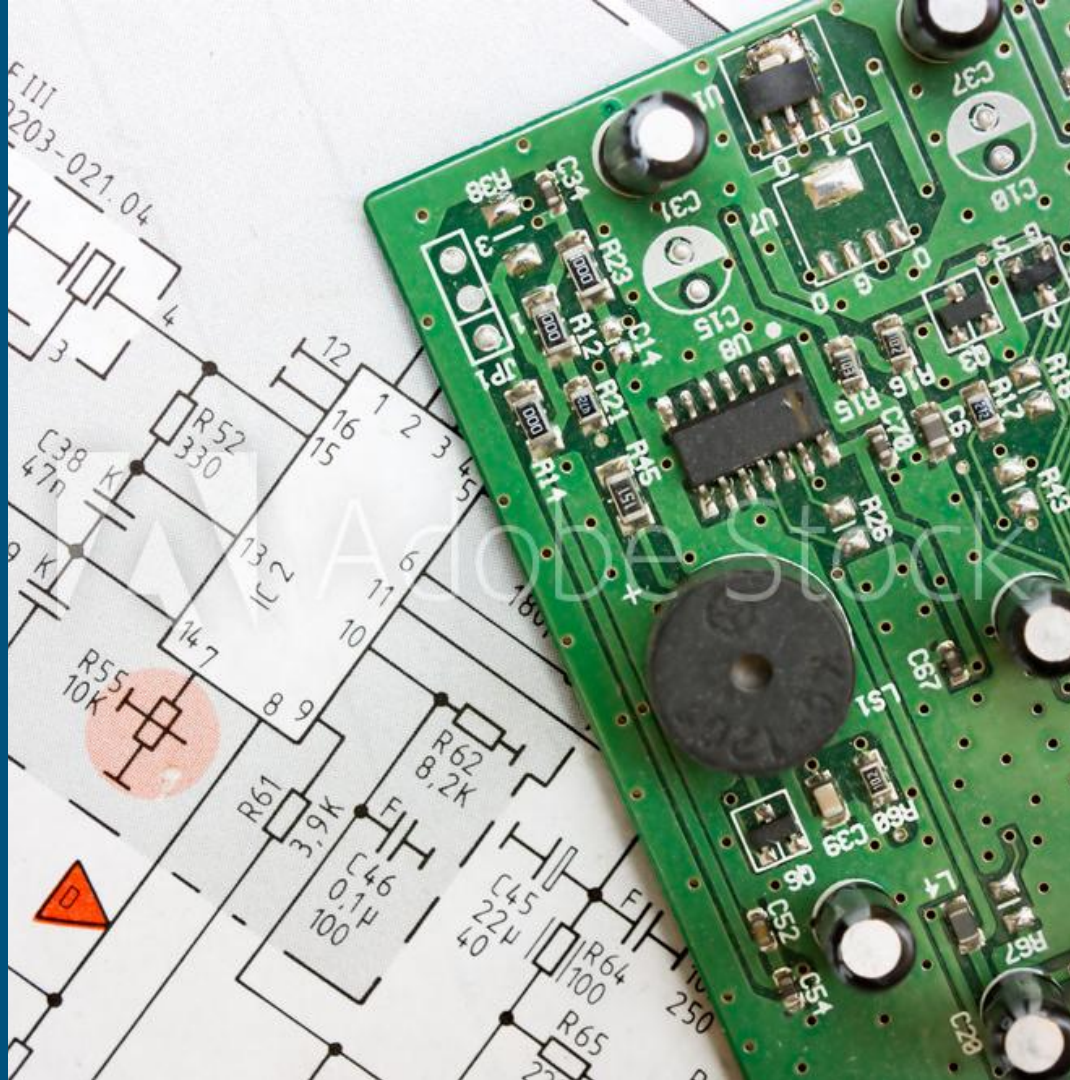
Do Everything objects

Poorly planned object organization

OOP and Design Patterns

OOP is a high-level abstraction / Design Pattern

Our design patterns are based on Object Oriented Concepts



C#

- OO at the core, with procedural and functional capabilities
- Built with the .NET Framework that uses many design patterns
- Improved and expanded on regularly, incorporating design patterns into the language

What are Design Patterns?

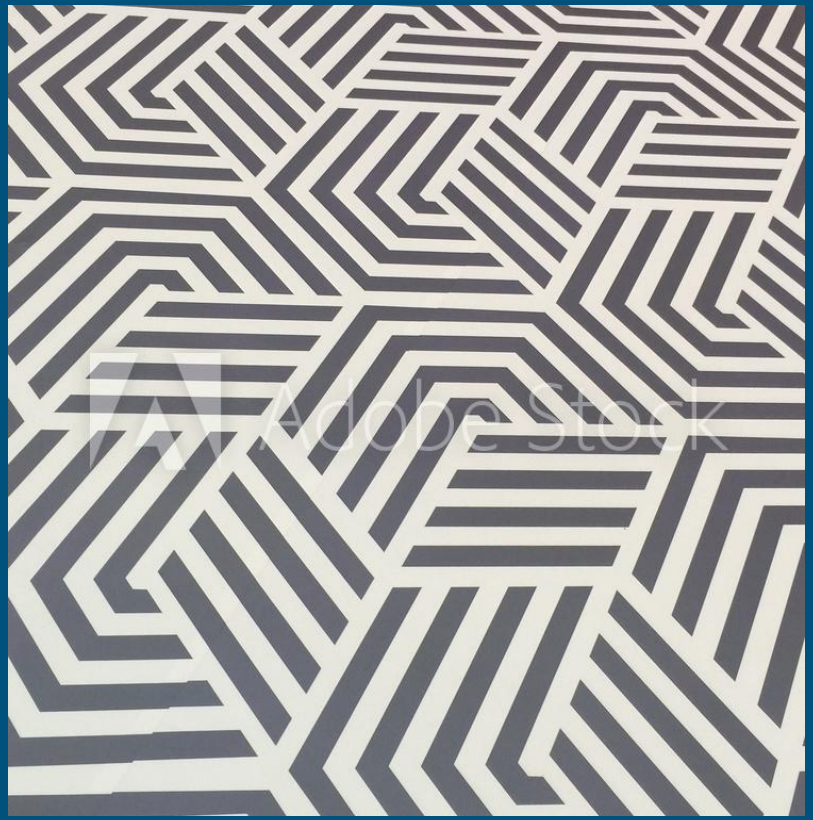
The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Book:
A Pattern Language

Towns, Buildings,
Construction

Christopher Alexander et al
1977

Design Patterns are a way to understand recurring problems and the types of solutions effective for that problem.



Design Patterns: Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
1994

Pattern Languages

How a Pattern is described and shared

- Name
- Motivation
- Applicability
- Structure Diagram
- Participating Classes
- Consequences
- Examples

Patterns are a Guide

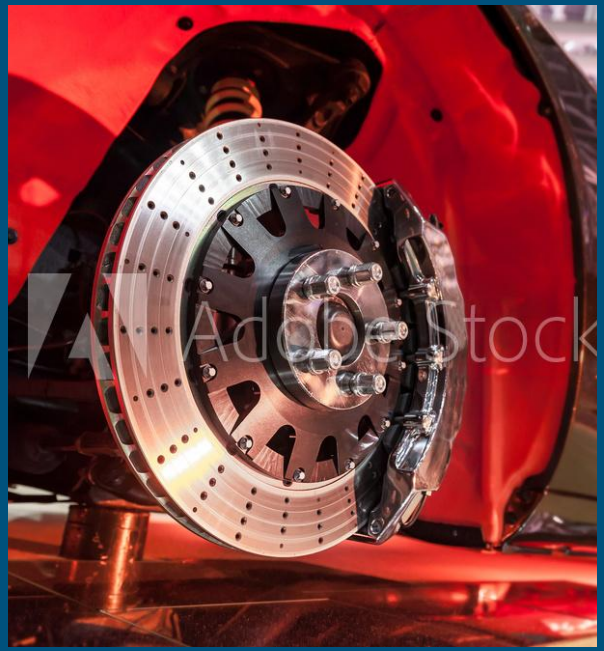
Effective use of patterns requires understanding the bigger picture goals

Why Learn Design Patterns?

Patterns and Grammar

Know why you are making a choice

Wheel Patterns



Pattern Matching

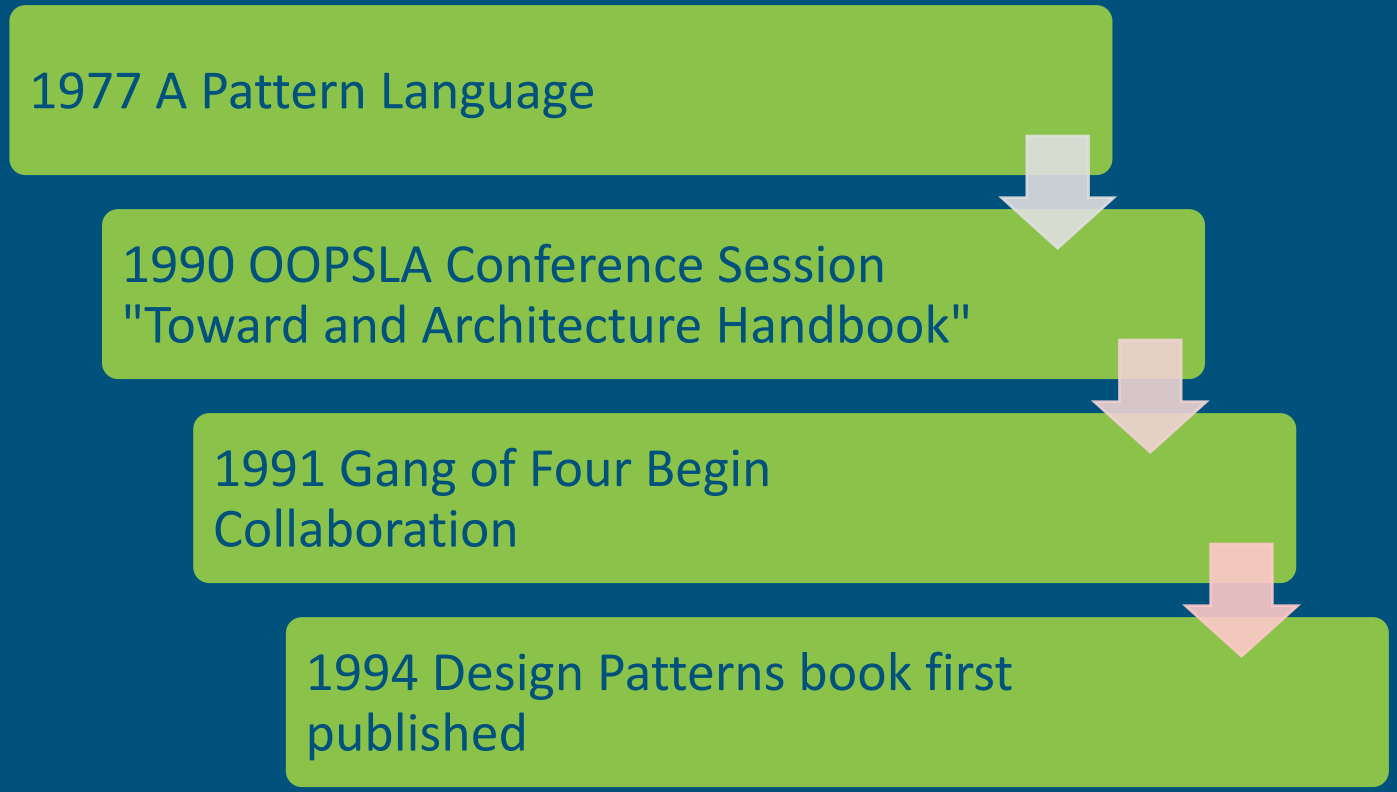
If X, then Y

Pattern Benefits

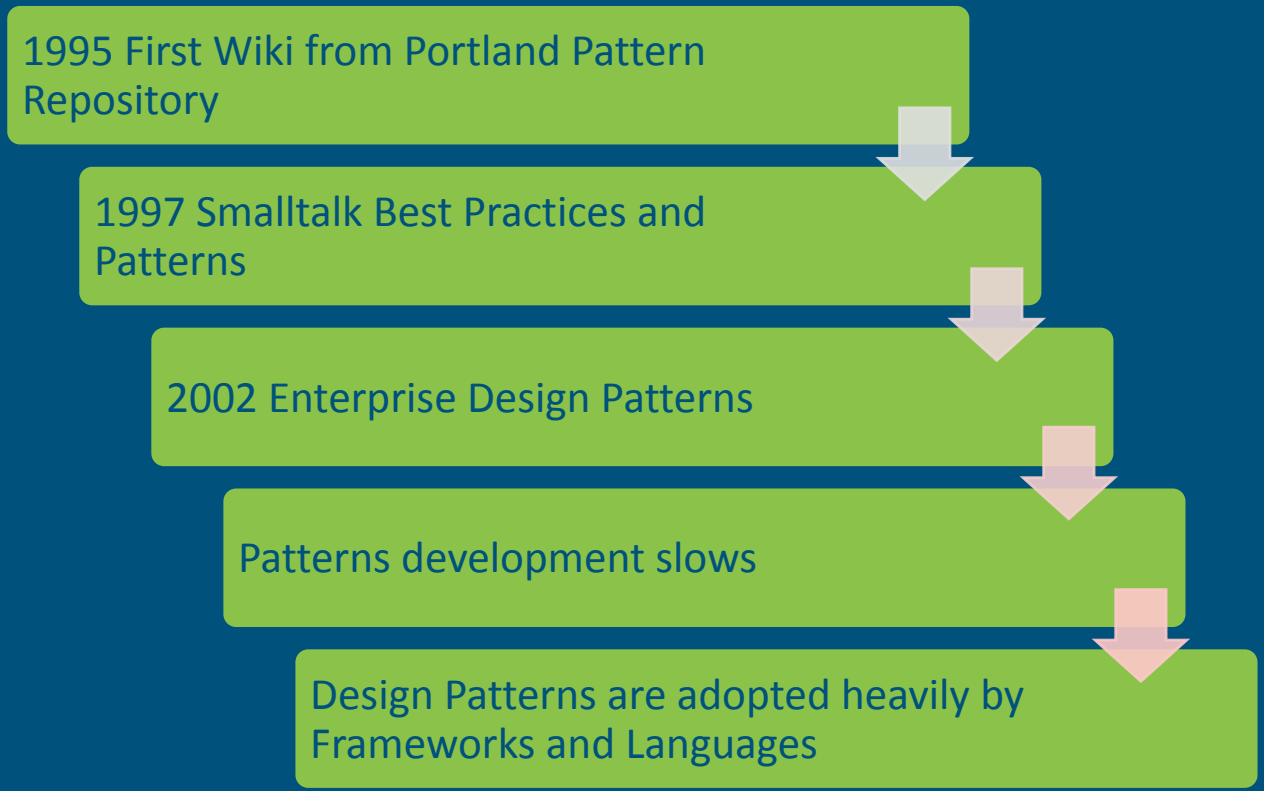
- Shared common industry knowledge
- Learn from other’s mistakes
- Improved understanding for future maintenance
- Don’t reinvent the wheel

A Short History of Design Patterns

Timeline


1977 A Pattern Language
1990 OOPSLA Conference Session "Toward and Architecture Handbook"
1991 Gang of Four Begin Collaboration
1994 Design Patterns book first published

Timeline


1995 First Wiki from Portland Pattern Repository
1997 Smalltalk Best Practices and Patterns
2002 Enterprise Design Patterns
Patterns development slows
Design Patterns are adopted heavily by Frameworks and Languages

Organizing Design Patterns

Creational Purpose

How and when objects are created

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Structural Purpose

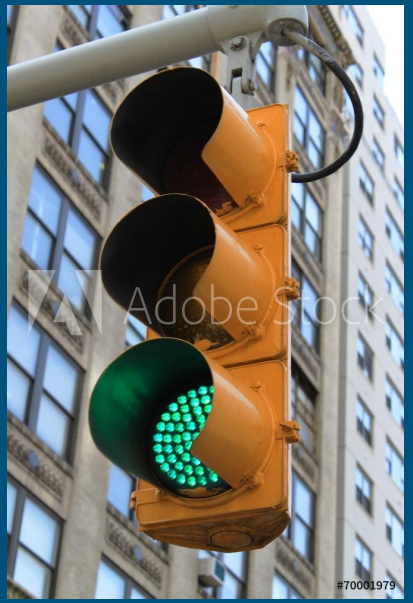
How classes and objects relate

- Adapter
- Bridge
- Composite
- Flyweight
- Decorator
- Façade
- Proxy

Behavioral Purpose

Responsibility and Action

- Interpreter
- Template Method
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor



Concurrency Purpose

Handling Concurrent Actions

Scope

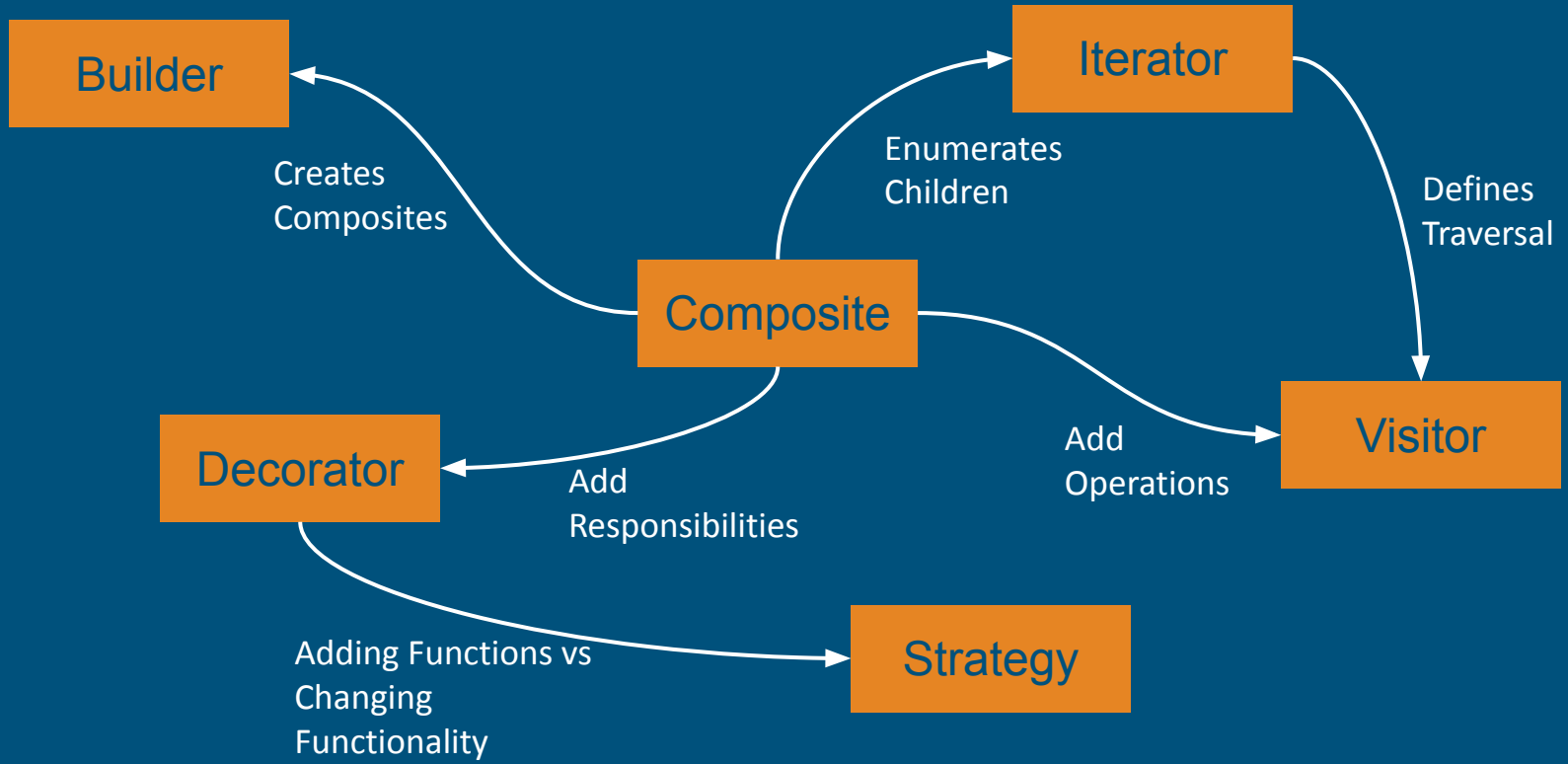
Class

- Operates at the type and class level relationships between classes and sub-classes
- Factory, Adapter, Interpreter, Template Method

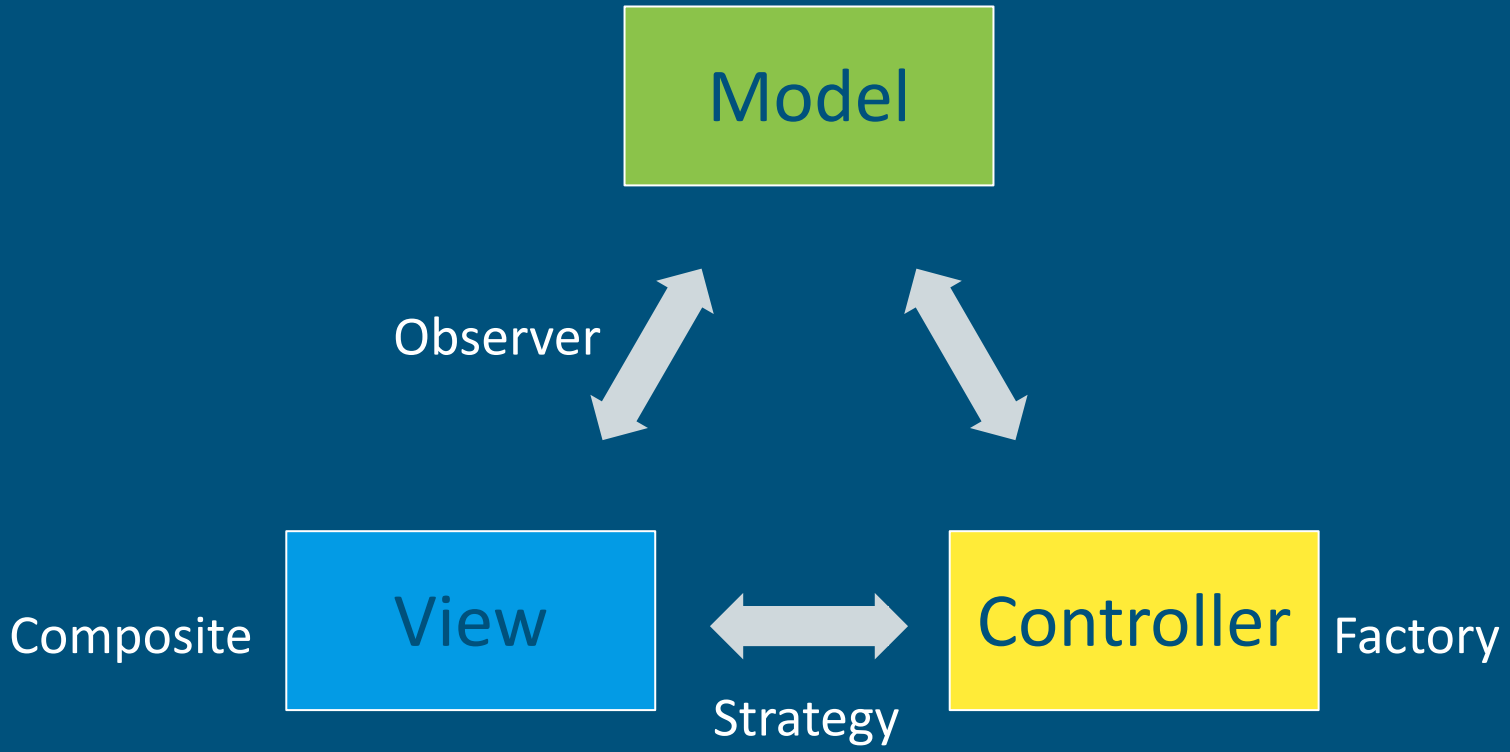
Object

- Operates on or through instantiated objects
- Abstract Factory, Adapter again, All the other patterns

Patterns by Relationship Example



Patterns in MVC



How to Use Design Patterns

Nothing New

- Patterns are patterns because these problems have happened before

Learn Patterns

- Learn design patterns to profit from the mistakes of others

Steps to use a design pattern

1. Read the overview
2. Understand the Parts
3. Read code that uses the pattern
4. Try it on for size
    - Do it well

Practice Learning Patterns

- You don’t get better at something without practice

Design Patterns Drawbacks

Pattern Cost

- Implementing a design pattern is not free

Technical Debt

- Understand the trade-off
- Don’t save a little now to spend a lot later

Understand the Consequences

- Don’t create unnecessary work
- Don’t add flexibility where you don’t need it
- Learn the common issues with a specific pattern
- Don’t re-create a pattern built in to your language or framework

Creational Pattern Overview

Factory Method

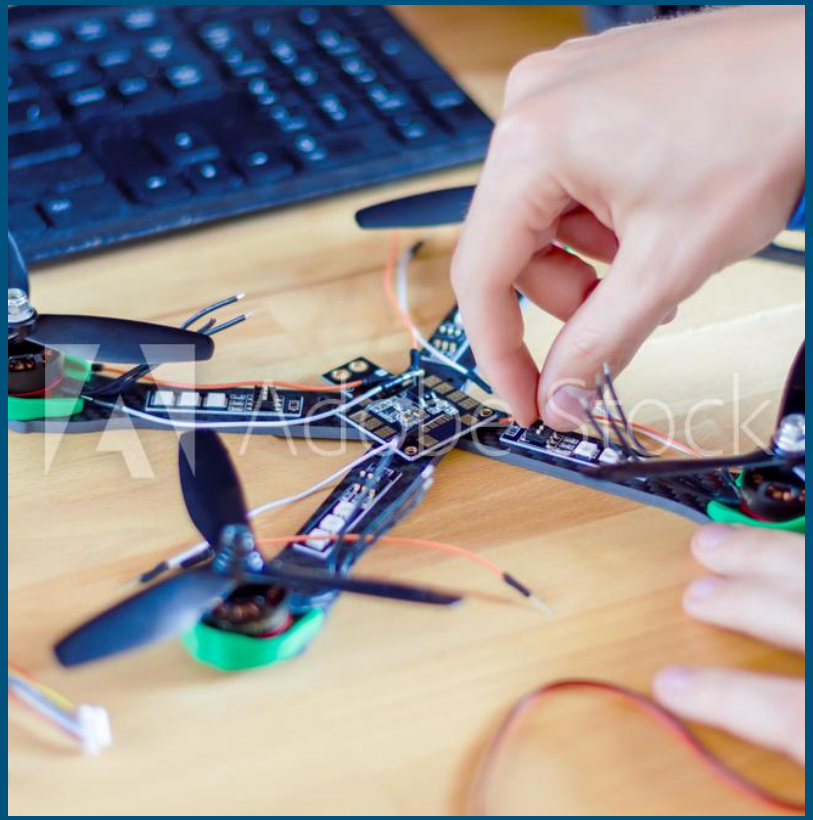
- Separate using an object from creating it
- Use dynamic context to decide what to build
- Ex: A Logger Factory

Abstract Factory

- Factory Interface for getting the right type of factory
- Create a Family of related dependent objects
- IHTTPClientFactory

Builder

- An object with the purpose of constructing another, more complex object
- StringBuilder
- ApplicationBuilder
- EF Core ModelBuilder



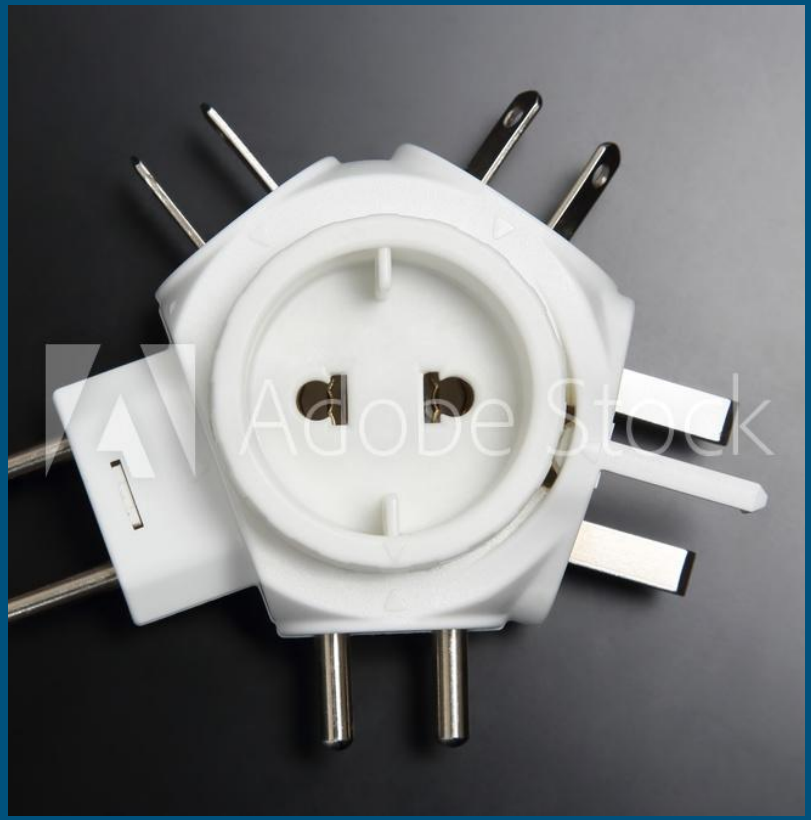
Prototype

- Create an object by cloning an existing concrete object
- System.Net.Http.Headers

Singleton

- Can be dangerous
- Allow one an only one of an object that is shared application wide
- Dependency Injection Service Provider / Container

Structural Pattern Overview



Adapter

- Convert from one interface to another
- Mapping from a data model to a view model

Bridge

- Independently change an abstraction and its concrete implementation

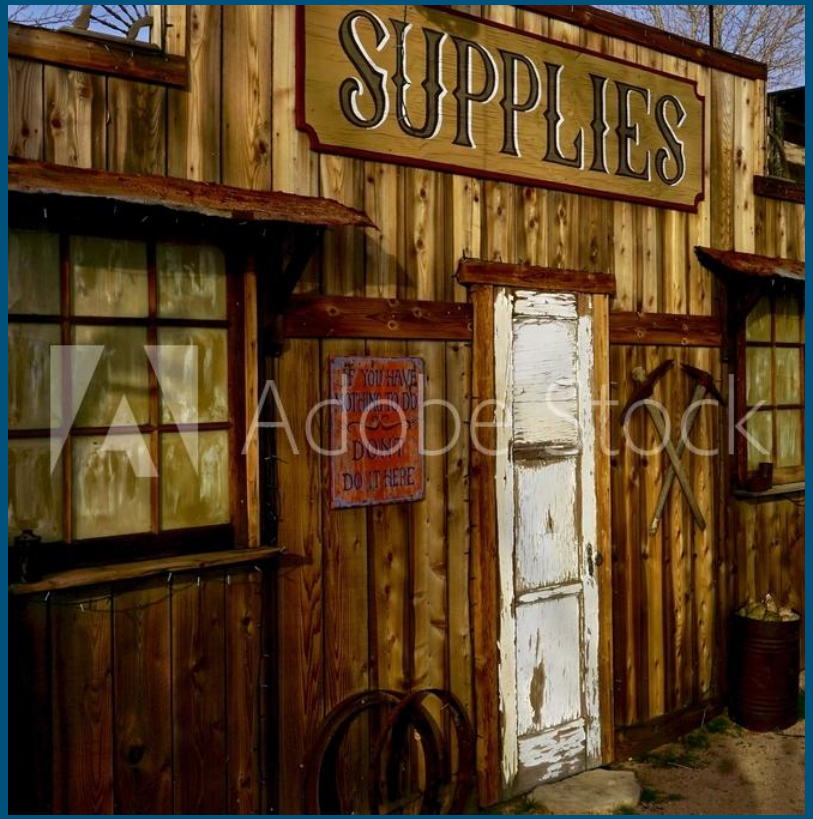
Composite

- Treat an object the same as a group of that object
- MVC View hierarchies

Decorator

- Add functionality to an existing object by wrapping it
- Doesn’t require sub classing
- IO Streams in .Net

Façade



- Create an interface for a specific purpose
- Simplify a single interface, or group a set of interfaces together
- .Net System.Environment

Flyweight

- Create representational versions of an object
- Save memory by keeping fewer physical copies
- C# String Interning

Proxy

- Gateway object to control access to another object or resource
- Concurrency Checks
- Role based access
- EF Core data context

Behavioral Pattern Overview

Interpreter

- A context specific language
- Compose ‘sentences’ that can be interpreted
- LINQ Expression Trees

Template Method

- Create places where subclasses can plug in part of an algorithm
- IComparable Sorting

Chain of Responsibility

- Pass an object down a chain until it is handled appropriately
- Exception Handling
- ASP Net Core Middleware

Command

- Encapsulate an action as its own object
- Request fulfillment separate from the request itself
- EventArgs

Iterator

- Sequentially move through a set of objects
- Type of the object is irrelevant
- Foreach loop in C#

Visitor

- Perform the same operation on each item in a set of similar items
- Content of the foreach loop
- ExpressionVisitor in LINQ

Mediator

- An object to encapsulate how two objects interact with one another
- The mediated objects don’t need to have knowledge of each other

Memento

- Save State for an object
- Provide ‘Undo’ functionality on an object
- C# Serialization (Sometimes)

Observer

- Wide array of usage
- One to many relationship
- Subject doesn’t have to know about observers
- Events and Handlers
- INotifyPropertyChanged

State

- Separate States of an object from the object itself
- Reduce modifications to the original object
- Add states without affecting existing states

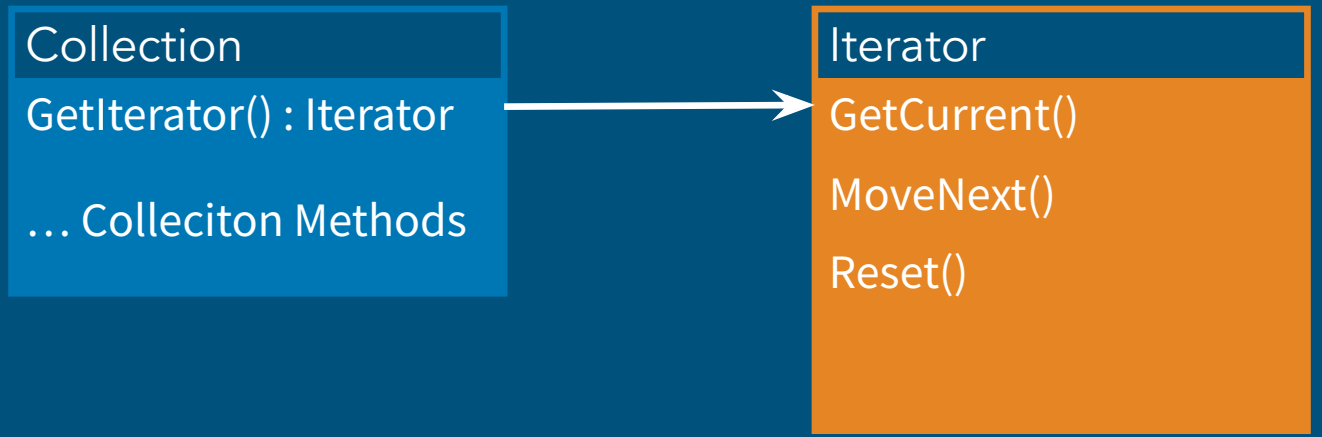
Strategy

- Swappable algorithms or actions
- Decouple an action from the object that calls it
- .Net Cryptographic Libraries

Define Iterator

1	2	3
Name	**Description**	**Goal**
Iterator Cursor Enumerator	Access items in a set or collection one by one	Separate Moving through the collection from the collection itself

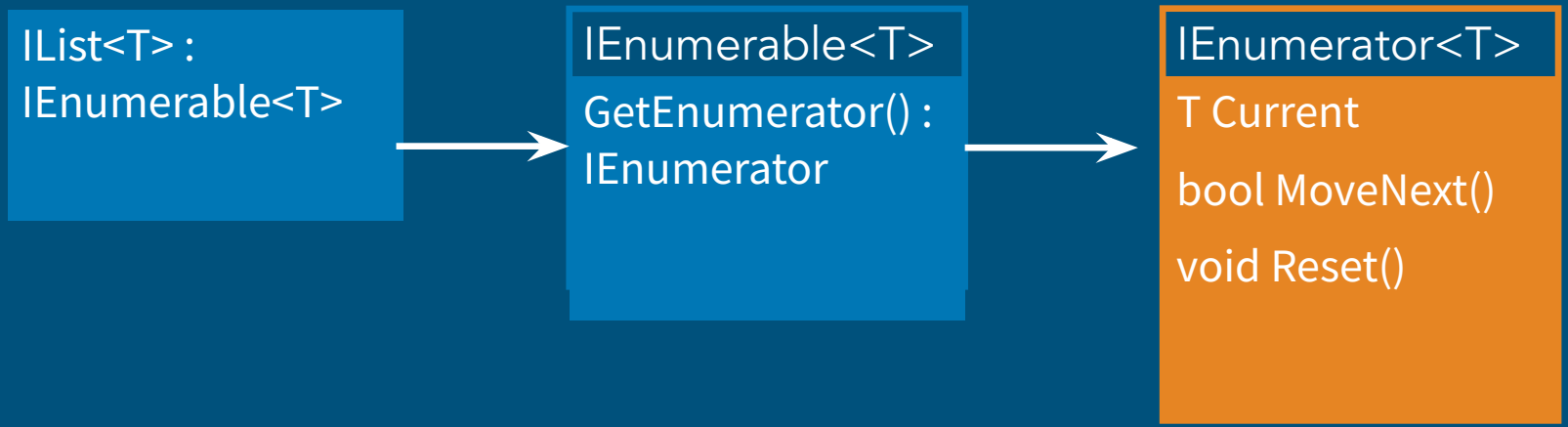
Iterator


Collection
GetIterator() : Iterator
… Colleciton Methods
Iterator
GetCurrent()
MoveNext()
Reset()

Benefits

- Clean Collection Interface
- Access items without sharing how they are stored
- Access items in different structures the same way
- Can use more than one iterator at a time

C# Enumerator



```
var enumerator = SomeList.GetEnumerator();

while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    //item specific code
}
```

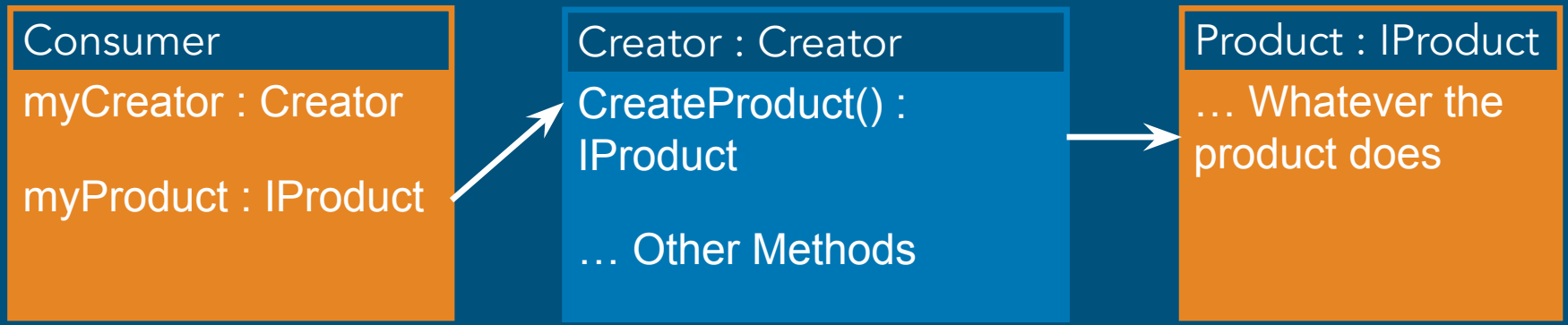
Implementation Choices

- Internal Or External Iterator?
- Will your Iterator tolerate collection changes?
- How much functionality will the Iterator have?

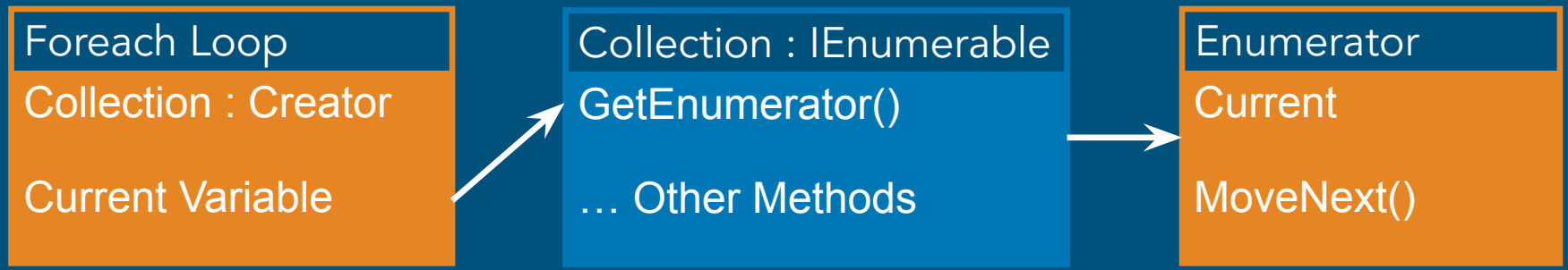
Factory Method

1	2	3
Name	**Description**	**Goal**
Factory Method Virtual Constructor	Construct objects dynamically from another type, returning an object of the expected interface	Encapsulate the returned type and separate the construction details from the dependent type.

Factory


Consumer
myCreator : Creator
myProduct : IProduct
Creator : Creator
CreateProduct() : IProduct
… Other Methods
Product : IProduct
… Whatever the product does

C# Enumerable as Factory



Benefits

- Decouple the consumer from the product
- Use details from the current state
  or user input to create the product
- Subclassed objects can be created
- Enable Dependency Injection

Implementation Choices

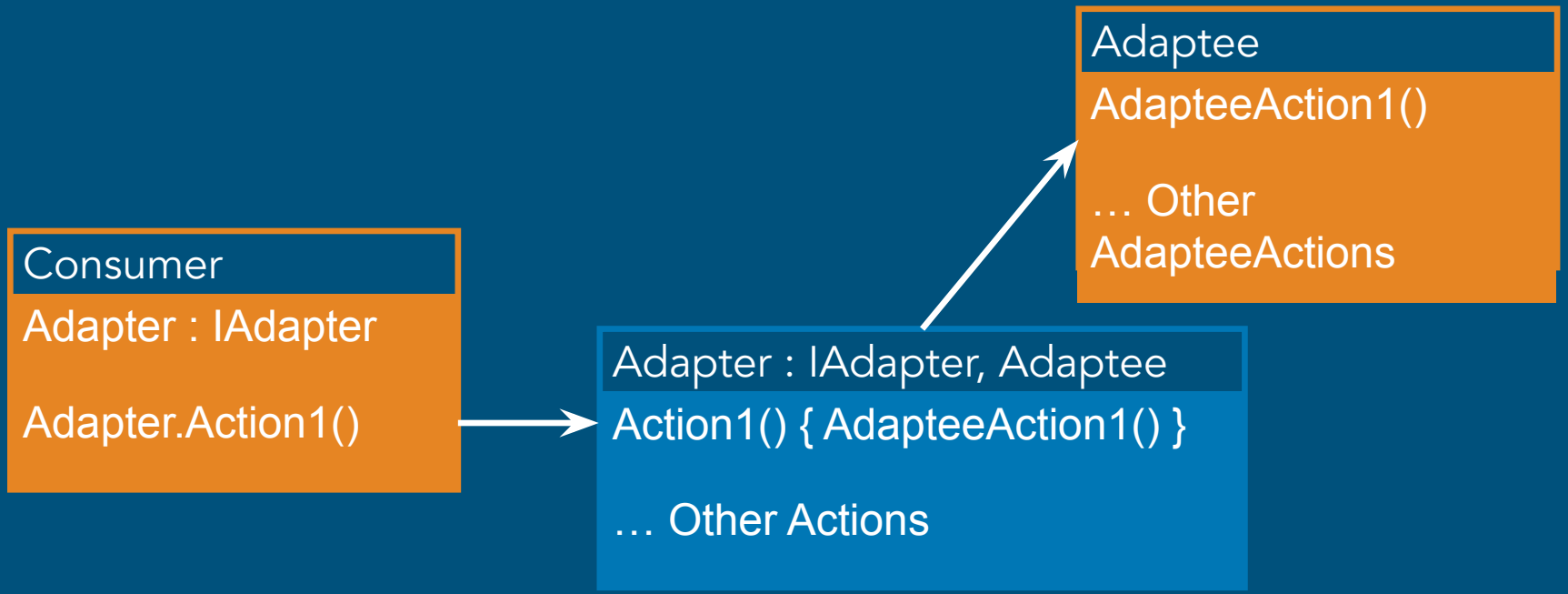
- Concrete vs Abstract Creator Class
- Parameterized factory methods.
- Generic types for the Factory Method

Adapter

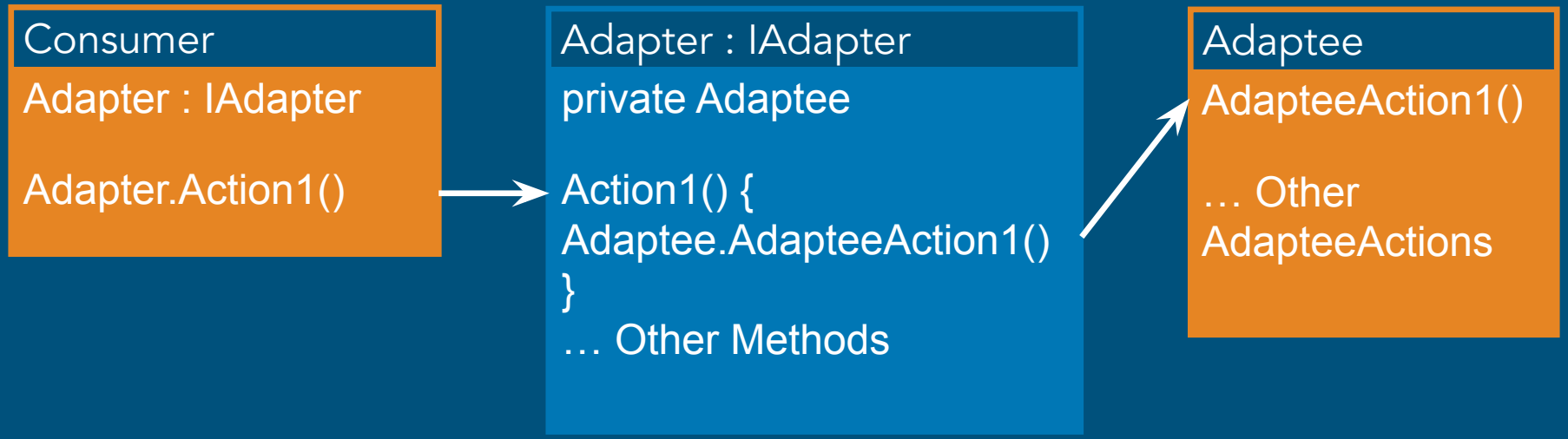
Adapter

1	2	3
Name	**Description**	**Goal**
Adapter Wrapper	Convert one interface to another by wrapping the original interface in the expected interface	Allow classes expecting one interface to use a class with a different interface

Class Adapter



Object Adapter



Benefits

- Use an interface in a new way without changing it
- Isolate one interfaces change from another interface

Adapter in C# and Dotnet Core

- System level calls, like file system access
- Generated Service References
- EF Core database specific code

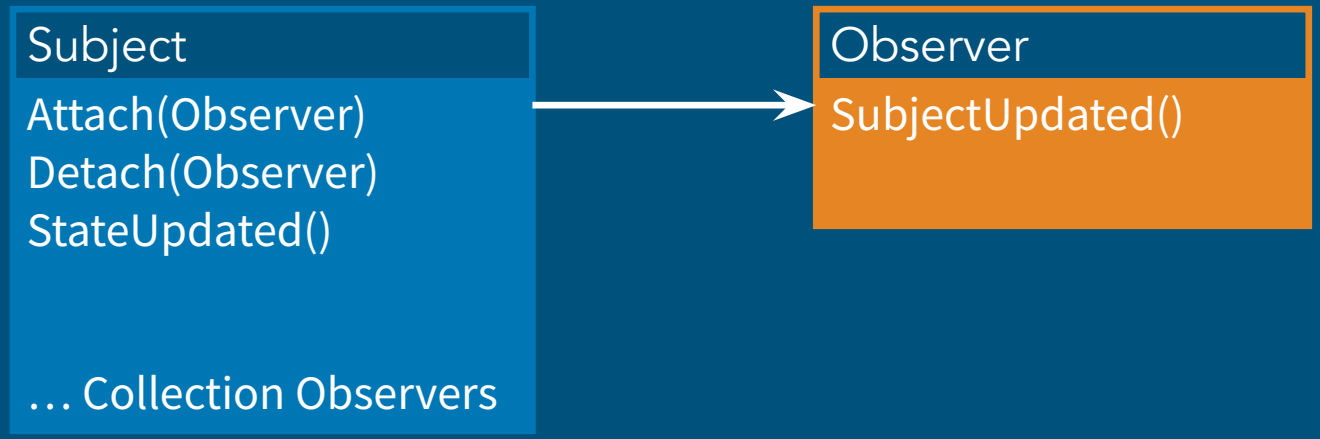
Implementation Choices

- Class level or object level?
- How active is the adapter?
- Multiple Clients?

Observer

1	2	3
Name	**Description**	**Goal**
Observer Publish / Subscribe (PubSub) Dependents	Any number of observer objects registered for changes to a subject object's state	Isolate objects that rely on the state of another. Promote consistency without coupling.

Observer


Subject
Attach(Observer)
Detach(Observer)
StateUpdated()
… Collection Observers
Observer
SubjectUpdated()

```
public class Subject {
    private List<IObserver> observers;
    public void Attach(IObserver obs){
        observers.Add(obs);
    }
    public void Detach(IObserver obs){
        observers.Remove(obs);
    }
    private void SomethingToWatchFor(){
        observers.ForEach(o => o.UpdateAction())
    }
}
```

Observable Patterns with C#

- Events and Delegates
- INotifyPropertyChanged and Observable Collection
- Reactive dotnet (Rx.net)
- Durable Queues (Rabbit, nServiceBus, MSMQ)
- Xamarin(RIP) Messaging Center

Benefits

- Minimal interface between objects, allowing objects to change independently from one another (Decoupling)
- Broadcasting between objects in your application
- Crossing Application Layers

Drawbacks

- Crossing application layers
- Unexpected consequences
- C# memory leaks (dangling references)

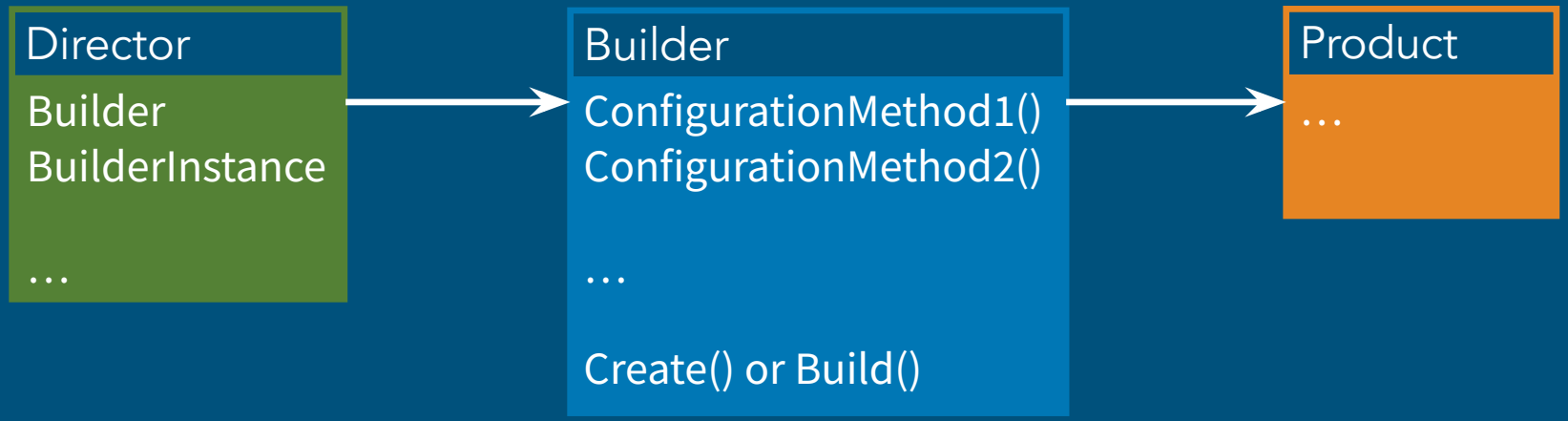
Implementation Choices

- Push vs Pull
- Guaranteed notification, and subject consistency
- Use Inheritance - or not
- Change Manager (Combine Observable with Mediator and Singleton)

Builder

1	2	3
Name	**Description**	**Goal**
Builder	An object type taking the responsibility for the construction of another type of object	Isolate creation of the object from the object itself The same process can build different results

Builder


Director
Builder
BuilderInstance
…
Builder
ConfigurationMethod1()
ConfigurationMethod2()
…
Create() or Build()
Product
…

Builder with C#

- StringBuilder
- UriBuilder
- AspNetCore ApplicationBuilder
- Fluent Builders in UI Frameworks

Benefits

- The builder encapsulates how the product is represented
- Re-use of the construction process
- Step by Step control of the object construction

Implementation Choices

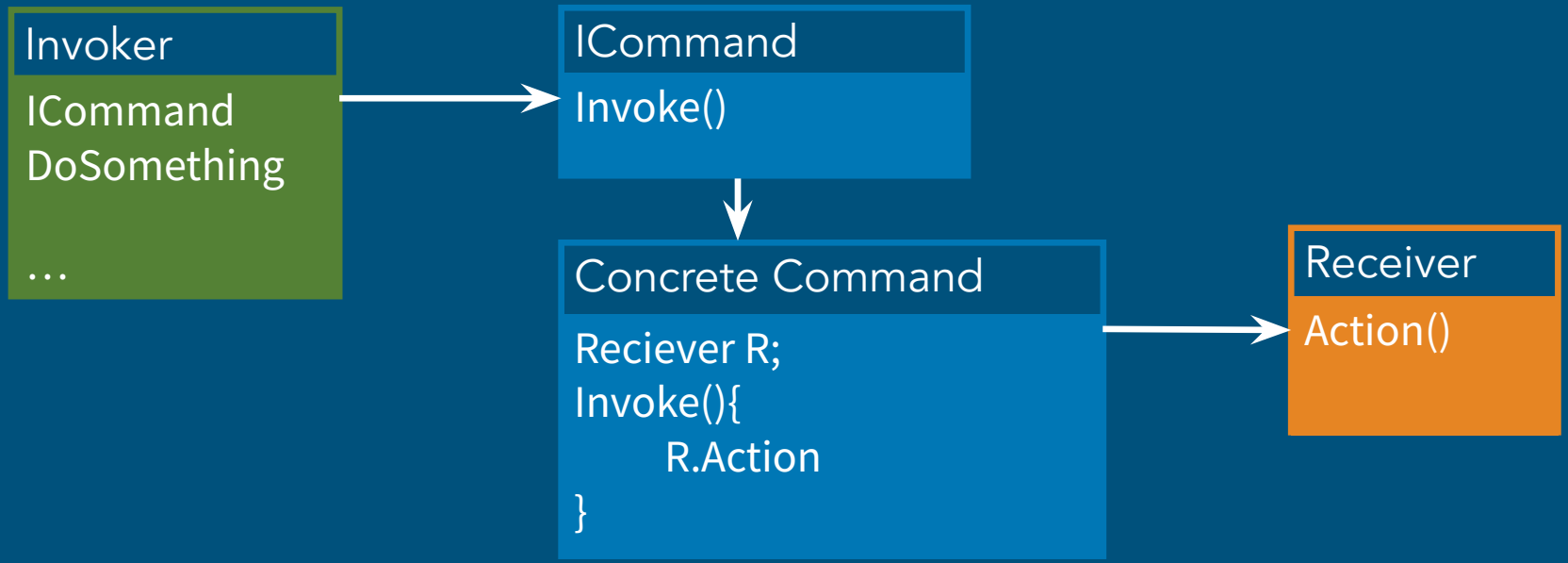
- The builder’s interface
- Fluent Builder

Command

1	2	3
Name	**Description**	**Goal**
Command Action Transaction	Represent actions, or commands, as objects in their own right	Command objects simplify request queueing or logging and separate data concerns from actions

The Command is an interface for executing a single operation.

Command


Invoker
ICommand
DoSomething
…
ICommand
Invoke()
Concrete Command
Reciever R;
Invoke(){
R.Action
}
Receiver
Action()

Command with C#

- ICommand
- CQRS
- EventArgs, in some ways

Benefits

- The invoker of an action is decoupled from the handler
- Testability
- Commands as first-class objects can be extended, serialized, etc

Drawbacks

- Can add significant complexity
- Not as common as some other patterns
- “Everything is a Command”

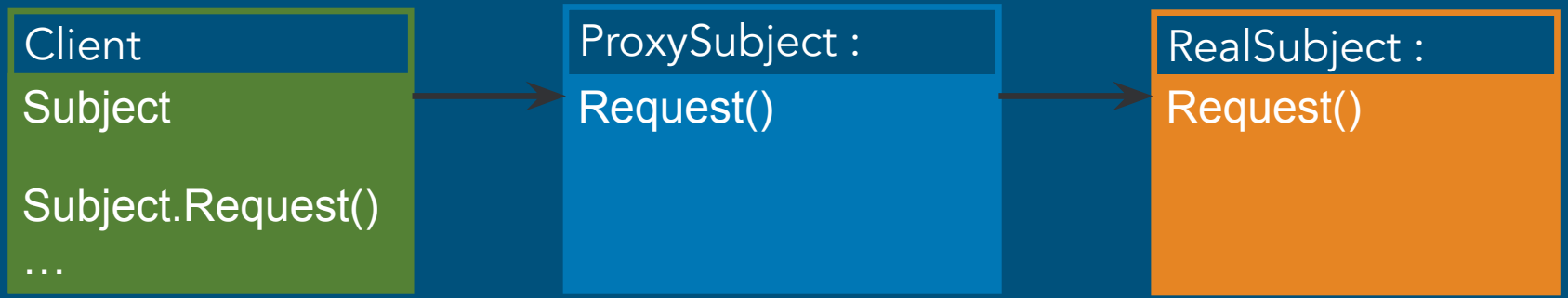
Implementation Choices

- Command Complexity
- Queue Actions
- Undo
- Transactions
- Composite Commands

Proxy

1	2	3
Name	**Description**	**Goal**
Proxy Surrogate	An object that stands in place of another object, altering access to it	Without requiring a change to an interface, provide a way to protect or hide a resource

Proxy


Client
Subject
Subject.Request()
…
ProxySubject :
Request()
RealSubject :
Request()

Ways to use Proxy

- Remote Resources
- Resource Virtualization
- Resource Caching
- Resource Protection

Proxy with C#

- EF Core Lazy Loading
- Image Libraries
- Virtual / Infinite Scrolling
- Role-based access to a resource

Benefits

- Code requesting a resource doesn’t need to change
- Long distance resources can look local
- Expensive resources can be loaded as-needed
- Limited resource access can be accessed simply

Drawbacks

- Hiding functionality from the caller
- Extra layer of indirection

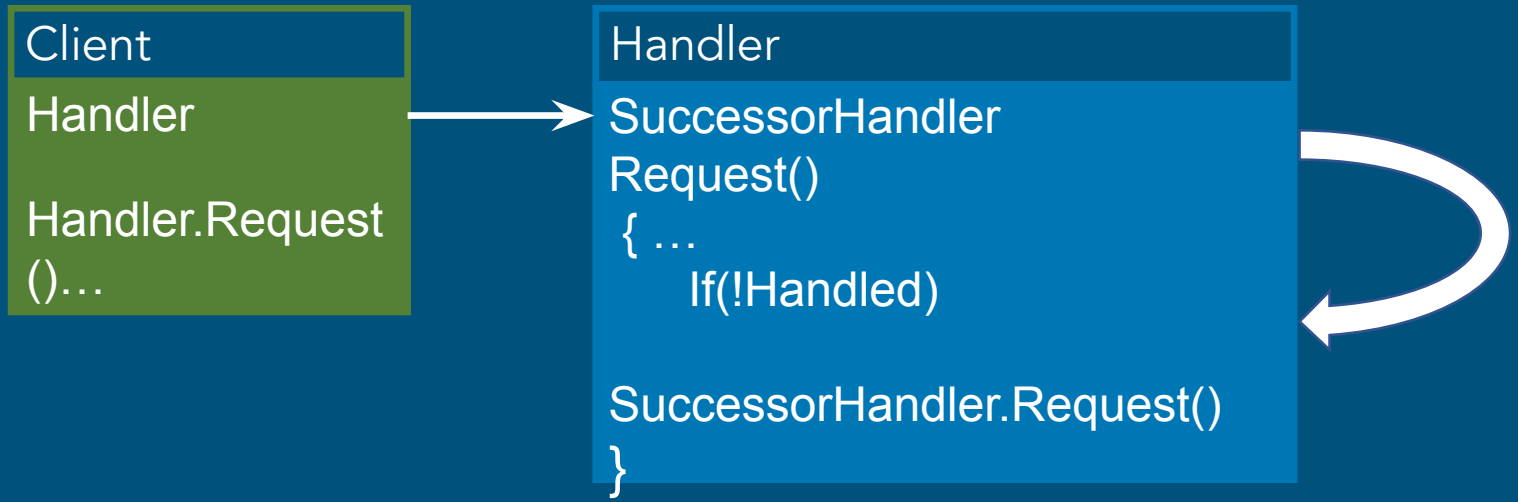
Implementation Choices

- Type of Proxy
- Type matched to the resource
- Combine Proxy with other patterns

Chain of Responsibility

1	2	3
Name	**Description**	**Goal**
Chain of Responsibility Pipeline	Chain handlers together and pass a request along until it is handled	Maximally decouple the handling of request from the requestor, allow runtime changes to request handling

Chain of Responsibility



Chain of Responsibility with C#

- Aspnet middleware
- MVC Action Filters
- Exception Handling
- RoutedEvent Handling in WPF
- Logging Systems

Benefits

- Code triggering an action or event doesn’t need to change with the way the event will be handled
- Dynamic addition of new kinds of handlers

Drawbacks

- No real guarantee of a handled event
- Can be difficult to debug some implementations

Implementation Choices

- How the handler knows about its successor
  Shared Context
  Call Stack
  UI Hierarchy
  Simple Reference

- The way in which requests are passed