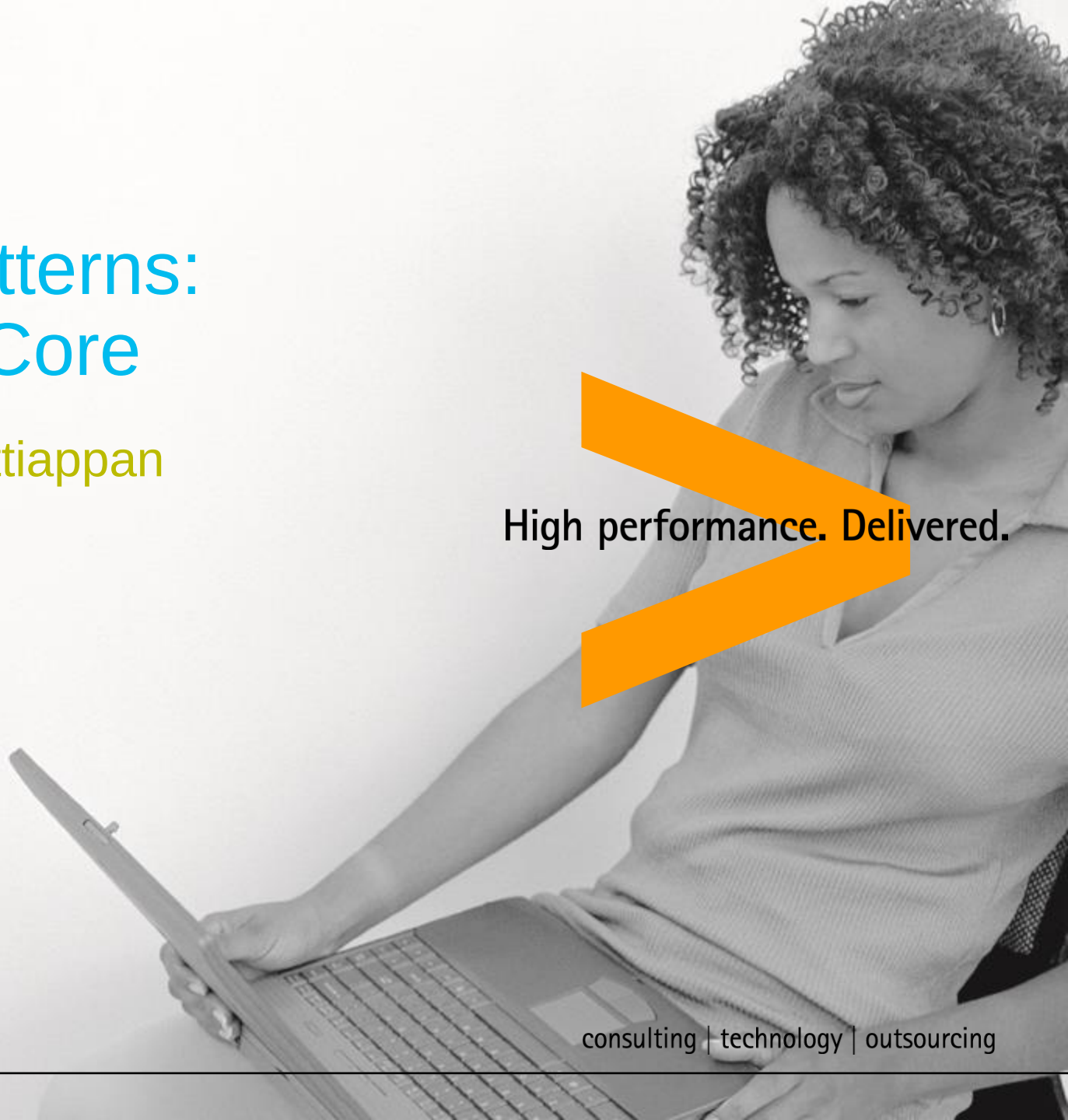
Design Patterns: ASP.NET Core

Parameswari Ettiappan



Design Patterns

Goals

- Introduction to Patterns
- MVC,MVP and MVVM
- Language and DSL Pattern
- Observer Pattern
- Iterator Pattern
- Singleton Pattern
- Patterns Relating to Factories
- Strategy Pattern
- Command Pattern

Goals

- Proxy Pattern
- Template Method Pattern
- Decorator and Adapter Pattern
- Observer Pattern
- Visitor Pattern
- State Pattern
- ASP.NET Core Architecture
- ASP.NET Core Application Development
- ASP.NET Core 2.0 Tag Helpers

Goals

- Structuring ASP.NET Application
- Dependency Injection, Configuring and Entity Framework
- TDD
- Custom Tag Helpers
- Navigation
- State Management
- Creating Restful Services using WEBAPI
- Industry best practices for .NET especially CORE based applications
- Efficient way of building project architecture – Do’s & Don’ts
- ML applications/use cases (if time permits)

Introduction to Patterns

The Pattern for successful applications

- John Lennon once wrote, “There are no problems, only solutions.”
- Now, Mr. Lennon did much in the way of ASP.NET programming;
- Still what he said is extremely relevant in the realm of software development and probably humanity.
- The job as software developers involves solving problems — problems that other developers have had to solve countless times.

Introduction to Patterns

- Throughout the lifetime of object-oriented programming, a number of patterns, principles, and best practices have been discovered, named, and catalogued.
- With knowledge of these patterns and a common solution vocabulary, we can begin to break down complex problems, encapsulate what varies, and develop applications in a uniformed way with tried and trusted solutions.

Design Patterns Explained

- Design patterns are high-level abstract solution templates.
- It is blueprint for solutions rather than the solutions themselves.
- It is difficult to find a framework that one can simply apply to your application; instead, we will typically arrive at design patterns through refactoring your code and generalizing your problem.
- Design patterns aren't just applicable to software development; design patterns can be found in all areas of life from engineering to architecture.
- In fact, it was the architect Christopher Alexander who introduced the idea of patterns in 1970 to build a common vocabulary for design discussion.

Design Patterns Explained

- He wrote: *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.*
- Alexander's comments are just as applicable to software design as they are to buildings and town planning.

DESIGN PATTERNS

Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

Origins

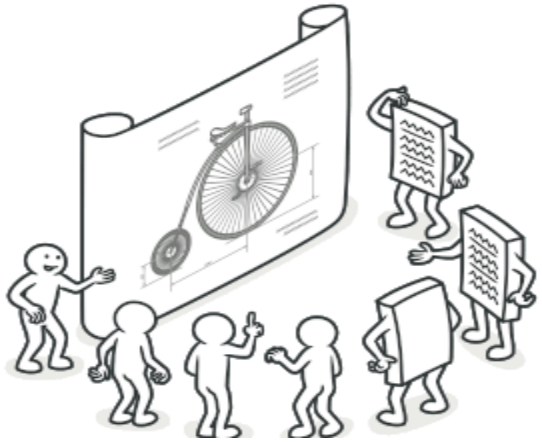
- The origins of the design patterns that are prevalent in software architecture today were born from the experiences and knowledge of programmers over many years of using object-oriented programming languages.
- A set of the most common patterns were catalogued in a book entitled *Design Patterns: Elements of Reusable Object-Oriented Software*, more affectionately known as the *Design Patterns Bible*.
- This book was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, better known as the Gang of Four

Origins

- They collected 23 design patterns and organized them into 3 groups:
- **Creational Patterns:** These deal with object construction and referencing.
- **Structural Patterns:** These deal with the relationships between objects and how they interact with each other to form larger complex objects.
- **Behavioral Patterns:** These deal with the communication between objects, especially in terms of responsibility and algorithms.

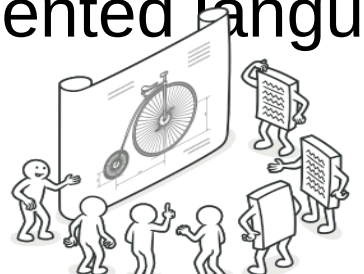
Necessity

- Patterns are essential to software design and development.
- They enable the expression of intent through a shared vocabulary when problem solving at the design stage as well as within the source code.
- Patterns promote the use of good object-oriented software design, as they are built around solid object-oriented design principles.



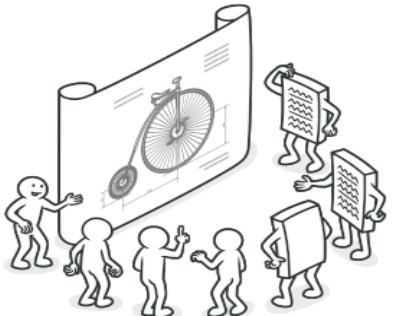
Necessity

- Patterns are an effective way to describe solutions to complex problems.
- With solid knowledge of design patterns, you can communicate quickly and easily with other members of a team without having to be concerned with the low-level implementation details.
- Patterns are language agnostic; therefore, they are transferable over other object-oriented languages.
- The knowledge one gain through learning patterns will serve him in any first-class object-oriented language he decides to program in.



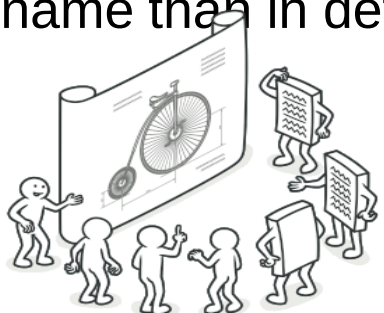
Usefulness

- The useful and ultimate value of design patterns lies in the fact that they are tried and tested solutions, which gives confidence in their effectiveness.
- For experienced developer and have been programming in .NET or another object-oriented language for a number of years, he might find that he is already using some of the design patterns mentioned in the Gang of Four book.
- By being able to identify the patterns what he is using, he must can communicate far more effectively with other developers who, with an understanding of the patterns, will understand the structure of the solution.



Usefulness

- Design patterns are all about the reuse of solutions.
- All problems are not equal, of course, if you can break down a problem and find the similarities with problems that have been solved before, one can then apply those solutions.
- After decades of object-oriented programming, most of the problems we will encounter have been solved countless times before, and there will be a pattern available to assist in solution implementation.
- Even if we believe the problem to be unique, by breaking it down to its root elements, one should be able to generalize it enough to find an appropriate solution.
- The name of the design pattern is useful because it reflects its behavior and purpose and provides a common vocabulary in solution brainstorming.
- It is far easier to talk in terms of a pattern name than in detail about how an implementation of it would work.



What they are not

- Design patterns are no silver bullet.
- We have to fully understand the problem, generalize it, and then apply a pattern applicable to it.
- However, not all problems require a design pattern.
- It’s true that design patterns can help make complex problems simple, but they can also make simple problems complex.
- After learning patterns, many developers fall into the trap of trying to apply patterns to everything they do, thus achieving quite the opposite of what patterns are all about — making things simple.
- The better way to apply patterns is by identifying the fundamental problem that we are trying to solve and looking for a solution that fits it.

What they are not

- This training will help with the identification of when and how to use patterns and goes on to cover the implementation from an ASP.NET point of view.
- We don’t always have to use design patterns.
- If we have arrived at a solution to a problem that is simple but not simplistic and is clear and maintainable.
- Don’t force it to fit the pattern.
- Otherwise, we will overcomplicate your design.

Common Design Principles

- Design pattern is one of the common design patterns.
- It has become best practice use design patterns when enterprise-level and maintainable software built.

Keep It Simple Stupid (KISS)

- An all-too-common issue in software programming is the need to overcomplicate a solution.
- The goal of the KISS principle is concerned with the need to keep code simple but not simplistic, thus avoiding any unnecessary complexities.

Don’t Repeat Yourself (DRY)

- The DRY principle aims to avoiding repetition of any part of a system by abstracting out things that are common and placing those things in a single location.
- This principle is not only concerned with code but any logic that is duplicated in a system; ultimately there should only be one representation for every piece of knowledge in a system.

Common Design Principles

Tell, Don’t Ask

- The Tell, Don’t Ask principle is closely aligned with encapsulation and the assigning of responsibilities to their correct classes.
- The principle states that you should to tell objects what actions you want them to perform rather than asking questions about the state of the object and then making a decision yourself on what action you want to perform. This helps to align the responsibilities and avoid tight coupling between classes.

You Ain’t Gonna Need It (YAGNI)

- The YAGNI principle refers to the need to only include functionality that is necessary for the application and put off any temptation to add other features that you may think you need.
- A design methodology that adheres to YAGNI is test-driven development (TDD).
- TDD is all about writing tests that prove the functionality of a system and then writing only the code to get the test to pass.

Common Design Principles

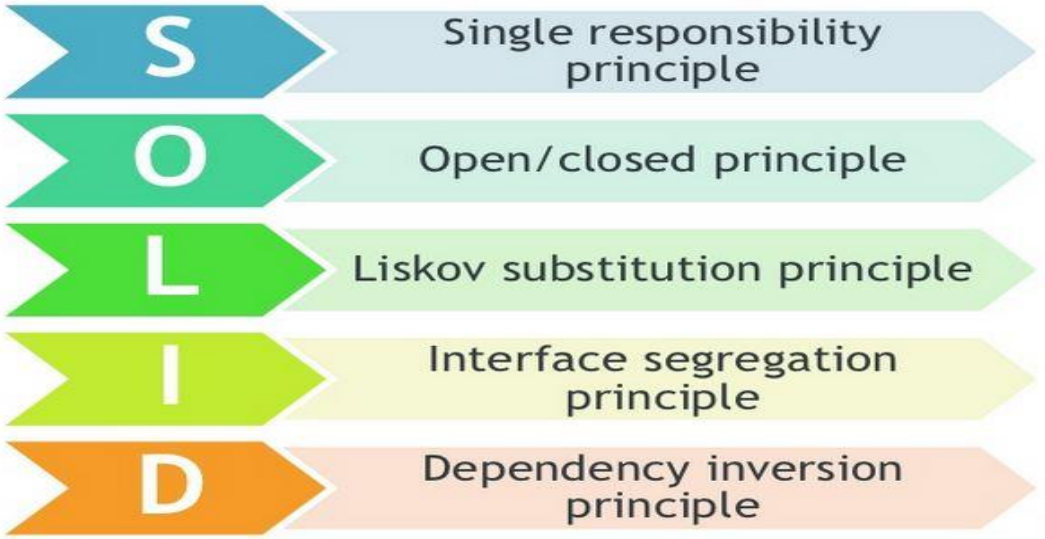
Separation of Concerns (SoC)

- SoC is the process of dissecting a piece of software into distinct features that encapsulate unique behavior and data that can be used by other classes.
- Generally, a concern represents a feature or behavior of a class.
- The act of separating a program into discrete responsibilities significantly increases code reuse, maintenance, and testability.

Common Design Principles

The S.O.L.I.D. Design Principles

- The S.O.L.I.D. design principles are a collection of best practices for object-oriented design.
- All of the Gang of Four design patterns adhere to these principles in one form or another.



Single Responsibility Principle(SRP)

- The principle of SRP is closely aligned with SoC(Separation of Concern).
- It states that every object should only have one reason to change and a single focus of responsibility.
- By adhering to this principle, you avoid the problem of monolithic class design that is the software equivalent of a Swiss army knife.
- By having concise objects, you again increase the readability and maintenance of a system.
- SRP is a way to divide the whole problem into small parts and each part will be dealt with any separate class.

Open/Closed Principle(OCP)

- The OCP states that classes should be open for extension and closed for modification, in that you should be able to add new features and extend a class without changing its internal behavior.
- Once you create the class and other parts of the application start using it, you should not change it.
- If you change the class it will have impact on the working application and it will break.
- If you require additional features you should extend that class rather than modifying it. This way the existing system won't get any impact from the new change.

Liskov Substitution Principle (LS P)

- The LSP dictates that you should be able to use any derived class in place of a parent class and have it behave in the same manner without modification.
- This principle is in line with OCP in that it ensures that a derived class does not affect the behavior of a parent class, or, put another way, derived classes must be substitutable for their base classes.

Interface Segregation Principle (ISP)

- The ISP is all about splitting the methods of a contract into groups of responsibility and assigning interfaces to these groups to prevent a client from needing to implement one large interface and a host of methods that they do not use.
- The purpose behind this is so that classes wanting to use the same interfaces only need to implement a specific set of methods as opposed to a monolithic interface of methods.
- Think of a class that has ten methods – five are needed by desktop and five are needed by mobile clients.

Interface Segregation Principle (ISP)

- Thus the same interface consisting of ten methods is being used by both desktop and mobile clients.
- If desktop application requires some additional methods then we have to add that at interface level and both the application have to be updated to new version.
- ISP suggests you avoid such situations, create two separate interfaces one for desktop and another for mobile clients.

Dependency Inversion Principle (DIP)

- The DIP is all about isolating your classes from concrete implementations and having them depend on abstract classes or interfaces.
- It promotes the mantra of coding to an interface rather than an implementation, which increases flexibility within a system by ensuring you are not tightly coupled to one implementation.

Dependency Injection (DI) and Inversion of Control (IoC)

- Closely linked to the DIP are the DI principle and the IOC principle.
- DI is the act of supplying a low level or dependent class via a constructor, method, or property.
- Used in conjunction with DI, these dependent classes can be inverted to interfaces or abstract classes that will lead to loosely coupled systems that are highly testable and easy to change.

Dependency Injection (DI) and Inversion of Control (IoC)

- In IoC, a system's flow of control is inverted compared to procedural programming.
- An example of this is an IoC container, whose purpose is to inject services into client code without having the client code specifying the concrete implementation.
- The control in this instance that is being inverted is the act of the client obtaining the service.

Gang of Four Pattern Template

- **Pattern Name and Classification:** The Pattern Name is important because it helps to form the common pattern vocabulary.
- The Classification defines the job of the pattern, be it Creational, Structural, or Behavioral.
- **Intent:** The Intent section reveals the problems that the pattern sets out to solve and why it is useful.
- **Also Known As:** The Also Known As section details the other names that some patterns are known as.
- **Motivation:** The Motivation section describes a problem scenario and how to use a design pattern to solve it.

Gang of Four Pattern Template

- **Applicability:** The Applicability section lists the situations when it is advantageous to apply the design pattern.
- **Structure:** The Structure section is a graphical representation of the pattern, including the collaborations and relationships between objects.
- Typically this is shown as a UML diagram.
- **Participants:** The Participants are all the objects involved in the design pattern.
- **Collaborations:** The Collaborations section details how the participants work together to form the design pattern.
- **Consequences:** The Consequences section lists any benefits and liabilities caused when implementing the design pattern.

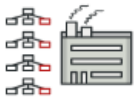
Gang of Four Pattern Template

- **Implementations: The Implementations section details any gotchas and best practices when implementing the design pattern.**
- **Sample Code: The Sample Code section shows an implementation of the design pattern.**
- **Known Uses: The Known Uses section shows implementations of the pattern in real-life applications.**
- **Related Patterns: The Related Patterns section lists other patterns that collaborate or work well with the design pattern**

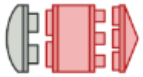
Design Pattern Groups

- Subgroups
- Creational, Structural, or Behavioral

Factory Method



Abstract Factory



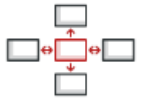
Adapter

Bridge

Chain of Responsibility

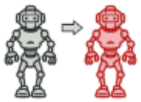
Command

Iterator



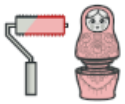
Mediator

Builder

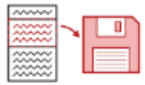


Prototype

Composite

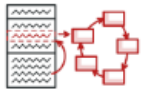


Decorator

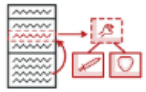


Memento

Observer



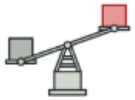
State



Strategy

Singleton

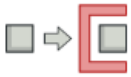
Facade



Flyweight

Template method

Visitor



Proxy

Creational

- Creational patterns deal with object construction and referencing.
- They abstract away the responsibility of instantiating instances of objects from the client, thus keeping code loosely coupled and the responsibility of creating complex objects in one place adhering to the Single Responsibility and Separation of Concerns principles.

Creational

- Following are the patterns in the Creational group:
- **Abstract Factory:** Provides an interface to create families of related objects.
- **Factory:** Enables a class to delegate the responsibility of creating a valid object.
- **Builder:** Enables various versions of an object to be constructed by separating the construction for the object itself.
- **Prototype:** Allows classes to be copied or cloned from a prototype instance rather than creating new instances.
- **Singleton:** Enables a class to be instantiated once with a single global point of access to it.

Structural

- Structural patterns deal with the composition and relationships of objects to fulfill the needs of larger systems.
- Following are the patterns in the Structural group:
- **Adapter:** Enables classes of incompatible interfaces to be used together.
- **Bridge:** As the name suggests, bridge design pattern is used to connect two pieces of code(Abstraction and Implementation). Also, sometimes we need to separate the code so that it is reusable and easy to maintain. And these separate pieces then can be joined together when needed by bridge design pattern.
- **Composite:** Allows a group of objects representing hierarchies to be treated in the same way as a single instance of an object.

Structural

- **Decorator:** Can dynamically surround a class and extend its behavior.
- **Facade:** Provides a simple interface and controls access to a series of complicated interfaces and subsystems.
- **Flyweight:** Provides a way to share data among many small classes in an efficient manner.
- **Proxy:** Provides a placeholder to a more complex class that is costly to instantiate.

Behavioral

- Behavioral patterns deal with the communication between objects in terms of responsibility and algorithms.
- The patterns in this group encapsulate complex behavior and abstract it away from the flow of control of a system, thus enabling complex systems to be easily understood and maintained.

Behavioral

- Following are the patterns in the Behavioral group:
- **Chain of Responsibility:** Allows commands to be chained together dynamically to handle a request.
- **Command:** Encapsulates a method as an object and separates the execution of a command from its invoker.
- **Interpreter:** Specifies how to evaluate sentences in a language.
- **Iterator:** Provides a way to navigate a collection in a formalized manner.
- **Mediator:** Defines an object that allows communication between two other objects without them knowing about one another.

Behavioral

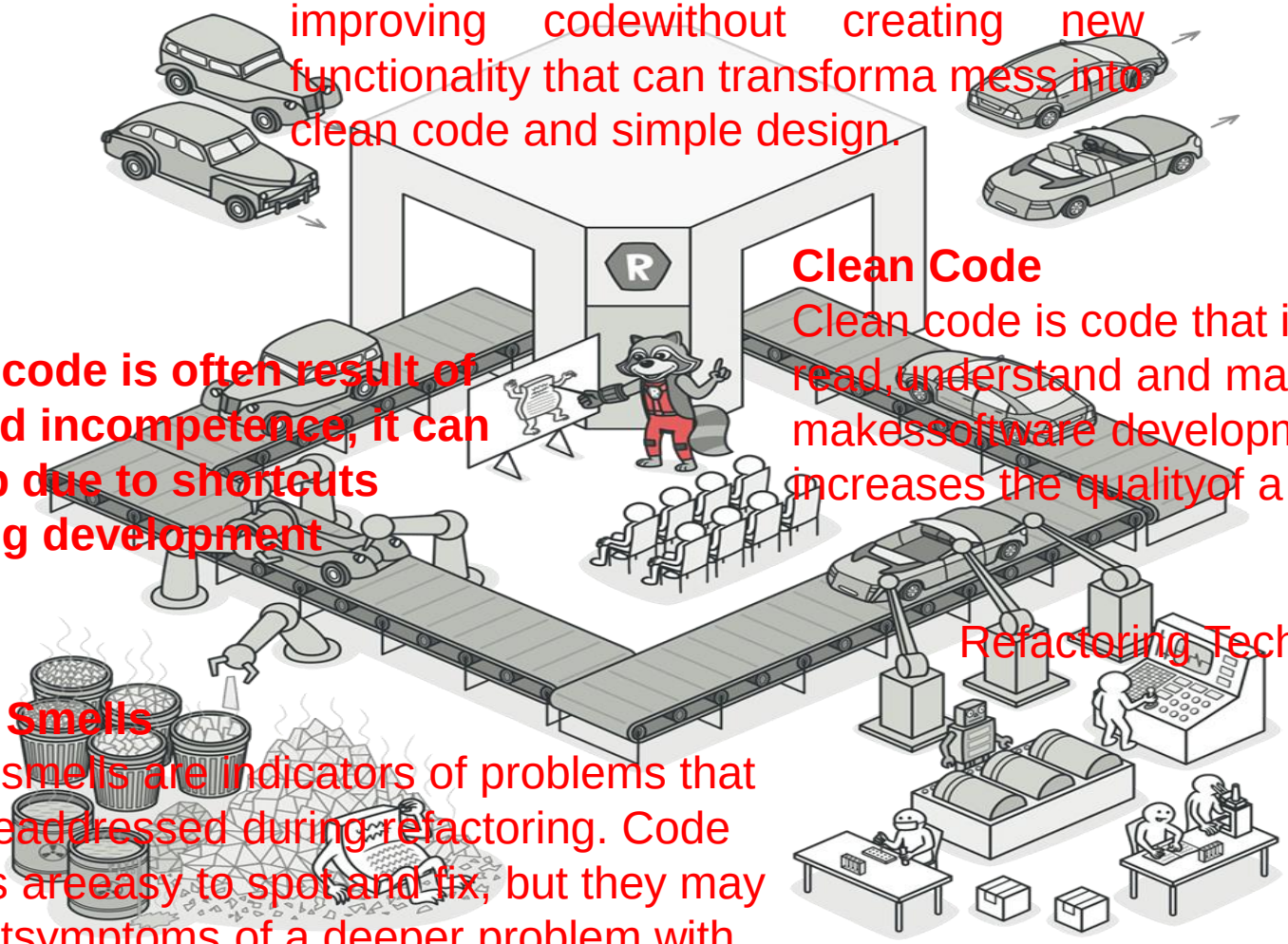
- **Memento:** Allows you to restore an object to its previous state.
- **Observer:** Defines the way one or more classes can be alerted to a change in another class.
- **State:** Allows an object to alter its behavior by delegating to a separate and changeable state object.
- **Strategy:** Enables an algorithm to be encapsulated within a class and switched at run time to alter an object's behavior.
- **Template Method:** Defines the control of flow of an algorithm but allows subclasses to override or implement execution steps.
- **Visitor:** Enables new functionality to be performed on a class without affecting its structure.

Categorization of EAA

- Domain-logic patterns
- Data-source architectural patterns
- Object relational behavioural patterns
- Object relational structural patterns
- Object relational metadata-mapping patterns
- Web presentation patterns
- Distribution patterns
- Offline concurrency patterns
- Session-state patterns
- Base patterns

Refactoring

Refactoring is a systematic process of improving codewithout creating new functionality that can transforma mess into clean code and simple design.



Dirty Code

While dirty code is often result of laziness and incompetence, it can also pile-up due to shortcuts taken during development process.

Clean Code

Clean code is code that is easy to read,understand and maintain. Clean c makessoftware development predictab increases the qualityof a resulting proc

Refactoring Techniques

Code Smells

Code smells are indicators of problems that can beaddressed during refactoring. Code smells areeasy to spot and fix, but they may be justsymptoms of a deeper problem with code.

Code Smells

Bloaters

- Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List

Code Smells

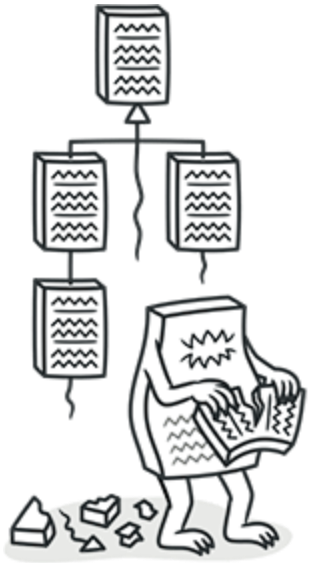
Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.

Alternative Classes with Different Interfaces

Refused Request

Switch Statements



Code Smells

Change Preventers

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too.

Program development becomes much more complicated and expensive as a result.

Divergent Change

Parallel Inheritance Hierarchies

Shotgun Surgery

Code Smells

Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

Comments

Duplicate Code

Data Class

Dead Code

Lazy Class

Speculative Generality

Code Smells

Couplers

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

Feature Envy

Inappropriate Intimacy

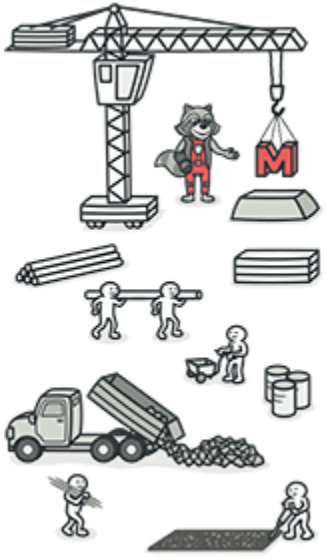
Incomplete Library Class

Message Chains

Refactoring Techniques

Composing Methods

- Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand – and even harder to change.
- The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.
- Extract Method
- Inline Method
- Extract Variable
- Inline Temp



Refactoring Techniques

- Replace Temp with Query
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Substitute Algorithm

Refactoring Techniques

Moving Features between Objects

- Even if you have distributed functionality among different classes in a less-than-perfect way, there is still hope.
- These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access.

- Move Method
- Move Field
- Extract Class
- Inline Class

Refactoring Techniques

- **Hide Delegate**
- **Remove Middle Man**
- **Introduce Foreign Method**
- **Introduce Local Extension**

Refactoring Techniques

Organizing Data

- These refactoring techniques help with data handling, replacing primitives with rich class functionality. Another important result is untangling of class associations, which makes classes more portable and reusable.
- **Change Value to Reference**
- **Change Reference to Value**
- **Duplicate Observed Data**
- **Self Encapsulate Field**
- **Replace Data Value with Object**

Refactoring Techniques

- Replace Array with Object
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional
- Encapsulate Field
- Encapsulate Collection
- Replace Magic Number with Symbolic Constant
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Subclass with Fields

Refactoring Techniques

Simplifying Conditional Expressions

- Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.
- **Consolidate Conditional Expression**
- **Consolidate Duplicate Conditional Fragments**
- **Decompose Conditional**
- **Replace Conditional with Polymorphism**
- **Remove Control Flag**
- **Replace Nested Conditional with Guard Clauses**

Refactoring Techniques

Simplifying Method Calls

These techniques make method calls simpler and easier to understand. This, in turn, simplifies the interfaces for interaction between classes.

Add Parameter

Remove Parameter

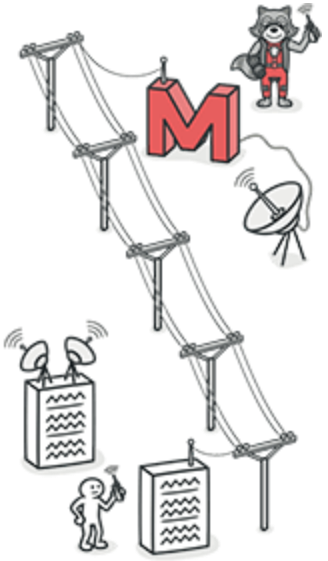
Rename Method

Separate Query from Modifier

Parameterize Method

Introduce Parameter Object

Preserve Whole Object



Refactoring Techniques

Remove Setting Method

Replace Parameter with Explicit Methods

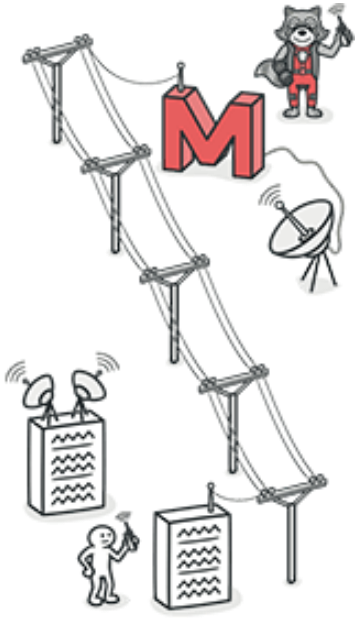
Replace Parameter with Method Call

Hide Method

Replace Constructor with Factory Method

Replace Error Code with Exception

Replace Exception with Test



Refactoring Techniques

Dealing with Generalization

Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa.

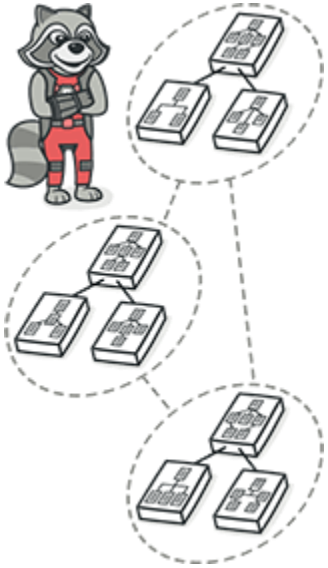
Pull Up Field

Pull Up Method

Pull Up Constructor Body

Push Down Field

Push Down Method



Refactoring Techniques

Extract Subclass

Extract Superclass

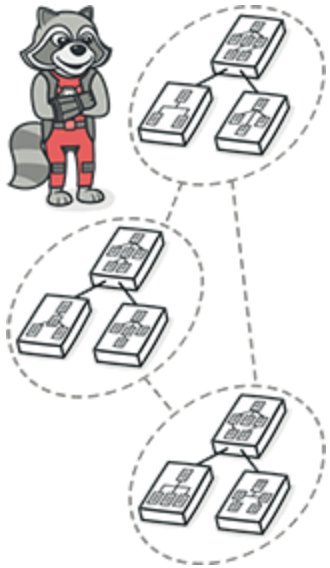
Extract Interface

Collapse Hierarchy

Form Template Method

Replace Inheritance with Delegation

Replace Delegation with Inheritance



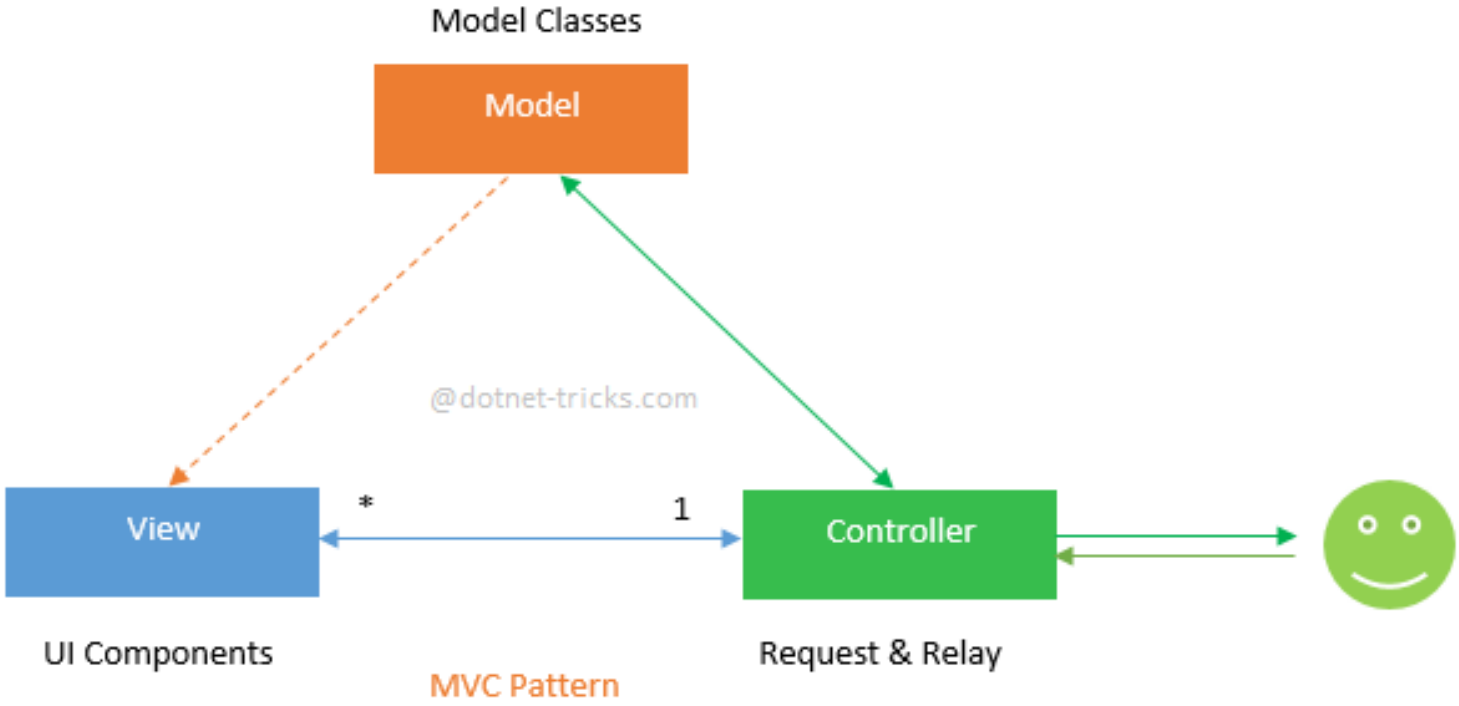
MVC, MVP and MVVM

MVC Pattern

- MVC stands for Model-View-Controller. It is a software design pattern which was introduced in 1970s.
- Also, MVC pattern forces a separation of concerns, it means domain model and controller logic are decoupled from user interface (view).
- As a result maintenance and testing of the application become simpler and easier.
- MVC design pattern splits an application into three main aspects: Model, View and Controller

MVC, MVP and MVVM

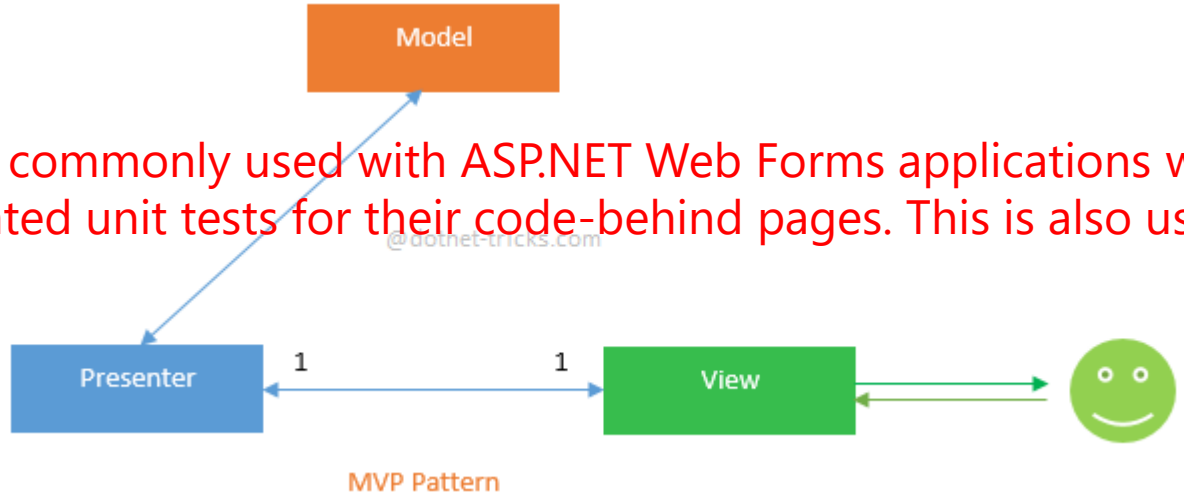
Today, this pattern is used by many popular framework like as Ruby on Rails, Spring Framework, Apple iOS Development and ASP.NET MVC.



MVC, MVP and MVVM

MVP pattern

- This pattern is similar to MVC pattern in which controller has been replaced by the presenter. This design pattern splits an application into three main aspects: Model, View and Presenter.



This pattern is commonly used with ASP.NET Web Forms applications which require to create automated unit tests for their code-behind pages. This is also used with windows forms.

MVC, MVP and MVVM

Presenter

- The Presenter is responsible for handling all UI events on behalf of the view.
- This receive input from users via the View, then process the user's data with the help of Model and passing the results back to the View.
- Unlike view and controller, view and presenter are completely decoupled from each other’s and communicate to each other’s by an interface.
- Also, presenter does not manage the incoming request traffic as controller.

MVC, MVP and MVVM

Key Points about MVP Pattern:

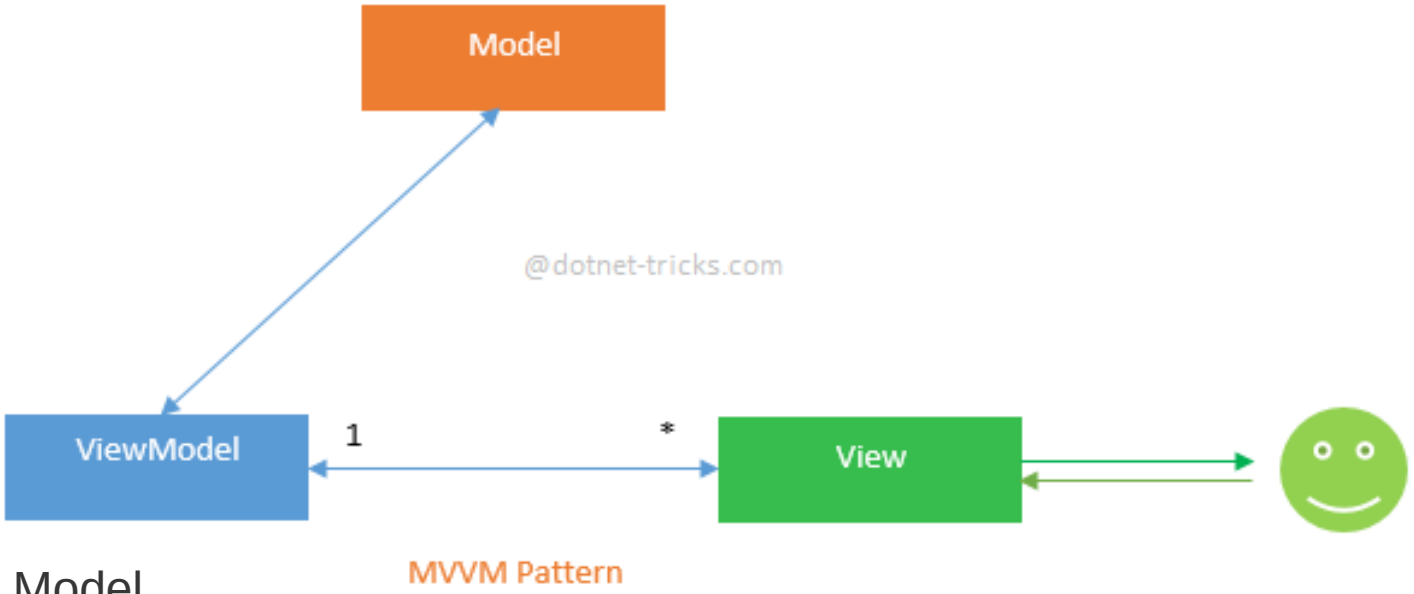
- User interacts with the View.
- There is one-to-one relationship between View and Presenter means one View is mapped to only one Presenter.
- View has a reference to Presenter but View has not reference to Model.
- Provides two way communication between View and Presenter.

MVC, MVP and MVVM

MVVM pattern

- MVVM stands for Model-View-View Model. This pattern supports two-way data binding between view and View model.
- This enables automatic propagation of changes, within the state of view model to the View.
- Typically, the view model uses the observer pattern to notify changes in the view model to model.

MVC, MVP and MVVM

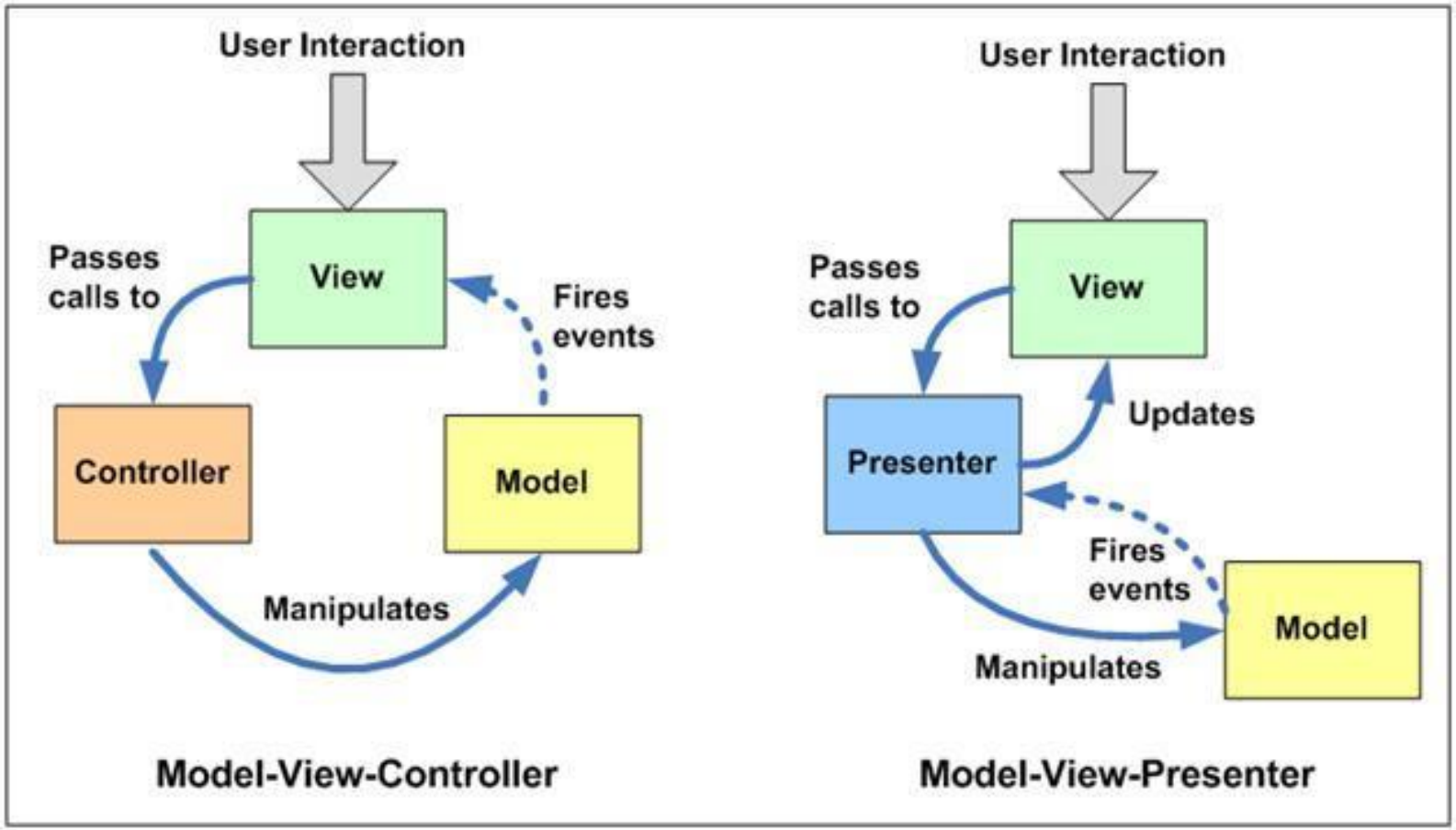


View Model

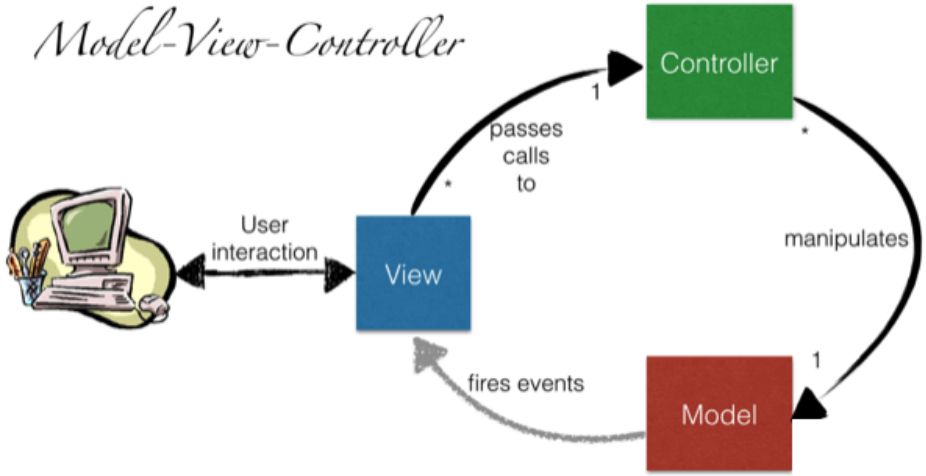
The View Model is responsible for exposing methods, commands, and other properties that helps to maintain the state of the view, manipulate the model as the result of actions on the view, and trigger events in the view itself

MVC, MVP and MVVM

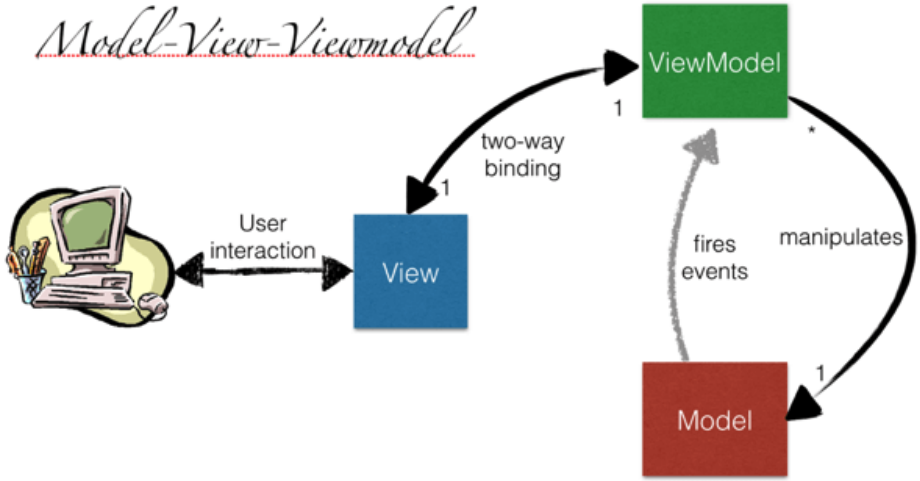
- Key Points about MVVM Pattern:
- User interacts with the View.
- There is many-to-one relationship between View and ViewModel means many View can be mapped to one ViewModel.
- View has a reference to ViewModel but View Model has no information about the View.
- Supports two-way data binding between View and ViewModel.



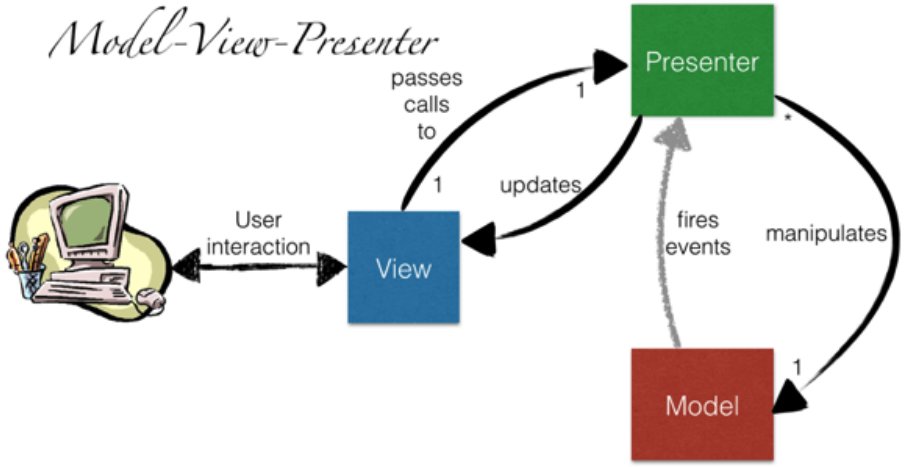
Model-View-Controller



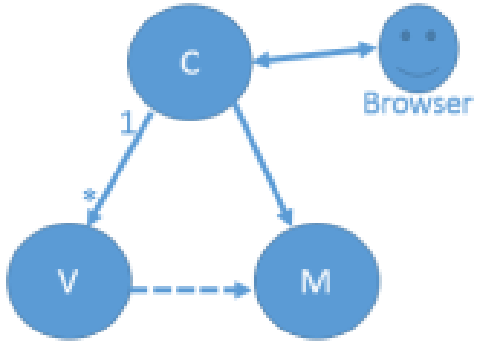
Model-View-Viewmodel



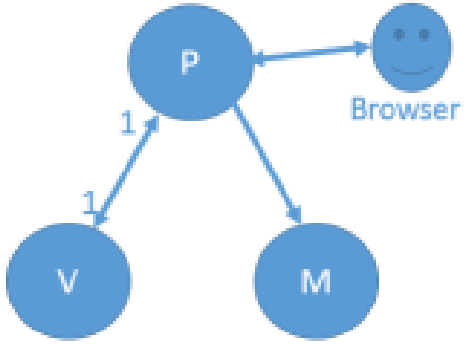
Model-View-Presenter



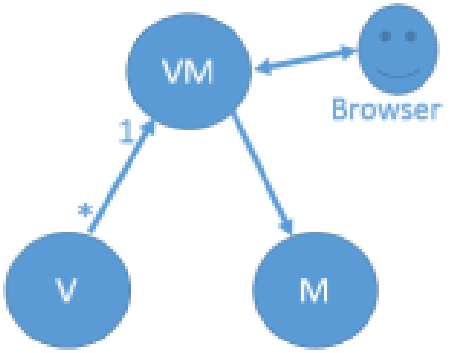
MVC – MVP - MVVM



- Controller is the entry point to the application
- One to Many relationship between Controller and View
- View does not have reference to the Controller
- View is very well aware of the Model
- Smalltalk, ASP.Net MVC

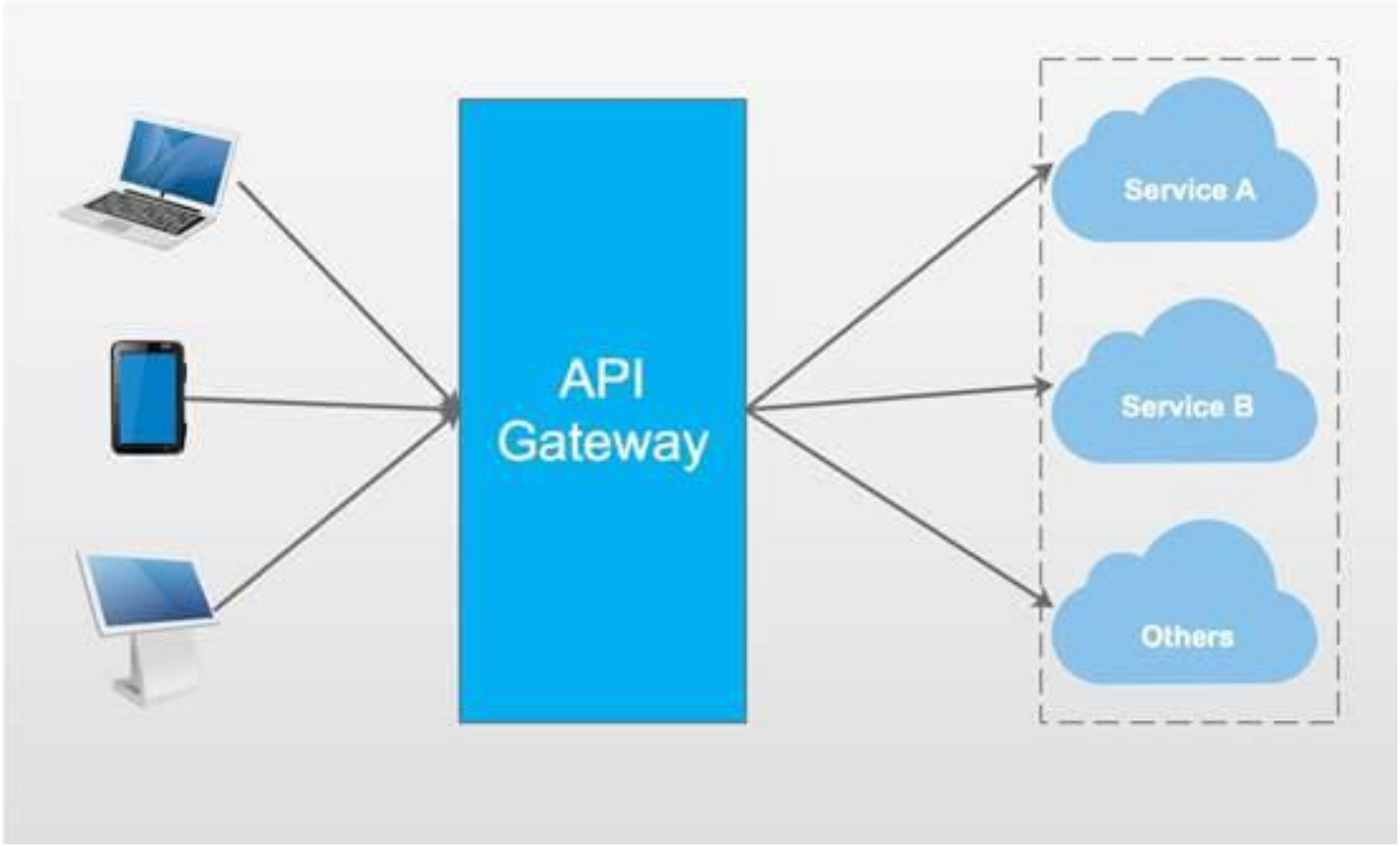


- View is the entry point to the application
- One to One mapping between View and Presenter
- View have the reference to the Presenter
- View is not aware of the Model
- Windows forms



- View is the entry point to the application
- One to Many relationship between View and ViewModel
- View have the reference to the View Model
- View is not aware of the Model
- Silverlight, WPF, HTML5 with Knockout/AngularJS

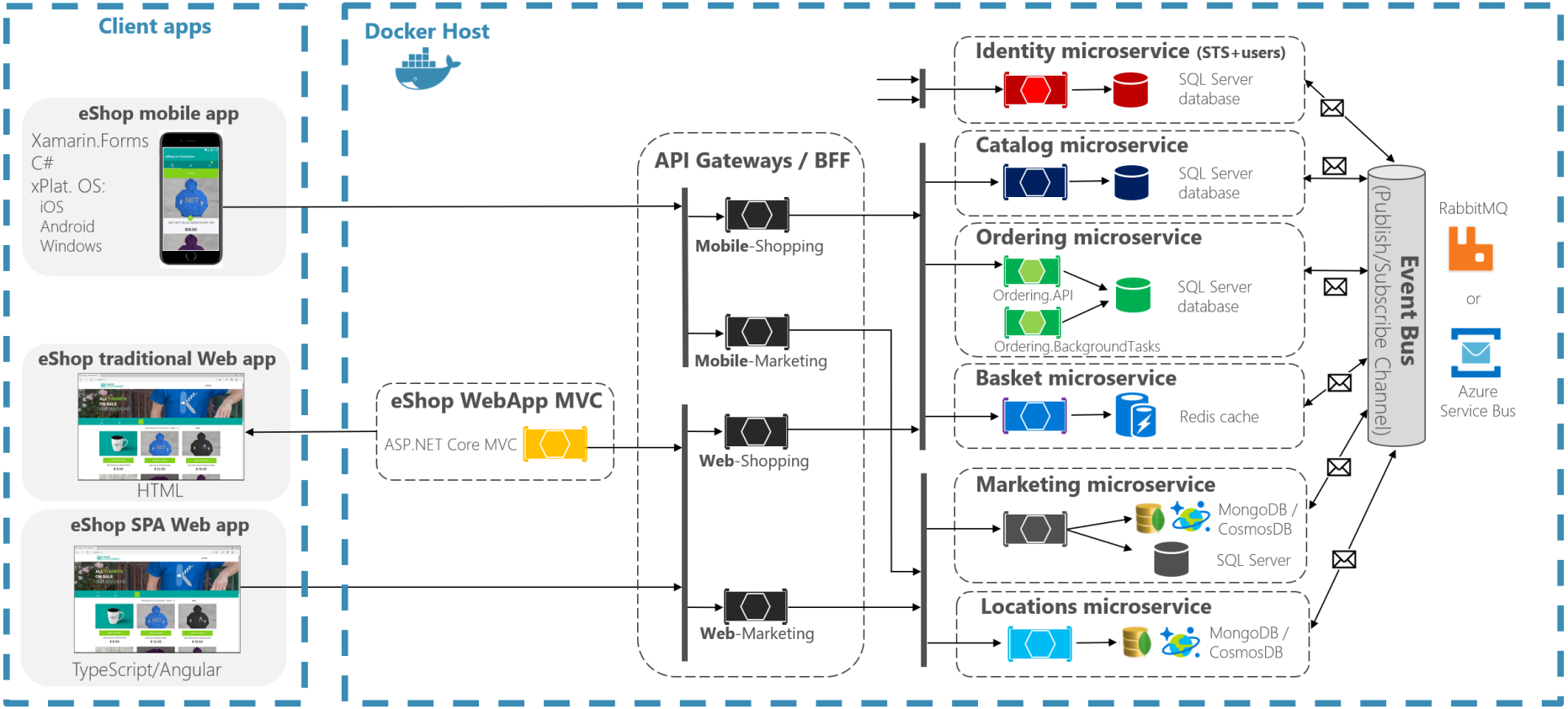
Ocelot API Gateway



Microservice

eShopOnContainers reference application

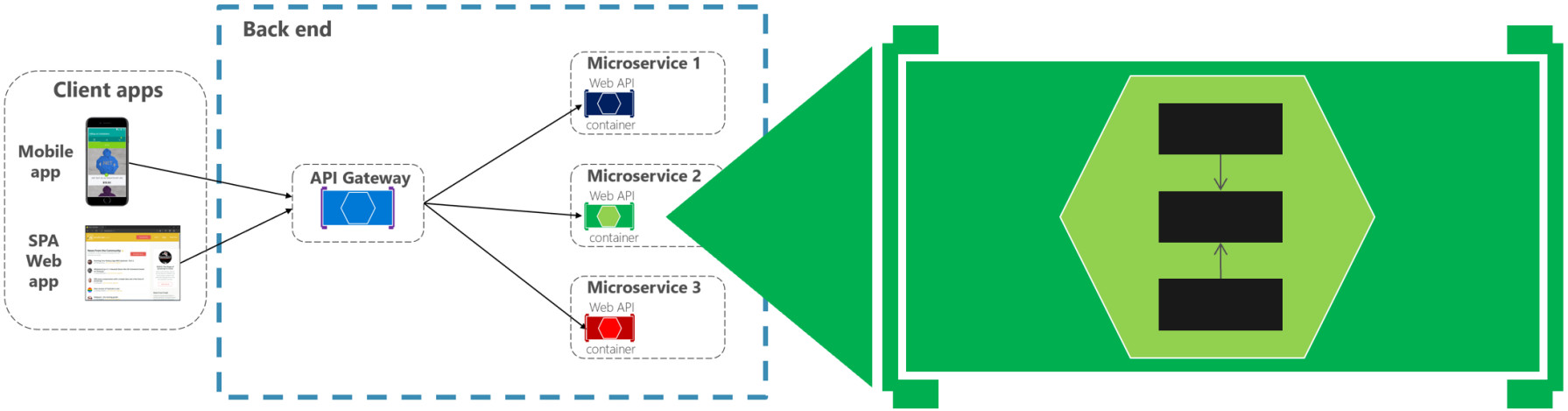
(Development environment architecture)



Microservice

External architecture per application

Internal architecture per microservice



- External microservice patterns
- API Gateway
- Resilient communication
- Pub/Sub and event driven

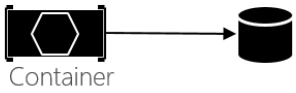
- Simple data driven/CRUD design versus advanced design patterns, DDD, etc.
- Single or multiple libraries
- Dependency Injection, IoC, and SOLID

The Multi-Architectural-Patterns and polyglot microservices world

Microservice 1

- **ASP.NET Core**
- Simple CRUD Design
- Entity Framework Core

Microservice 2

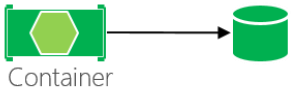


- **ASP.NET Core**
- DDD & CQRS patterns
- EF Core + Dapper

Microservice 3

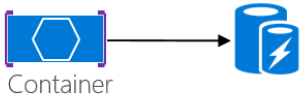
- **ASP.NET Core**
- Queries projection
- DocDB/MongoDB API

Microservice 4



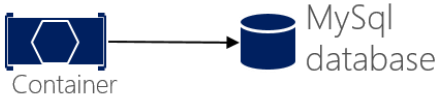
- **NancyFX (.NET Core)**
- Simple CRUD Design
- Massive

Microservice 5



- **ASP.NET Core**
- Simple CRUD Design
- Redis API

Microservice 6

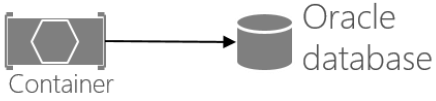


- **Node.js**
- Simple CRUD Design

Microservice 7

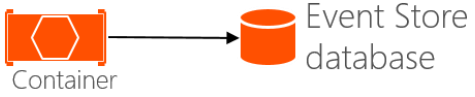
- **Python**
- Simple CRUD Design

Microservice 8



- **Java**
- DDD patterns

Microservice 9



- **ASP.NET Core**
- Event Sourcing patterns
- Event Store API

Microservice 10

- **SignalR (.NET Core 2)**
- Hub for Real Time comm.

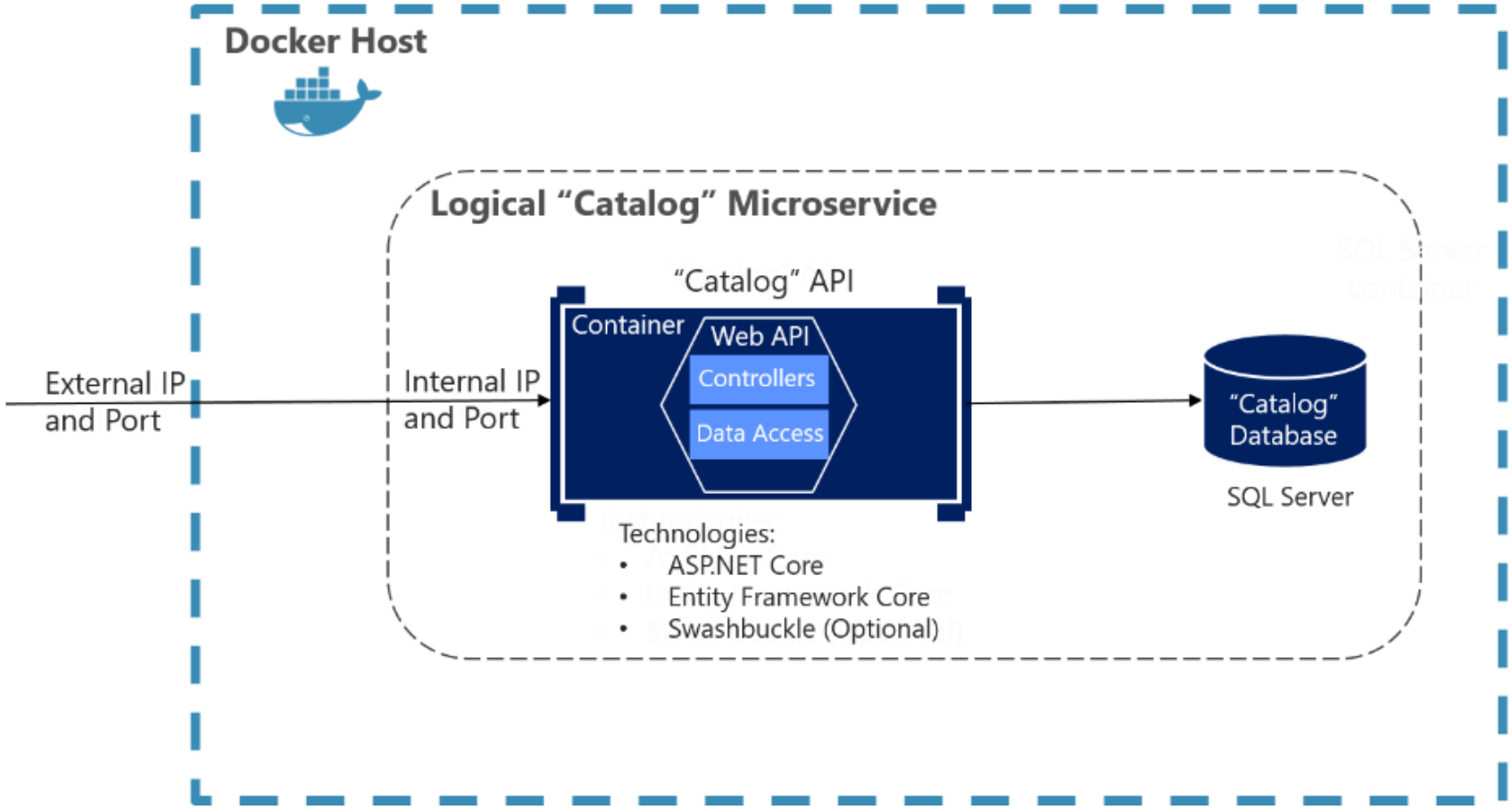
Microservice 11

- **F# .NET Core**
- i.e. Calculus focused

Microservice 10

- **GoLang**
- Stateless process

Data-Driven/CRUD microservice container



Dockerfile	docker-compose.override.yml
This file is the entry point for running any Docker application. It is used to build an image of the application's published code in "obj/Docker/publish."	This file is used when running an application using Visual Studio.
ose.override.yml Dockerfile CoreAppWithDocker FROM microsoft/aspnetcore:2.0 ARG source WORKDIR /app EXPOSE 80 COPY ${source:-obj/Docker/publish} . ENTRYPOINT ["dotnet", "CoreAppWithDocker.dll"]	docker-compose.override.yml Dockerfile CoreAppWithDocker 1 version: '3' 2 3 services: 4 coreappwithdocker: 5 environment: 6 - ASPNETCORE_ENVIRONMENT=Development 7 ports: 8 - "80"

Dockerfile	docker-compose.override.yml

pose.override.yml | Dockerfile | CoreAppWithDocker

```
FROM microsoft/aspnetcore:2.0
ARG source
WORKDIR /app
ENV ASPNETCORE_URLS http://+:83
EXPOSE 83
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", "CoreAppWithDocker.dll"]
```

docker-compose.override.yml | Dockerfile | CoreAppWithDock

```
1  version: '3'
2
3  services:
4    coreappwithdocker:
5      environment:
6        - ASPNETCORE_ENVIRONMENT=Development
7      ports:
8        - "83"
```

Docker ps commands

- Docker ps –a –q
- Docker rm $(docker ps –a –q)

- Dotnet restore
- dotnet publish -o obj/Docker/publish
- docker build -t imagename .
- docker run -d -p 8001:83 –name core1 coreappimage

DSL

- A domain-specific language (DSL) is designed to express statements in a particular problem space, or domain.
- Well-known DSLs include regular expressions and SQL.
- Each DSL is much better than a general-purpose language for describing operations on text strings or a database, but much worse for describing ideas that are outside its own scope.
- Individual industries also have their own DSLs.
- For example, in the telecommunications industry, call description languages are widely used to specify the sequence of states in a telephone call, and in the air travel industry a standard DSL is used to describe flight bookings.

DSL Applications

- Plan of navigation paths in a website.
- Wiring diagrams for electronic components.
- Networks of conveyor belts and baggage handling equipment for an airport.

DSL benefits

- Contains constructs that exactly fit the problem space.
  - An insurance policy application must include elements for policies and claims.
  - A domain-specific language makes it easier to design the application, and find and correct errors of logic.
- Lets non-developers and people who do not know the domain understand the overall design.
  - By using a graphical domain-specific language, you can create a visual representation of the domain so that non-developers can easily understand the design of the application.

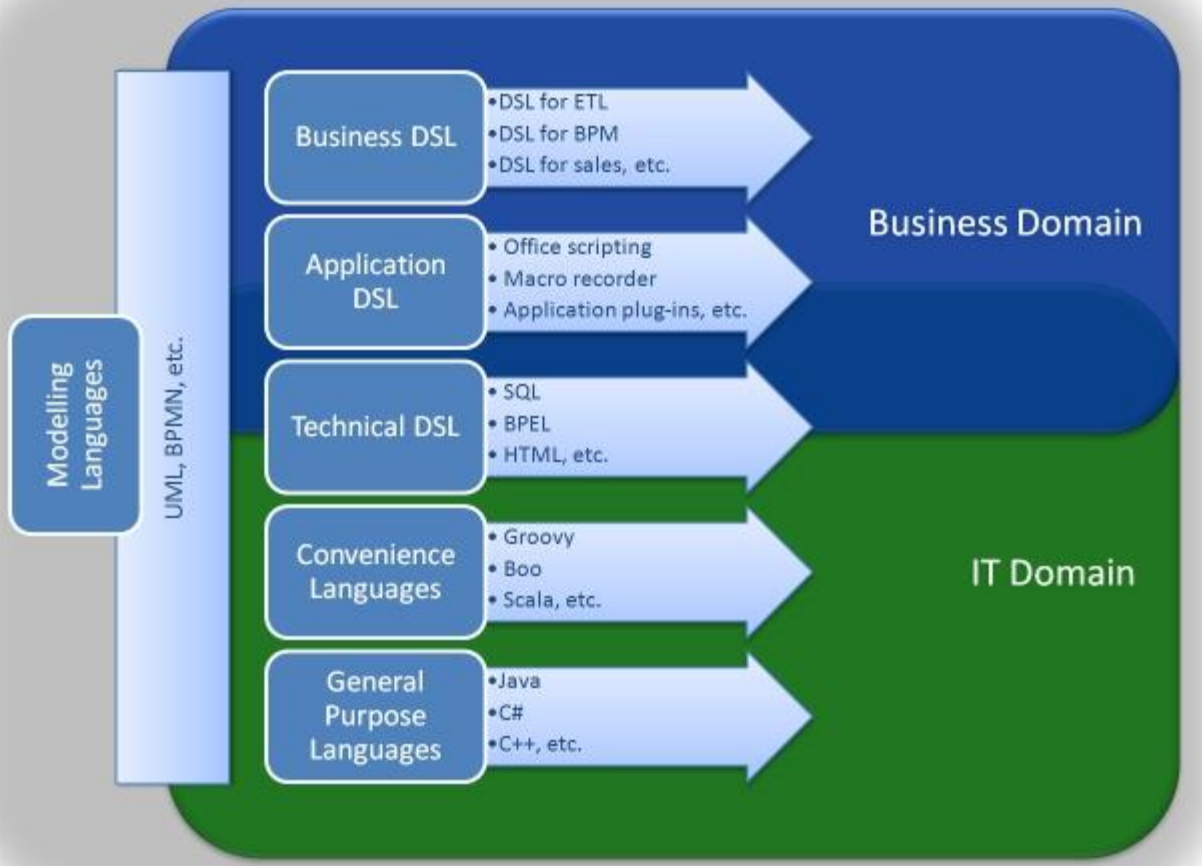
DSL benefits

- Makes it easier to create a prototype of the final application.
  - Developers can use the code that their model generates to create a prototype application that they can show to clients.

The DSL Tools Solution

- Task Flow
- Class Diagrams
- Minimal Language
- Component Models
- Minimal WPF
- Minimal Windows.Forms
- DSL Library

Implementing a Domain Specific Language on .NET



Creational Pattern

- **Singleton Pattern**
- GoF Definition
  - Ensure a class has only one instance, and provide a global point of access to it.
- Concept
  - A particular class should have only one instance. You can use this instance whenever you need it and therefore avoid creating unnecessary objects.

Creational Pattern

Real-Life Example

- Suppose you are a member of a sports team and your team is participating in a tournament.
- When your team plays against another team, as per the rules of the game, the captains of the two sides must have a coin toss.
- If your team does not have a captain, you need to elect someone to be the captain first.
- Your team must have one and only one captain.

Creational Pattern

Computer World Example

- In some software systems, you may decide to maintain only one file system so that you can use it for the centralized management of resources.

Creational Pattern

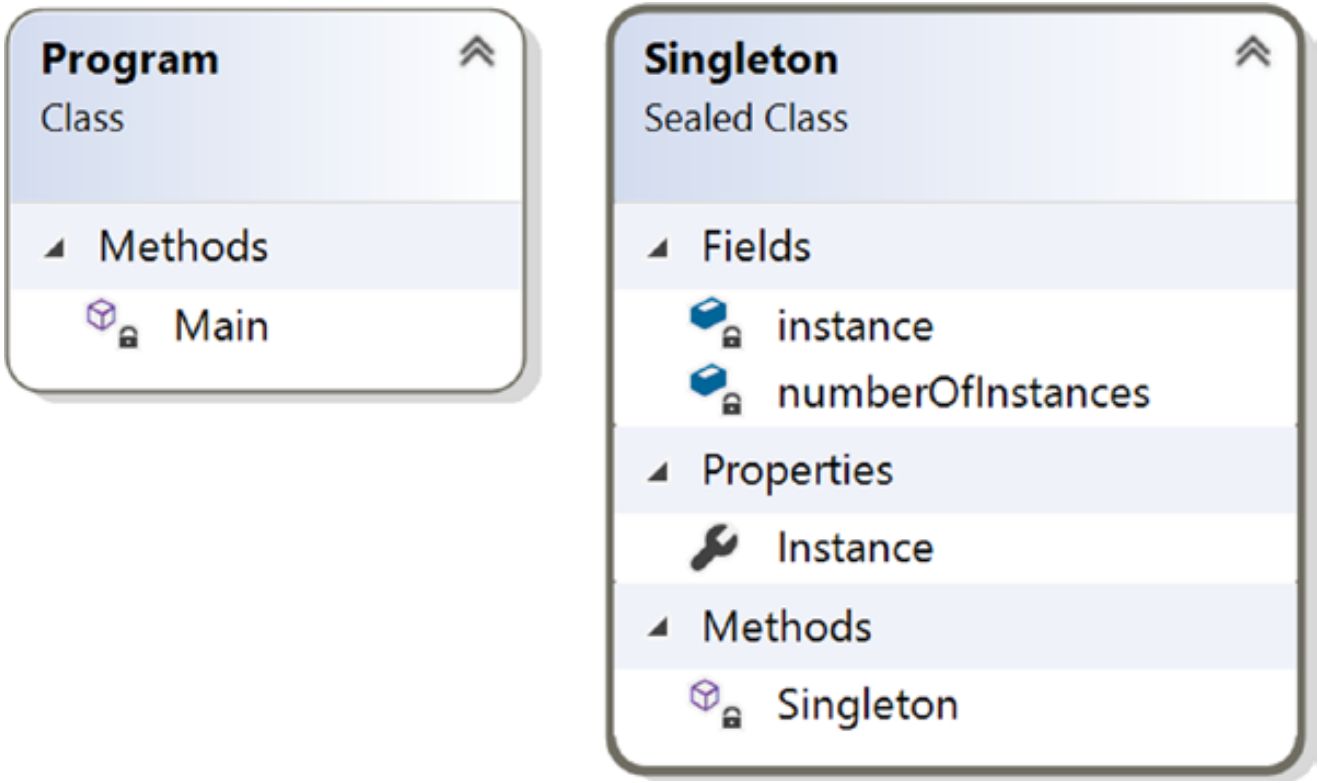
Illustration

- These are the key characteristics in the following implementation:

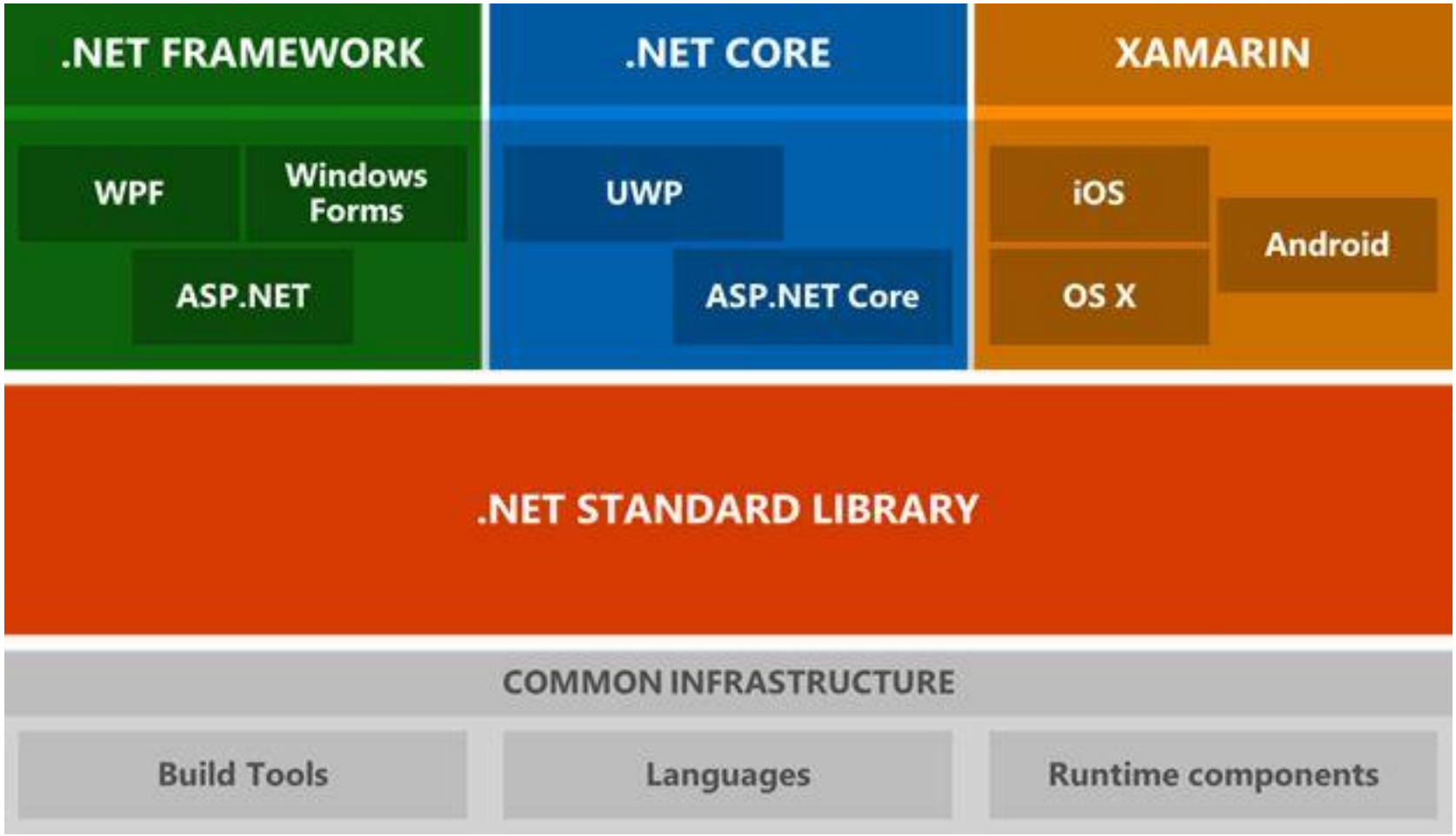
• The constructor is private in this example. So, you cannot instantiate in a normal fashion (using new).

- Before you attempt to create an instance of a class, you check whether you already have an available copy.
- If you do not have any such copy, you create it; otherwise, you simply reuse the existing copy.

Singleton Design Pattern



.net core



Decision

.NET Framework is a better choice if you:

- Do not have time to learn a new technology.
- Need a stable environment to work in.
- Have nearer release schedules.
- Are already working on an existing app and extending its functionality.
- Already have an existing team with .NET expertise and building production ready software.
- Do not want to deal with continuous upgrades and changes.
- Building Windows client applications using Windows Forms or WPF

Decision

. NET Core is a better choice if you:

- Want to target your apps on Windows, Linux, and Mac operating systems.
- Are not afraid of learning new things.
- Are not afraid of breaking and fixing things since .NET Core is not fully matured yet.
- A student who is just learning .NET.
- Love open source.

Decision

High-performance and scalable system without UI	.NET Core is much faster.
Docker containers support	Both. But .NET Core is born to live in a container.
Heavily reply on command line	.NET Core has a better support.
Cross-platform needs	.NET Core
Using Microservices	Both but .NET Core is designed to keep in mind today's needs.
User interface centric Web applications	.NET Framework is better now until .NET Core catches up.
Windows client applications using Windows Forms and WPF	.NET Framework
Already have a pre-configured environment and systems	.NET Framework is better.
Stable version for immediate need to build and deploy	.NET Framework has a been around since 2001. .NET Core is just a baby.
Have existing experienced .NET team	.NET Core has a learning curve.
Time is not a problem. Experiments are acceptable. No rush to deployment.	.NET Core is the future of .NET.

Alternative Approach for single threaded model

```
public static Singleton Instance
{
get
{
if (instance == null)
{
instance = new Singleton();
}
return instance;
}
}
```

Alternative Approach for single threaded model

- //Double checked locking
- using System;
- public sealed class Singleton

{

//We are using volatile to ensure that

//assignment to the instance variable finishes before it's

//access.

private static volatile Singleton instance;

private static object lockObject = new Object();

private Singleton() { }

Alternative Approach for single threaded model

```
public static Singleton Instance
{
get
{
if (instance == null)
{
lock (lockObject)
{
if (instance == null)
instance = new Singleton();
}
}
return instance;
}
}
```

Singleton

- The volatile keyword indicates that a field might be modified by multiple threads that are executing at the same time.
- Fields that are declared volatile are not subject to compiler optimizations that assume access by a single thread.
- This ensures that the most up-todate value is present in the field at all times.
- In simple terms, the volatile keyword can help you to provide a serialize access mechanism.
- In other words, all threads will observe the changes by any other thread as per their execution order.
- You will also remember that the volatile keyword is applicable for class (or struct) fields; you cannot apply it to local variables.

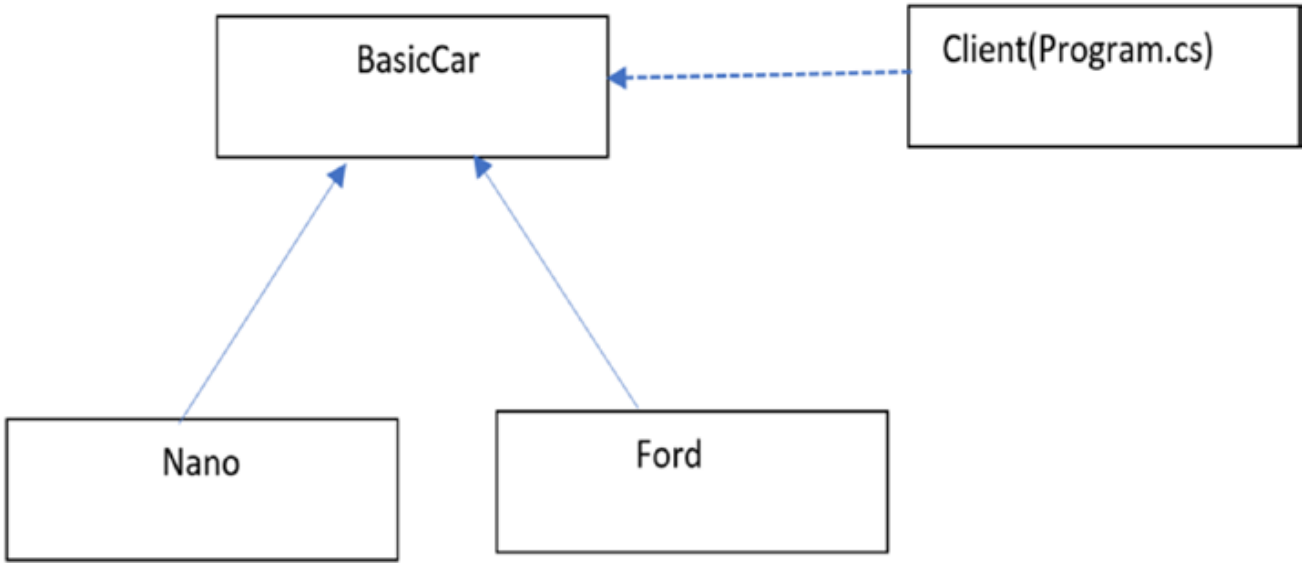
Prototype Pattern

- GoF Definition
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Concept
- This pattern provides an alternative method for instantiating new objects by copying or cloning an instance of an existing object.
- You can avoid the expense of creating a new instance using this concept.

Prototype Pattern

- Real-Life Example
- Suppose you have a master copy of a valuable document.
- You need to incorporate some change into it to analyze the effect of the change.
- In this case, you can make a photocopy of the original document and edit the changes in the photocopied document.
- Computer World Example
- Let’s assume that you already have an application that is stable.
- In the future, you may want to modify the application with some small changes.
- You must start with a copy of your original application, make the changes, and then analyze further.
- Surely you do not want to start from scratch to merely make a change; this would cost you time and money.

Prototype Pattern



Prototype Pattern

What are the advantages of using the Prototype design pattern?

Answer:

• You can include or discard products at runtime.

• In some contexts, you can create new instances with a cheaper cost.

• You can focus on the key activities rather than focusing on complicated instance creation processes.

• Clients can ignore the complex creation process for objects and instead clone or copy objects.

Prototype Pattern

What are the challenges associated with using the Prototype design pattern?

Answer:

• Each subclass must implement the cloning or copying mechanism.

• Implementing the cloning mechanism can be challenging if the objects under consideration do not support copying or if there are circular references.

• In this example, I have used the MemberwiseClone() member that performs a shallow copy in C#.

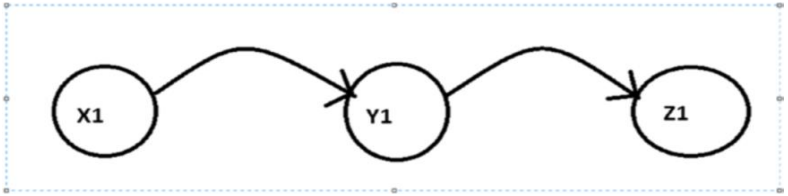
Actually, it creates an object and then copies the non static fields of the current object into the newly created object.

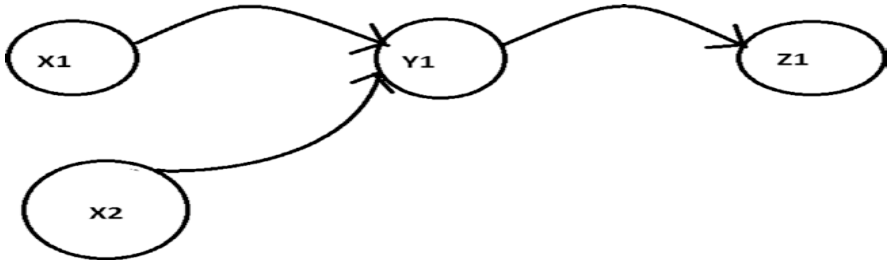
MSDN further says that for a value type field, it performs a bit-by-bit copy, but for a reference type field, the references are copied but referred objects are not copied. So, the original object and the cloned object both refer to the same object.

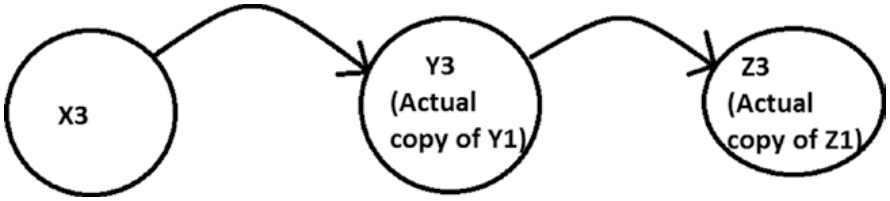
If you need a deep copy in your application, that can be expensive.

Prototype Pattern

- A shallow copy creates a new object and then copies the non static fields from the original object to the new object.
- If there exists a value type field in the original object, a bit-by-bit copy is performed.
- But if the field is a reference type, this method will copy the reference, not the actual object.



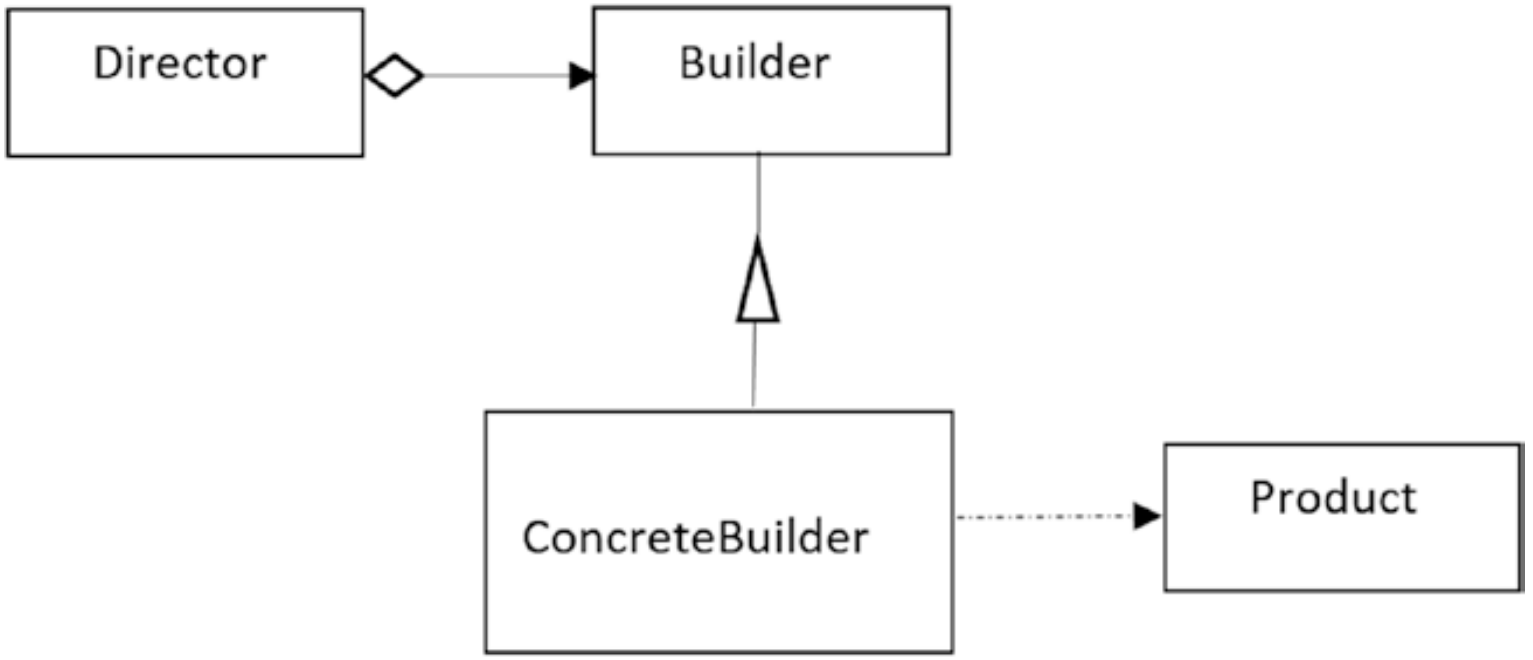




Builder Pattern

- GoF Definition
- Separate the construction of a complex object from its representation so that the same construction processes can create different representations.
- Concept
- The Builder pattern is useful for creating complex objects that have multiple parts.
- The creation process of an object should be independent of these parts; in other words, the construction process does not care how these parts are assembled.
- In addition, you should be able to use the same construction process to create different representations of the objects.

Builder Pattern



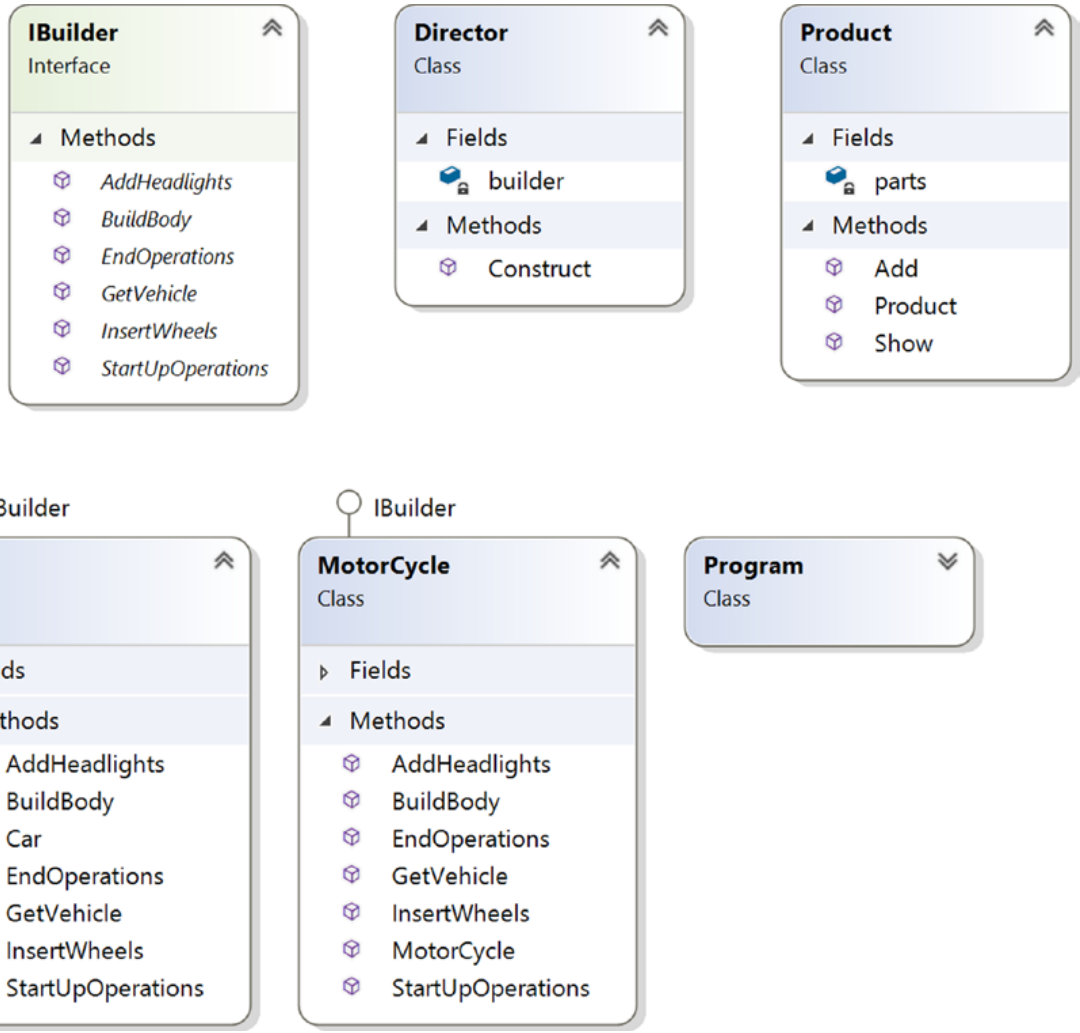
Builder Pattern

- Here, Product is the complex object under consideration.
- ConcreteBuilder constructs and assembles the parts of a Product object by implementing an abstract interface called Builder.
- The ConcreteBuilder objects builds the internal representations of the products and defines the creation process and assembly mechanisms.
- Director is responsible for creating the final object using the Builder interface.

Builder Pattern

- Real-Life Example
- To complete an order for a computer, different hardware parts are assembled based on customer preferences. For example, a customer can opt for a 500GB hard disk with an Intel processor, and another customer can choose a 250GB hard disk with an AMD processor.
- Computer World Example
- You can use this pattern when you want to convert one text format to another text format, such as converting from RTF to ASCII.

Builder Pattern



Builder Pattern Advantages

- You direct the builder to build the objects step-by-step, and you promote encapsulation by hiding the details of the complex construction process.
- The director can retrieve the final product from the builder when the whole construction is over.
- In general, at a high level, you seem to have only one method that makes the complete product, but other internal methods are involved in the creation process. So, you have finer control over the construction process.
- Using this pattern, the same construction process can produce different products.
- • You can also vary the internal representation of products.

Builder Pattern Drawbacks

Answer:

• It is not suitable if you want to deal with mutable objects (which can be modified later).

• You may need to duplicate some portion of the code. These duplications may have significant impact in some contexts.

• To create more products, you need to create more concrete builders.

Factory Method

- GoF Definition
- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- The Factory Method pattern lets a class defer instantiation to subclasses.
- Real-Life Example
- In a restaurant, based on customer inputs, a chef varies the taste of dishes to make the final products.

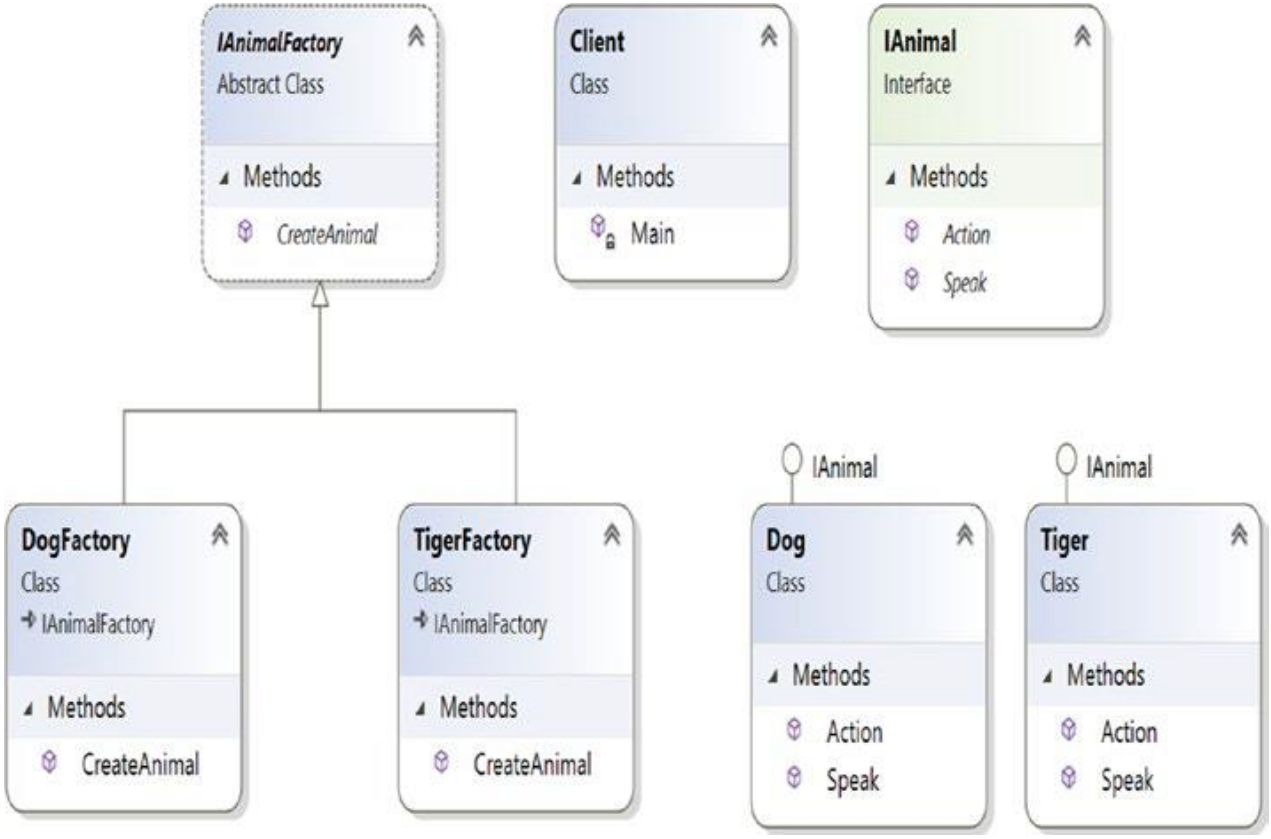
Factory Method

- GoF Definition
- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- The Factory Method pattern lets a class defer instantiation to subclasses.
- Real-Life Example
- In a restaurant, based on customer inputs, a chef varies the taste of dishes to make the final products.

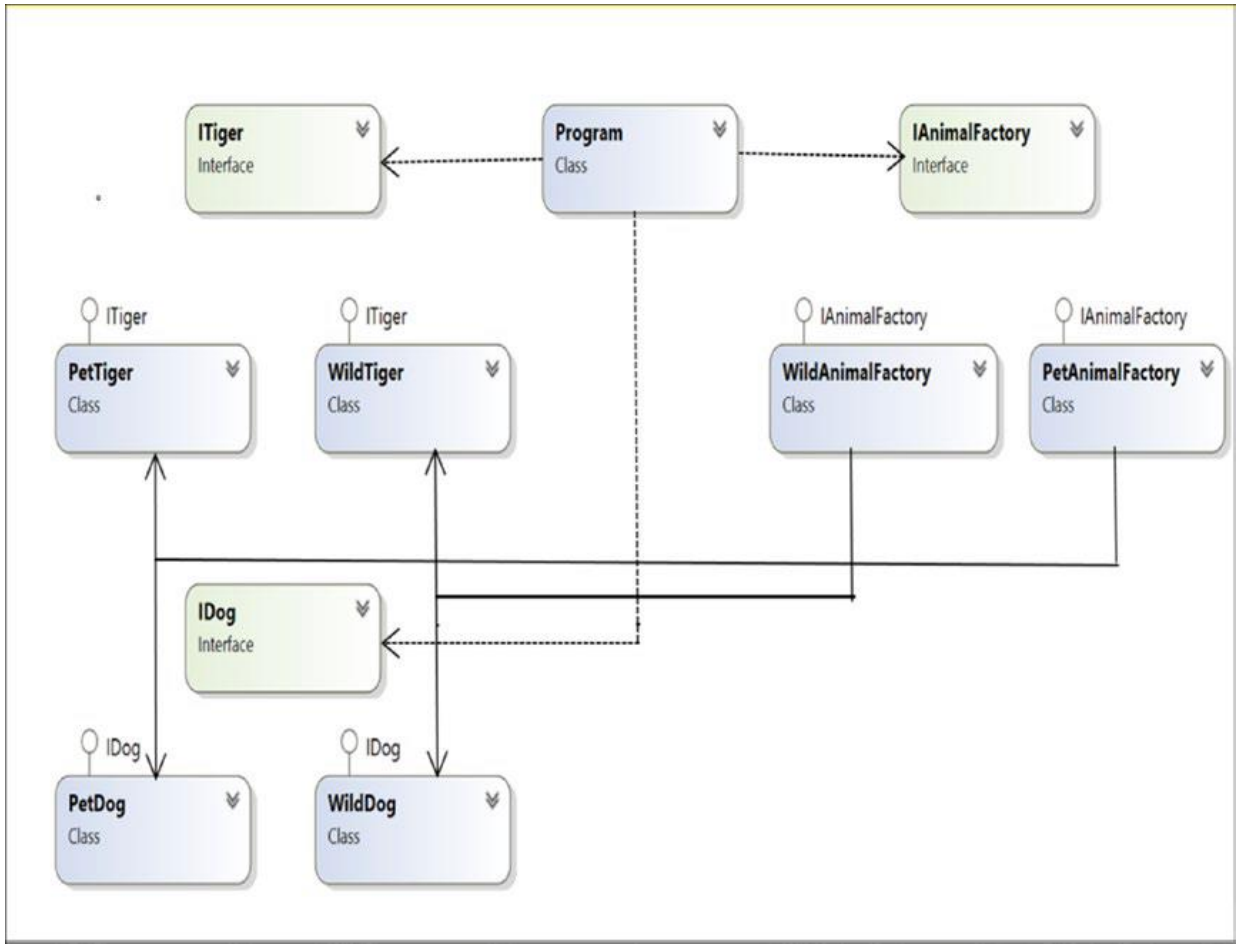
Factory Method

- Computer World Example
- In an application, you may have different database users. For example, one user may use Oracle, and the other may use SQL Server.
- Whenever you need to insert data into your database, you need to create either a SqlConnection or an OracleConnection and only then can you proceed.
- If you put the code into if-else (or switch) statements, you need to repeat a lot of code, which isn't easily maintainable.
- This is because whenever you need to support a new type of connection, you need to reopen your code and make those modifications. This type of problem can be resolved using the Factory Method pattern.
- Here I'll provide an abstract creator class (IAnimalFactory) to define the basic structure.
- As per the definition, the instantiation process will be carried out through the subclasses that derive from this abstract class.

Factory Method



Abstract Factory Pattern



Observer Pattern

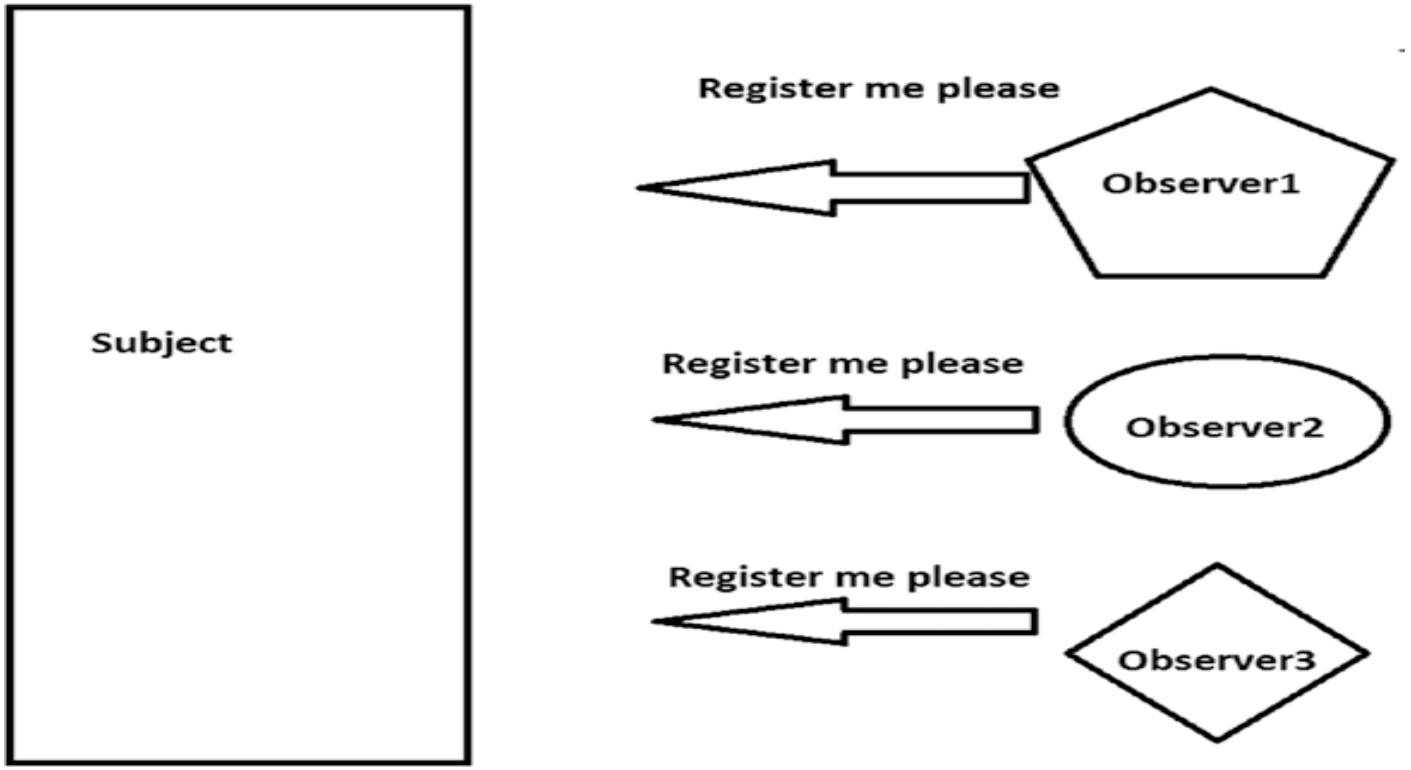
GoF Definition

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Concept

- In this pattern, there are many observers (objects) that are observing a particular subject (also an object).
- Observers want to be notified when there is a change made inside the subject.
- So, they register themselves to that subject.
- When they lose interest in the subject, they simply unregister from the subject.
- Sometimes this model is referred as the Publisher-Subscriber model. The whole idea can be summarized as follows: using this pattern, an object (subject) can send notifications to multiple observers (a set of objects) at the same time.

Observer Pattern



Observer Pattern

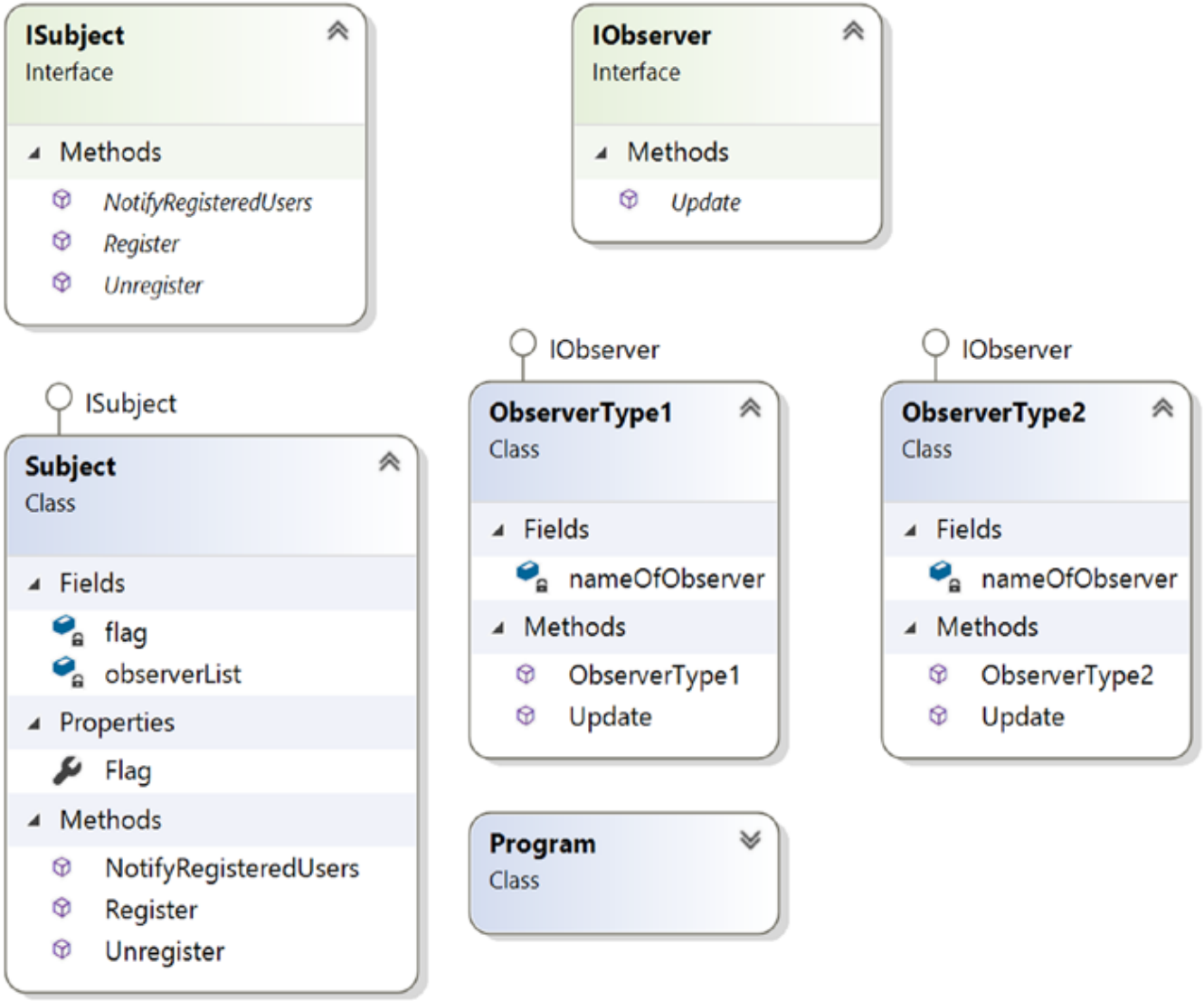
Real-Life Example

- Think about a celebrity who has many followers on social media.
- Each of these followers wants to get all the latest updates from their favorite celebrity.
- So, they follow the celebrity until their interest wanes.
- When they lose interest, they simply do not follow that celebrity.
- Think of each of these fans or followers as an observer and the celebrity as the subject.

Observer Pattern

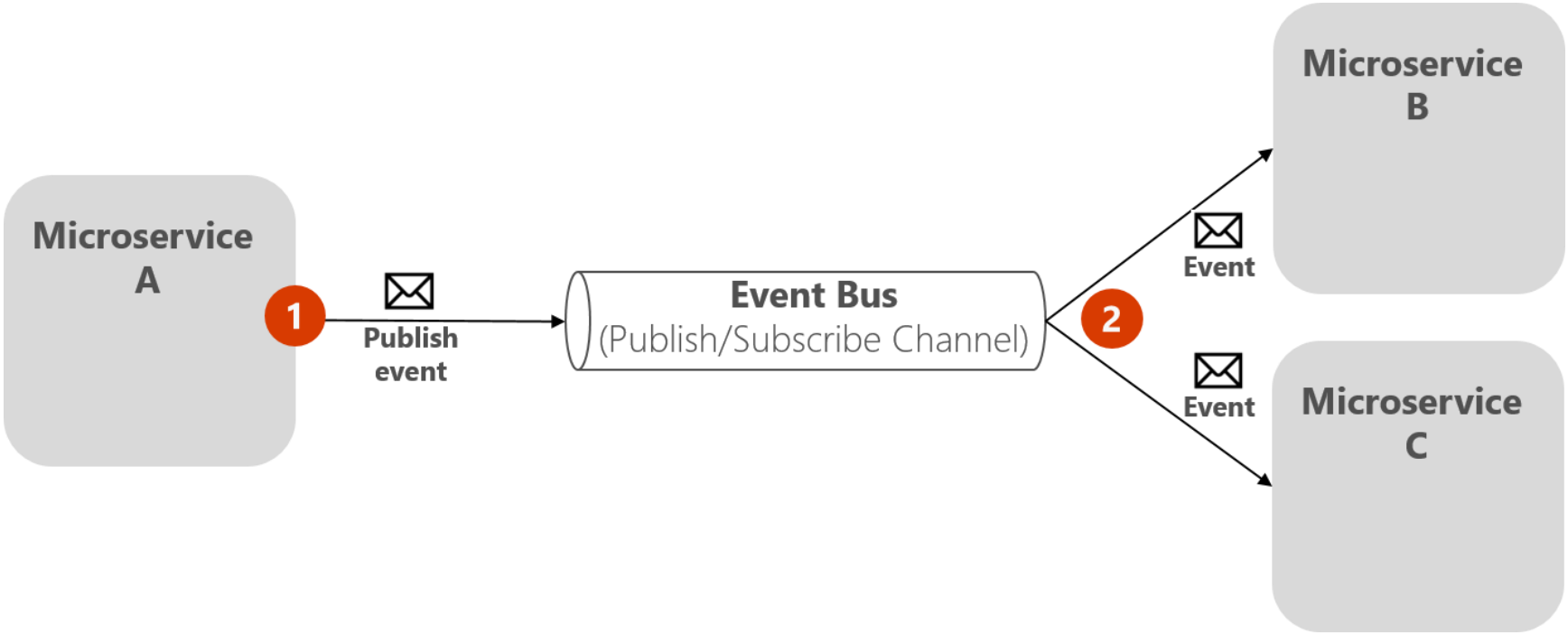
Computer World Example

- In the world of computer science, consider a simple UI-based example.
- This UI is connected to some database.
- A user can execute some query through that UI, and after searching the database, the result is returned in the UI.
- With this pattern, you segregate the UI from the database.
- If a change occurs in the database, the UI should be notified so that it can update its display according to the change.



The event bus

An event bus allows publish/subscribe-style communication between microservices without requiring the components to explicitly be aware of each other



The event bus is related to the Observer pattern and the publish-subscribe pattern.

Observer Pattern

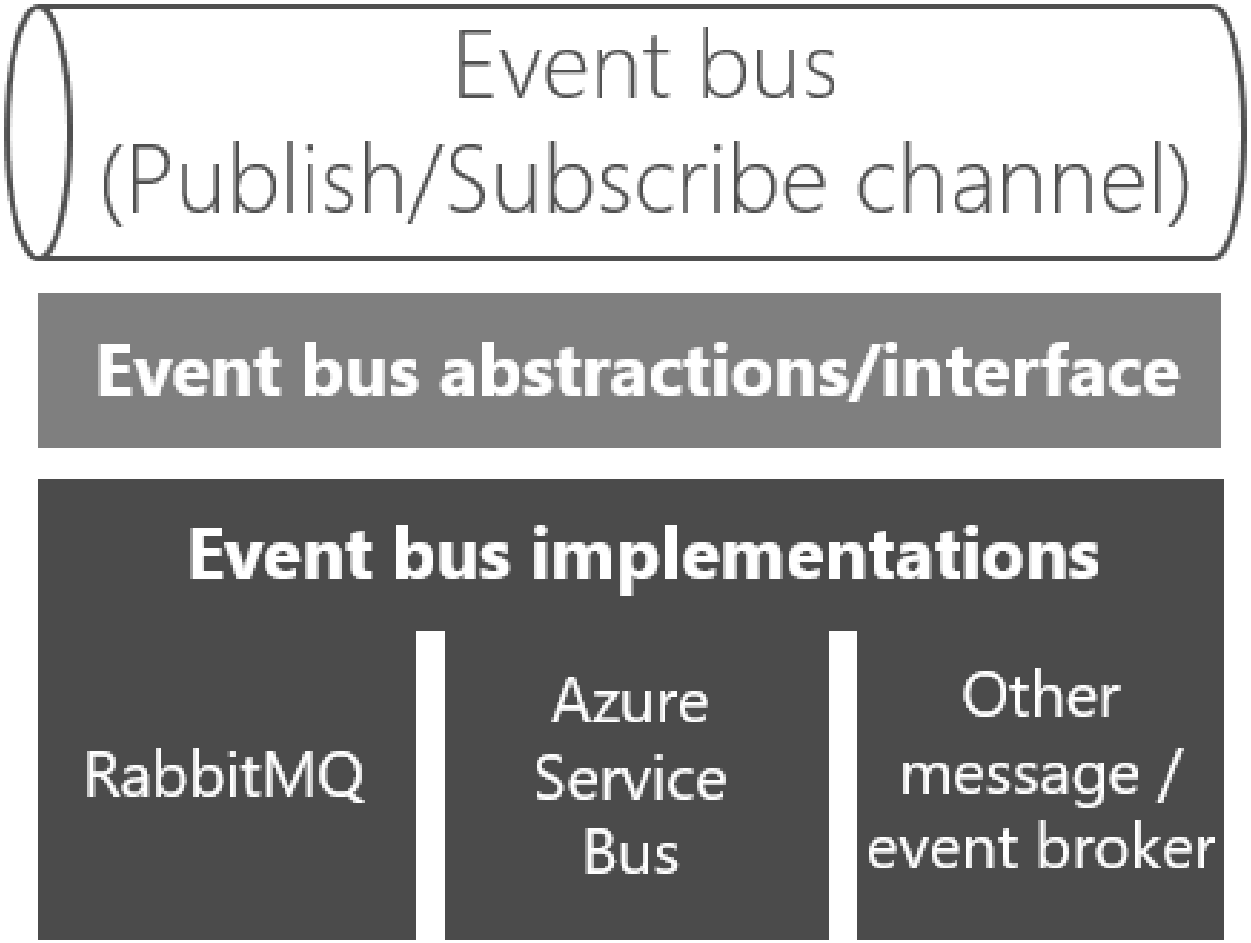
- In the Observer pattern, your primary object (known as the Observable) notifies other interested objects (known as Observers) with relevant information (events).

Publish/Subscribe (Pub/Sub) pattern

- The purpose of the Publish/Subscribe pattern is the same as the Observer pattern: you want to notify other services when certain events take place.
- But there is an important difference between the Observer and Pub/Sub patterns.
- In the observer pattern, the broadcast is performed directly from the observable to the observers, so they “know” each other.
- But when using a Pub/Sub pattern, there is a third component, called broker or message broker or event bus, which is known by both the publisher and subscriber.
- Therefore, when using the Pub/Sub pattern the publisher and the subscribers are precisely decoupled thanks to the mentioned event bus or message broker.

The middleman or event bus

- How do you achieve anonymity between publisher and subscriber? An easy way is let a middleman take care of all the communication. An event bus is one such middleman.
- An event bus is typically composed of two parts:
- The abstraction or interface.
- One or more implementations.



Iterator Pattern

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

Iterator Pattern

- Concept
- Iterators are generally used to traverse a container (which is basically an object) to access its elements, but you do not need to deal with the element's internal details.
- You will frequently use the concepts of iterators when you want to traverse different kinds of collection objects in a standard and uniform way.
- This concept is used frequently to traverse the nodes of a tree-like structure. So, in many scenarios, you may notice the use of the Iterator pattern with the Composite pattern.
- C# has its own iterators that were introduced in Visual Studio 2005.
- The foreach statement is frequently used in this context..

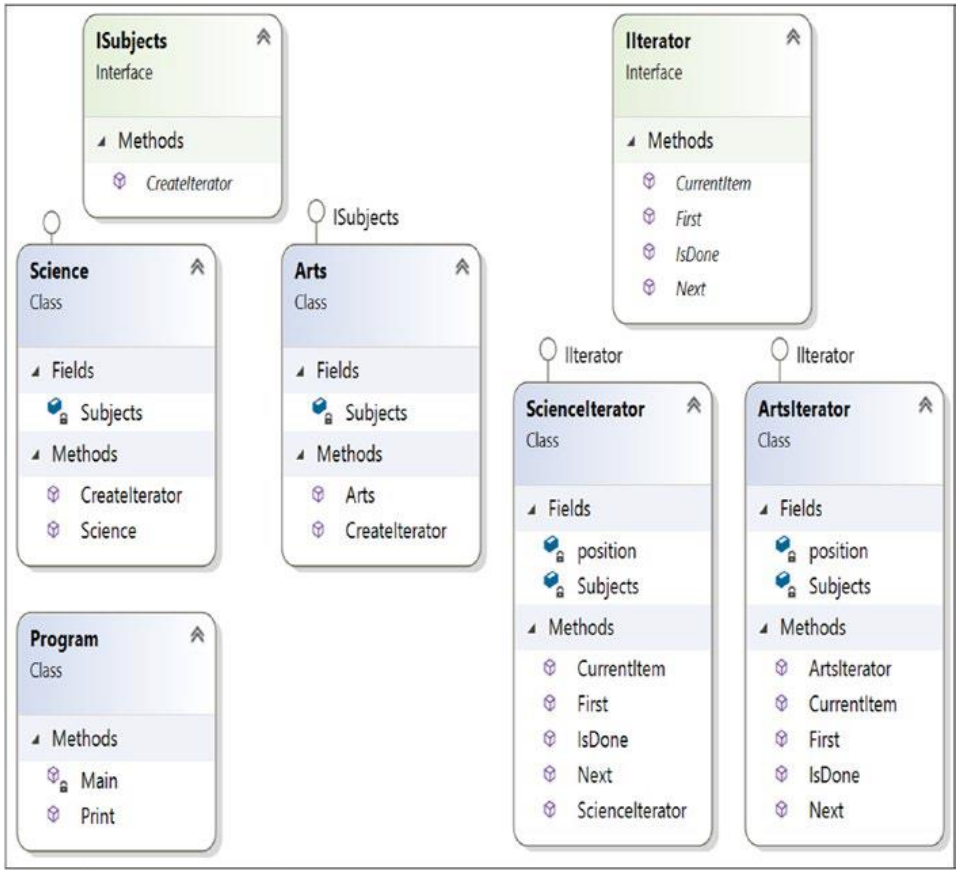
Iterator Pattern

- Real-Life Example
- Suppose there are two companies, Company A and Company B. Company A stores its employee records (say each employee's name, address, salary details, and so on) in a linked list data structure, and Company B stores its employee data in an array.
- One day the two companies decide to merge. The Iterator pattern becomes handy in such a situation because you do need not to write any code from scratch.
- In a situation like this, you can have a common interface through which you can access the data for both companies. So, you can simply call those methods without rewriting the code.

Iterator Pattern

- Computer World Example
- Similarly suppose, in a college, the arts department is using an array data structure to maintain its student records, and the science department is using a linked list data structure to keep its student records.
- The administrative department does not care about the different data structures.
- It is simply interested in getting the data from each department and wants to access the data in a uniform way.

Iterator Pattern



Services

File Action View Help

Services (Local)

Services (Local)

RabbitMQ

Start the service

Description:
Multi-protocol open source messaging broker

Command Prompt - rabbitmq-server.bat start

```
D:\Program Files\RabbitMQ Server\rabbitmq_server-3.6.10\sbin>rabbitmq-plugins.bat enable r
abbitmq_management --offline
The following plugins have been enabled:
  amqp_client
  cowlib
  cowboy
  rabbitmq_web_dispatch
  rabbitmq_management_agent
  rabbitmq_management
Offline change; changes will take effect at broker restart.

D:\Program Files\RabbitMQ Server\rabbitmq_server-3.6.10\sbin>rabbitmq-server.bat start

              RabbitMQ 3.6.10. Copyright (C) 2007-2017 Pivotal Software, Inc.
  ##  ##      Licensed under the MPL.  See http://www.rabbitmq.com/
  ##  ##
  ##########  Logs: C:/Users/BALASU~1/AppData/Roaming/RabbitMQ/log/RABBIT~1.LOG
  ######  ##        C:/Users/BALASU~1/AppData/Roaming/RabbitMQ/log/RABBIT~2.LOG
  ##########
              Starting broker...
 completed with 6 plugins.
```

Extended Standard

Type here to search

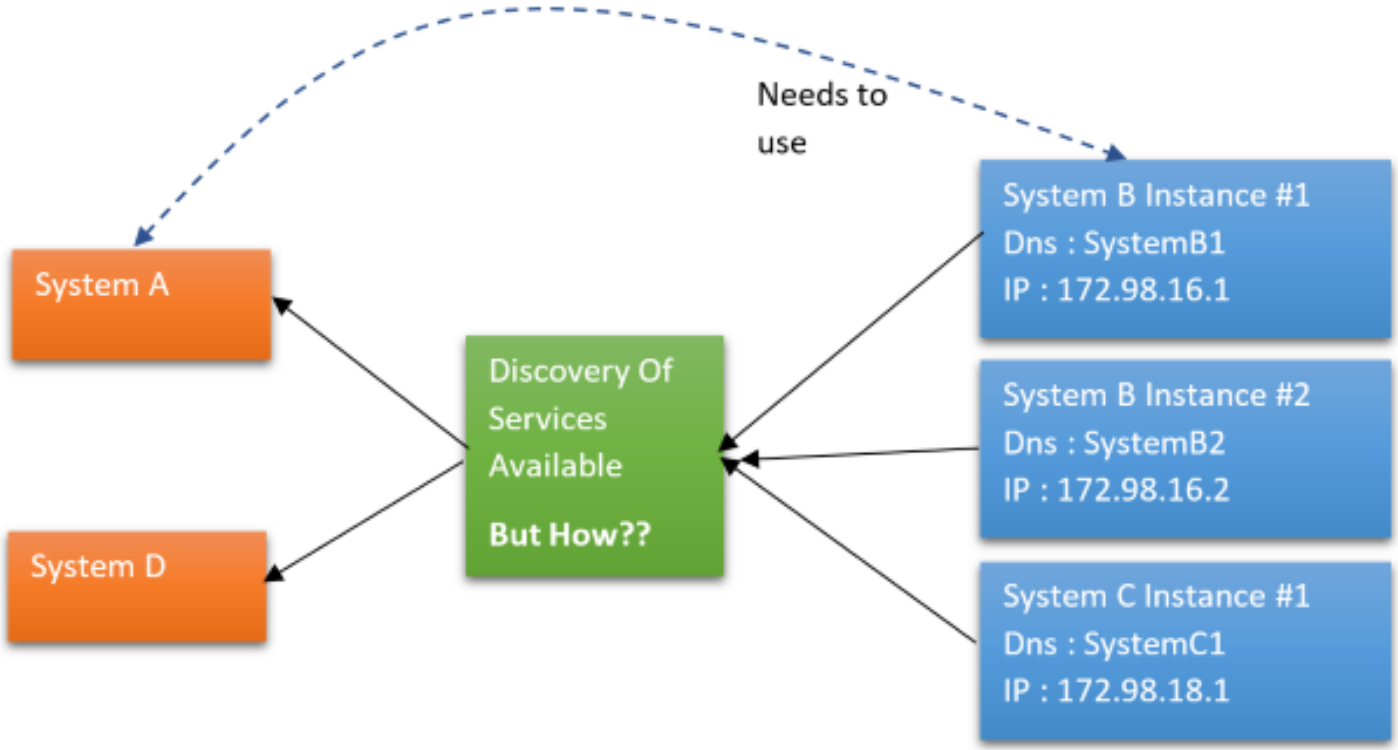
ENG 08:27 12/12/2018

Containerization

Administrator: RabbitMQ Command Prompt (sbin dir) - docker build -t dockerimageapp .

```
Microsoft Windows [Version 10.0.17134.471]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd G:\Local disk\dotnet_design_patterns\DockerContainerDemo\DockerCont
ainerDemo

C:\WINDOWS\system32>g:

G:\Local disk\dotnet_design_patterns\DockerContainerDemo\DockerContainerDemo>docker build
-t dockerimageapp .
Sending build context to Docker daemon  921.1kB
Step 1/7 : FROM microsoft/dotnet:onbuild
onbuild: Pulling from microsoft/dotnet
43c265008fae: Pull complete
af36d2c7a148: Pull complete
143e9d501644: Pull complete
882c59fec304: Pull complete
e8fd1a99b43e: Downloading  26.22MB/51.85MB
e4f74ce87b0c: Downloading  7.493MB/82.12MB
ee0b3487643d: Download complete
```

Discovery Pattern



Existing solutions

- Why do developers choose Zookeeper?

- High performance ,easy to generate node specific config
- Kafka support
- Java
- Supports extensive distributed IPC
- Spring Boot Support
- Supports DC/OS
- Embeddable In Java Service
- Used in Hadoop

Existing solutions

Consul

- Infrastructure
- Health checking
- Distributed key-value store
- Monitoring
- High-availability
- Web-UI
- Token-based acls
- Gossip clustering
- Dns server
- Docker integration

Existing solutions

- Redis/DataStore
- You could use Redis cache to be used by services to store meta data, and then a consumer could query the cache.
- However to make this resilient you really need some form of clustering, and some consensus/gossip to achieve consistency.
- This is fairly hard so most people just cut corners and make this a singleton, which is obviously a single point of failure.
- .

Existing solutions

- Kubernetes
- Kubernetes does a good job of "Disvovery" by way of Services/DNS addon/pods all of which can easily be load balanced.
- This is a great solution to run in containers (preferably in a cloud environment)
- Consul
- Consul (to my mind) is the only tool/framework that tackles "discovery" head on, and actually provides a rich tool that does that this job, and does it well, with little effort from the developer.

Configuring Consul in Local workstation

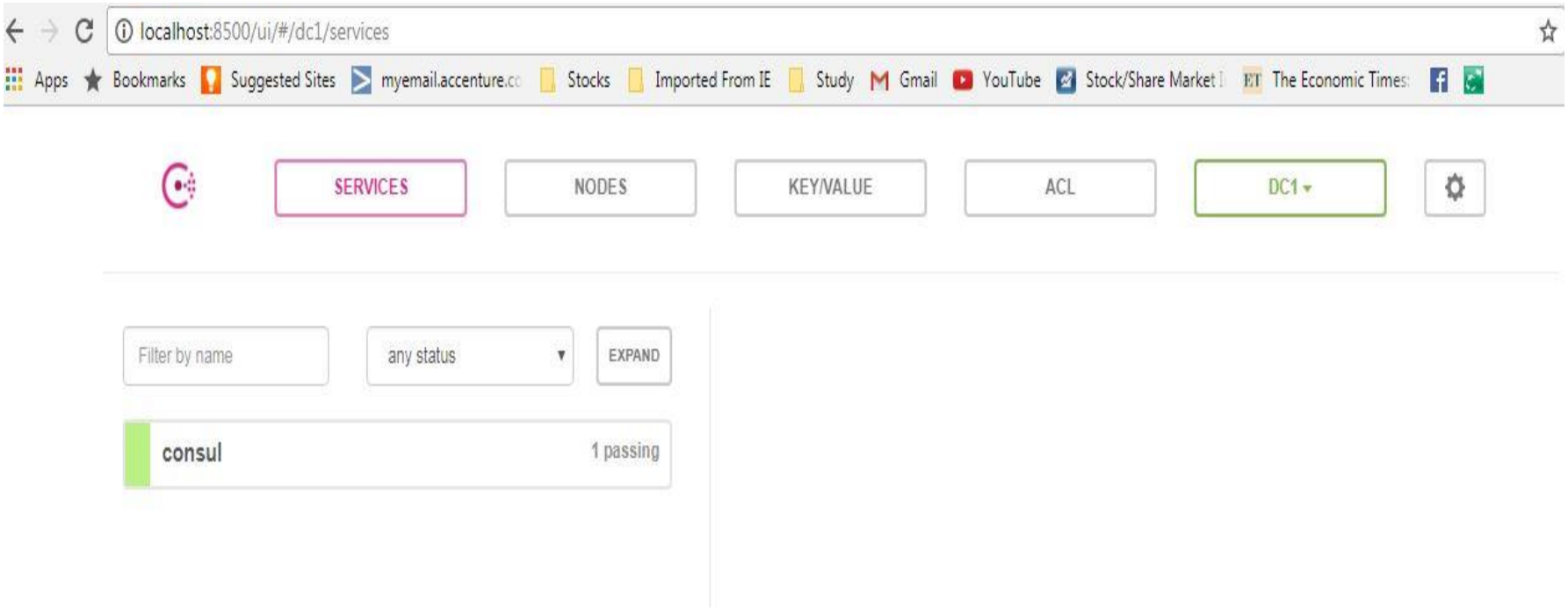
- Download from Consul portal. Choose particular package based on the operating System. Once downloaded the zip, we need to unzip it to desired place.
- Start Consul Agent in local workstation – The Zip file that we have unzipped, has only one exe file called consul.exe. We will start a command prompt here and use below command to start the agent.
- IPV4 is binding address.
- consul agent -server -bootstrap-expect=1 -data-dir=consul-data -ui -bind=192.168.99.1

Configuring Consul in Local workstation

```
C:\Windows\system32\cmd.exe - consul  agent -server -bootstrap-expect=1 -data-dir=consul-data -ui  -bind=192.168.6.1

P:\Study\installations\consul_0.9.0_windows_amd64>consul agent -server -bootstrap-expect=1 -data-dir=consul-data -ui  -bind=192.168.6.1
==> WARNING: BootstrapExpect Mode is specified as 1; this is the same as Bootstrap mode.
==> WARNING: Bootstrap mode enabled! Do not enable unless necessary
==> Starting Consul agent...
==> Consul agent running!
           Version: 'v0.9.0'
           Node ID: 'f8db8d09-ac67-e997-9306-aeb20e4ecd3e'
         Node name: 'Sajal-HP'
        Datacenter: 'dc1'
            Server: true (bootstrap: true)
       Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1, DNS: 8600)
      Cluster Addr: 192.168.6.1 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false

==> Log data will now stream in as it occurs:

    2017/07/20 13:31:42 [INFO] raft: Initial configuration (index=1): [{Suffrage:Voter ID:192.168.6.1:8300 Address:192.168.6.1:8300}]
    2017/07/20 13:31:42 [INFO] raft: Node at 192.168.6.1:8300 [Follower] entering Follower state (Leader: "")
    2017/07/20 13:31:42 [INFO] serf: EventMemberJoin: Sajal-HP.dc1 192.168.6.1
    2017/07/20 13:31:42 [WARN] serf: Failed to re-join any previously known node
    2017/07/20 13:31:42 [INFO] serf: EventMemberJoin: Sajal-HP 192.168.6.1
    2017/07/20 13:31:42 [INFO] consul: Handled member-join event for server "Sajal-HP.dc1" in area "wan"
    2017/07/20 13:31:42 [INFO] consul: Adding LAN server Sajal-HP (Addr: tcp/192.168.6.1:8300) (DC: dc1)
    2017/07/20 13:31:42 [WARN] serf: Failed to re-join any previously known node
    2017/07/20 13:31:42 [INFO] agent: Started DNS server 127.0.0.1:8600 (udp)
    2017/07/20 13:31:42 [INFO] agent: Started DNS server 127.0.0.1:8600 (tcp)
    2017/07/20 13:31:42 [INFO] agent: Started HTTP server on 127.0.0.1:8500
    2017/07/20 13:31:49 [ERR] agent: failed to sync remote state: No cluster leader
    2017/07/20 13:31:49 [ERR] http: Request GET /v1/catalog/services?wait=2s&index=169, error: No cluster leader from=127.0.0.1:53785
    2017/07/20 13:31:50 [ERR] http: Request GET /v1/catalog/services, error: No cluster leader from=127.0.0.1:53789
    2017/07/20 13:31:51 [ERR] http: Request GET /v1/catalog/services, error: No cluster leader from=127.0.0.1:53792
    2017/07/20 13:31:52 [WARN] raft: Heartbeat timeout from "" reached, starting election
    2017/07/20 13:31:52 [INFO] raft: Node at 192.168.6.1:8300 [Candidate] entering Candidate state in term 66
    2017/07/20 13:31:52 [INFO] raft: Election won. Tally: 1
    2017/07/20 13:31:52 [INFO] raft: Node at 192.168.6.1:8300 [Leader] entering Leader state
    2017/07/20 13:31:52 [INFO] consul: cluster leadership acquired
    2017/07/20 13:31:52 [INFO] consul: New leader elected: Sajal-HP
    2017/07/20 13:31:52 [WARN] agent: Check 'service:student-service-9099' is now critical
    2017/07/20 13:31:52 [WARN] agent: Check 'service:student-service-9097' is now critical
    2017/07/20 13:31:54 [INFO] agent: Synced check 'service:student-service-9097'
    2017/07/20 13:31:54 [INFO] agent: Synced check 'service:school-service-8098'
    2017/07/20 13:31:54 [INFO] agent: Synced check 'service:student-service-9099'
    2017/07/20 13:31:54 [INFO] agent: Synced check 'service:student-service-9098'
```

Configuring Consul in Local workstation

- **Test whether Consul Server is running** – Consul runs on default port and once agent started successfully, browse http://localhost:8500/uiand you should see a console screen like –



Consul Web

- Registered services
- The cluster nodes
- The Key/Values for the KeyValueStore
- Any ACL (Access Control List) values you have setup

Consul

- Install-Package consul

PWA

- PWAs are just websites, enhanced using modern design and web technologies (Responsive Design, Touch Friendly, Service Workers, Fetch networking, Cache API, Push notifications, Web App Manifest) to provide a more native app-like experience.

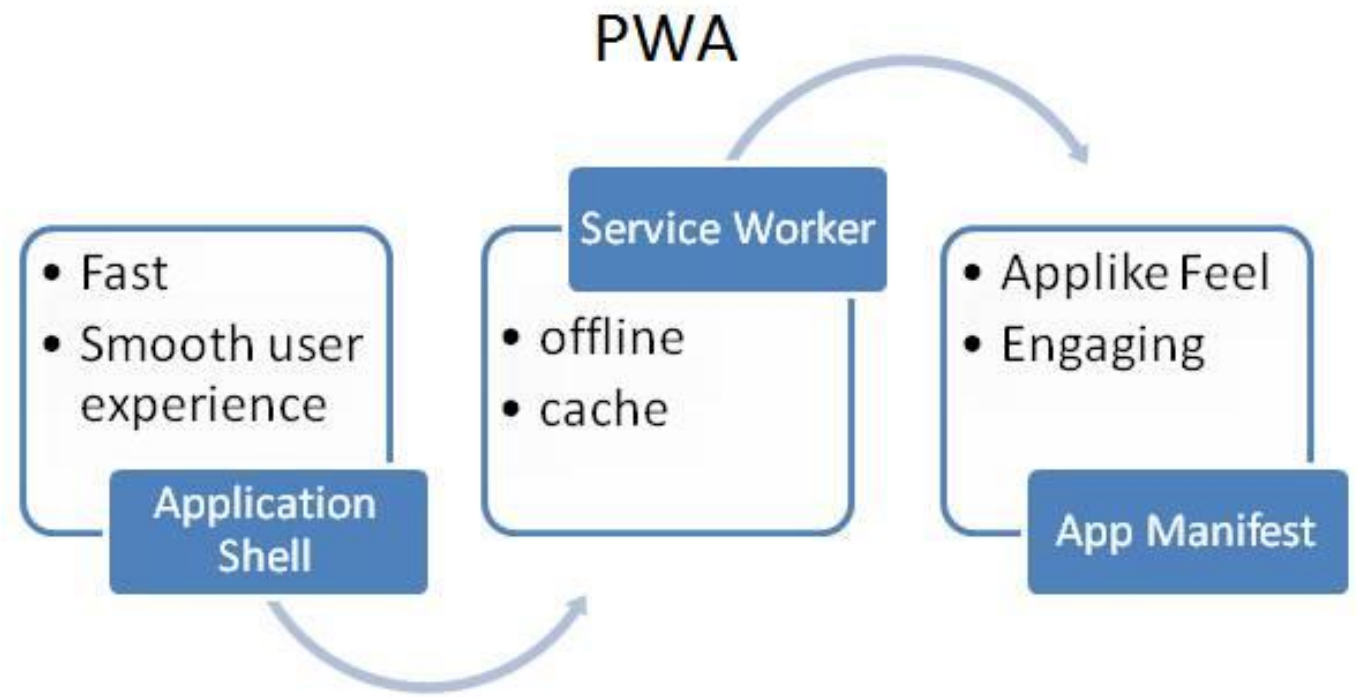
PWA

- **Manifest, Service Worker, Secure:**
- To create a PWA the website will need: a manifest, service workers, and be on Https:
- Standard manifest file: The site requires a manifest file to define the features and behavior of the PWA. The file follows the W3C standard. This includes everything from images, to language, or the start page of your web app.
- Service workers: The Progressive Web App should have a mechanism (e.g. through a service worker) to help control traffic when the network isn’t there or isn’t reliable. The app should be able to work independent of network.
- Secure: A secure connection (HTTPS) over your site makes sure all traffic is as safe as a native app. A secure endpoint also allows the service worker to securely take action on the behalf of your app.
- .

PWA

- **Manifest, Service Worker, Secure:**
- The manifest defines some details for your site if it's downloaded, which includes custom splash-screens, icon-standards and more.
- It also prompts the user to install it as a PWA, via an install-banner, if they visit the site frequently on a phone.
- This is one of the core properties of a PWA (Progressive Web App) since it gives the immersive App part.

PWA



Progressive Web Apps are applications that use modern web capabilities to provide a mobile application-like experience to users via the web.

Progressive Web Apps are

- *Reliable*
  Service Workers enable PWA to load instantly, regardless of the network state. It never shows that awful dinosaur screen, even if there is no network.

- *Fast*
  Application Shell makes it faster and provides a smooth user experience.

- *Engaging*
  Manifest makes more engaging web apps. Manifest file makes PWA installable and live on the user's home screen. We can even make the users re-engaged with the help of push notifications.

- *Progressive*
  Works for every user, regardless of browser.

- *Safe*
  PWA Served via HTTPS to prevent snooping and ensure the security of content.

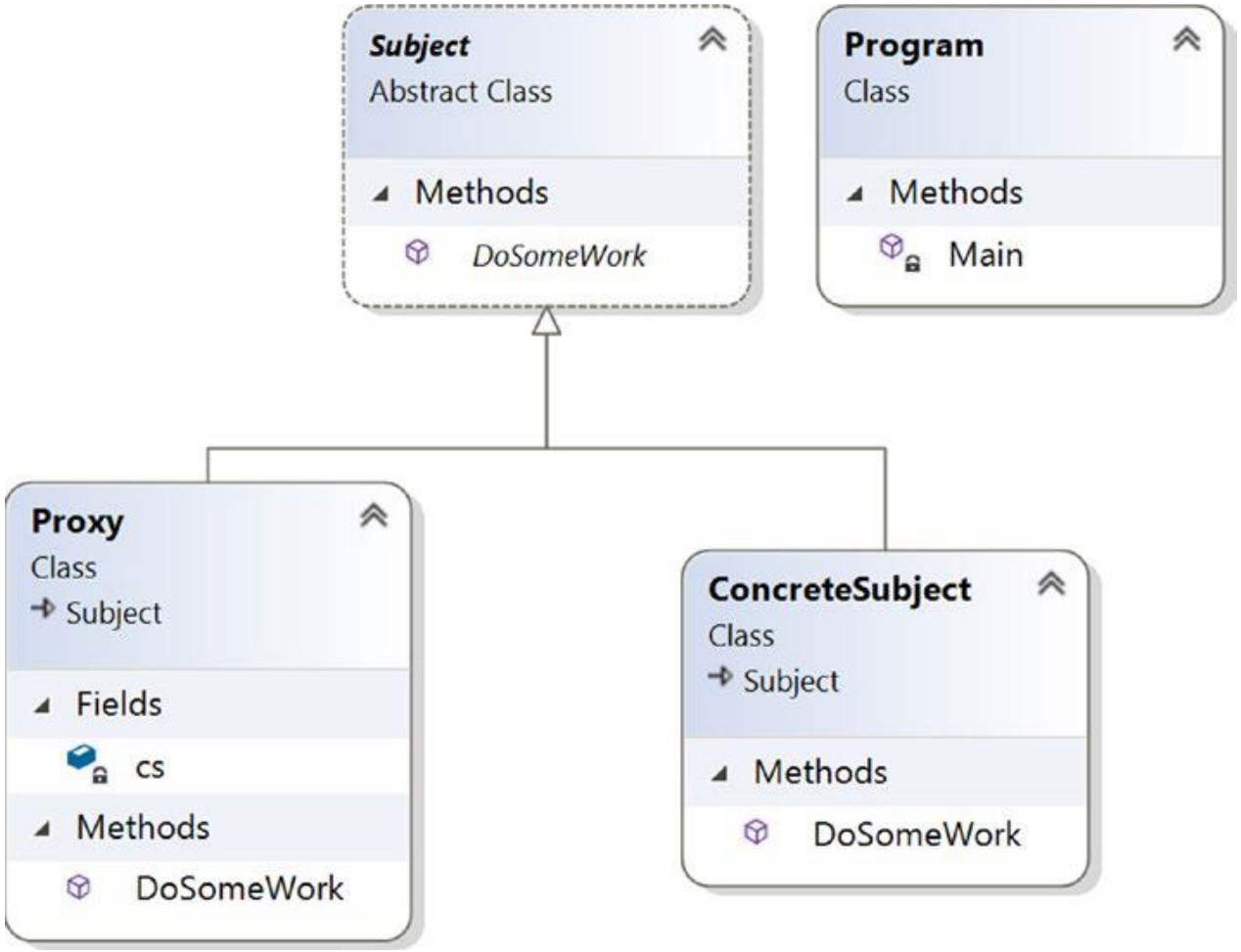
- *Linkable*
  It can be easily shared via URL.

Proxy Pattern

- Provide a surrogate or placeholder for another object to control access to it.
- Concept
- A proxy is basically a substitute for an intended object.
- When a client deals with a proxy object, it thinks that it is dealing with the actual object.
- You need to support this kind of design because dealing with an original object is not always possible.
- This is because of many factors such as security issues, for example.
- So, in this pattern, you may want to use a class that can perform as an interface to something else.

Proxy Pattern

- An ATM implementation will hold proxy objects for bank information that exists on a remote server.
- In the real programming world, creating multiple instances of a complex object (a heavy object) is costly in general.
- So, whenever you can, you should create multiple proxy objects that can point to the original object.
- This mechanism can also help you to save the computer/system memory.



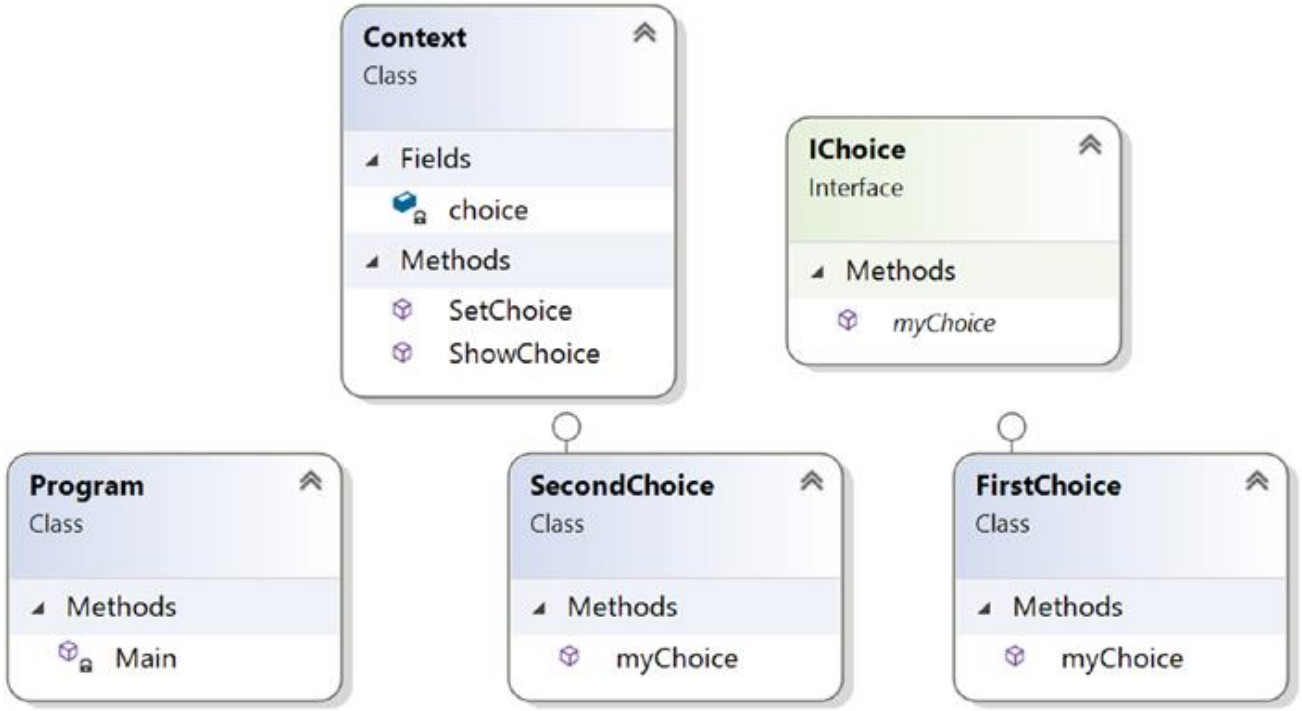
Strategy (Policy) Pattern

- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- Concept
- You can select the behavior of an algorithm dynamically at runtime.

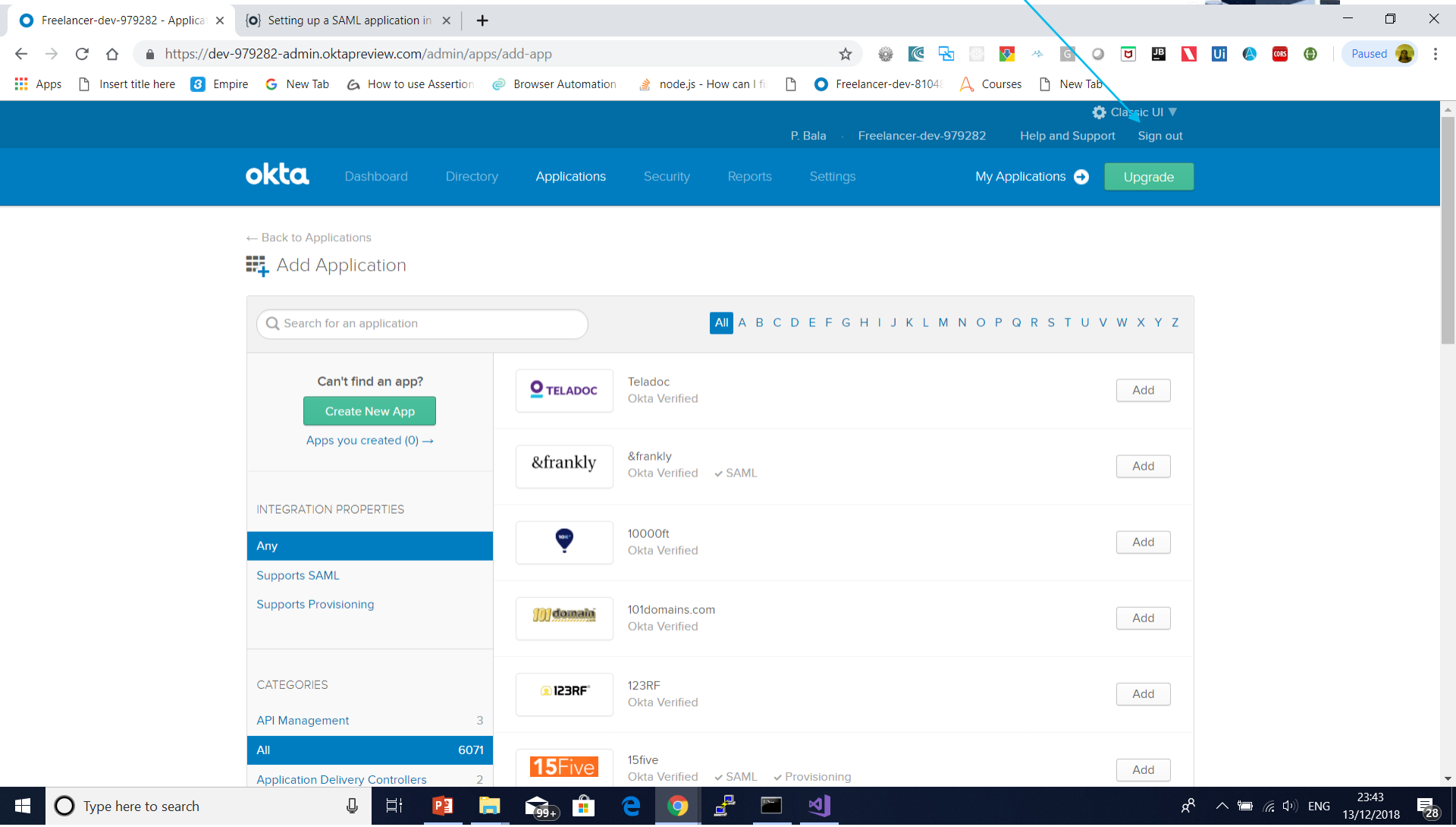
Strategy (Policy) Pattern

- Suppose you have a backup memory slot.
- If your primary memory is full and you need to store more data, you can store it in the backup memory slot.
- If you do not have this backup memory slot and you try to store additional data into your primary memory (when it is full), then that data will be discarded, you will receive exceptions, or you will encounter some peculiar behavior (based on the architecture of the program).
- So, a runtime check is necessary before storing the data, and then you can proceed

Moq Framework



Switch to classic ui



Freelancer-dev-979282 - Applica | Setting up a SAML application in

https://dev-979282-admin.oktapreview.com/admin/apps/add-app

Apps | Insert title here | Empire | New Tab | How to use Assertion | Browser Automation | node.js - How can I fi | Freelancer-dev-8104 | Courses | New Tab

Paused

Classic UI

P. Bala · Freelancer-dev-979282 Help and Support Sign out

okta Dashboard Directory Applications Security Reports Settings My Applications Upgrade

← Back to Applica

Add App

Search for a

Can't

Cre

Apps yo

INTEGRATION PR

Any

Supports SAML

Supports Provisio

CATEGORIES

API Management 3

All 6071

Application Delivery Controllers 2

V W X Y Z

Add

Add

Add

Add

Add

Add

123RF

Okta Verified

15five

Okta Verified ✓SAML ✓Provisioning

Create a New Application Integration

Platform

Web

Sign on method

Secure Web Authentication (SWA)

Uses credentials to sign in. This integration works with most apps.

SAML 2.0

Uses the SAML protocol to log users into the app. This is a better option than SWA, if the app supports it.

OpenID Connect

Uses the OpenID Connect protocol to log users into an app you've built.

Create Cancel

Type here to search

ENG 23:44 13/12/2018

Preview SAML

```
<?xml version="1.0" encoding="UTF-8"?>
<saml2:Assertion
   xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion" ID="id845221844524673663710936 9" IssueInstant="2018-12-13T18:19:26.791Z" Version="2.0">
   <saml2:Issuer Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity">http://www.okta.com/Issuer</saml2:Issuer>
   <saml2:Subject>
      <saml2:NameID Format="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified">userName</saml2:NameID>
      <saml2:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
         <saml2:SubjectConfirmationData NotOnOrAfter="2018-12-13T18:24:27.004Z" Recipient="http://example.com/saml/sso/example-okta-com"/>
      </saml2:SubjectConfirmation>
   </saml2:Subject>
   <saml2:Conditions NotBefore="2018-12-13T18:14:27.004Z" NotOnOrAfter="2018-12-13T18:24:27.004Z">
      <saml2:AudienceRestriction>
         <saml2:Audience>http://example.com/saml/sso/example-okta-com</saml2:Audience>
      </saml2:AudienceRestriction>
   </saml2:Conditions>
   <saml2:AuthnStatement AuthnInstant="2018-12-13T18:19:26.791Z">
      <saml2:AuthnContext>
         <saml2:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport</saml2:AuthnContextClassRef>
      </saml2:AuthnContext>
   </saml2:AuthnStatement>
</saml2:Assertion>
```

SAML Authentication Steps

https://developer.okta.com/standards/SAML/setting_up_a_saml_application_in_okta

Client Id 0oaibarqtmkLjQy0o0h7

Client Secret KM48VeTKtPW0W2_Vh5iOy95Ej6aHMuDWvjtXBGVK

Template method pattern

- GoF Definition
- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- The Template Method pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure

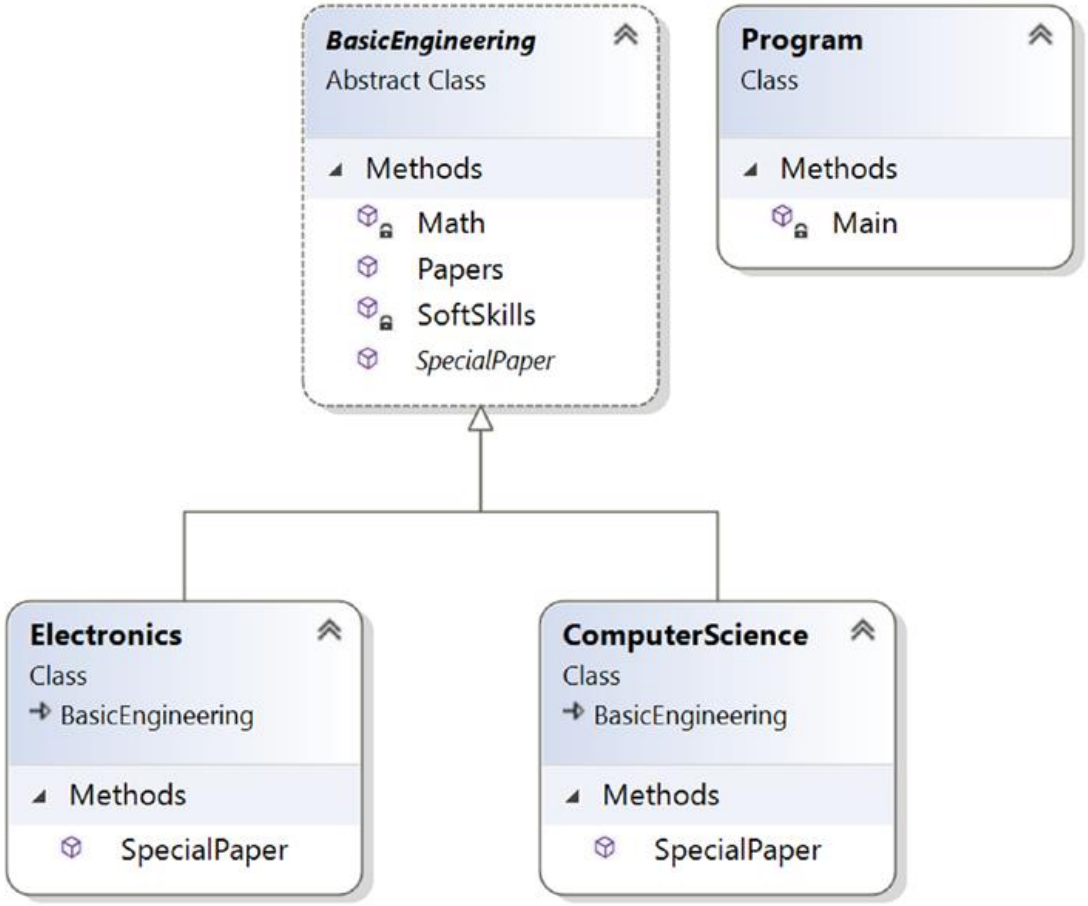
Template method pattern

- With the Template Method pattern, you define the minimum or essential structure of an algorithm.
- Then you defer some responsibility to the subclasses. The key idea is that you can redefine certain steps of an algorithm, but those changes will not impact the core architecture.

Template method pattern

- The Template Method pattern makes sense when you want to avoid duplicate code in your application but allow subclasses to change some specific details of the base class workflow to bring varying behavior to the application.

Template Pattern



Decorator

- GoF Definition
- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator

- This pattern promotes the concept that your class should be closed for modification but open for extension.
- In other words, you can add a functionality without disturbing the existing functionalities.
- The concept is useful when you want to add some special functionality to a specific object instead of the whole class.
- This pattern prefers object composition over inheritance.
- Once you master this technique, you can add new responsibilities to an object without affecting the underlying classes

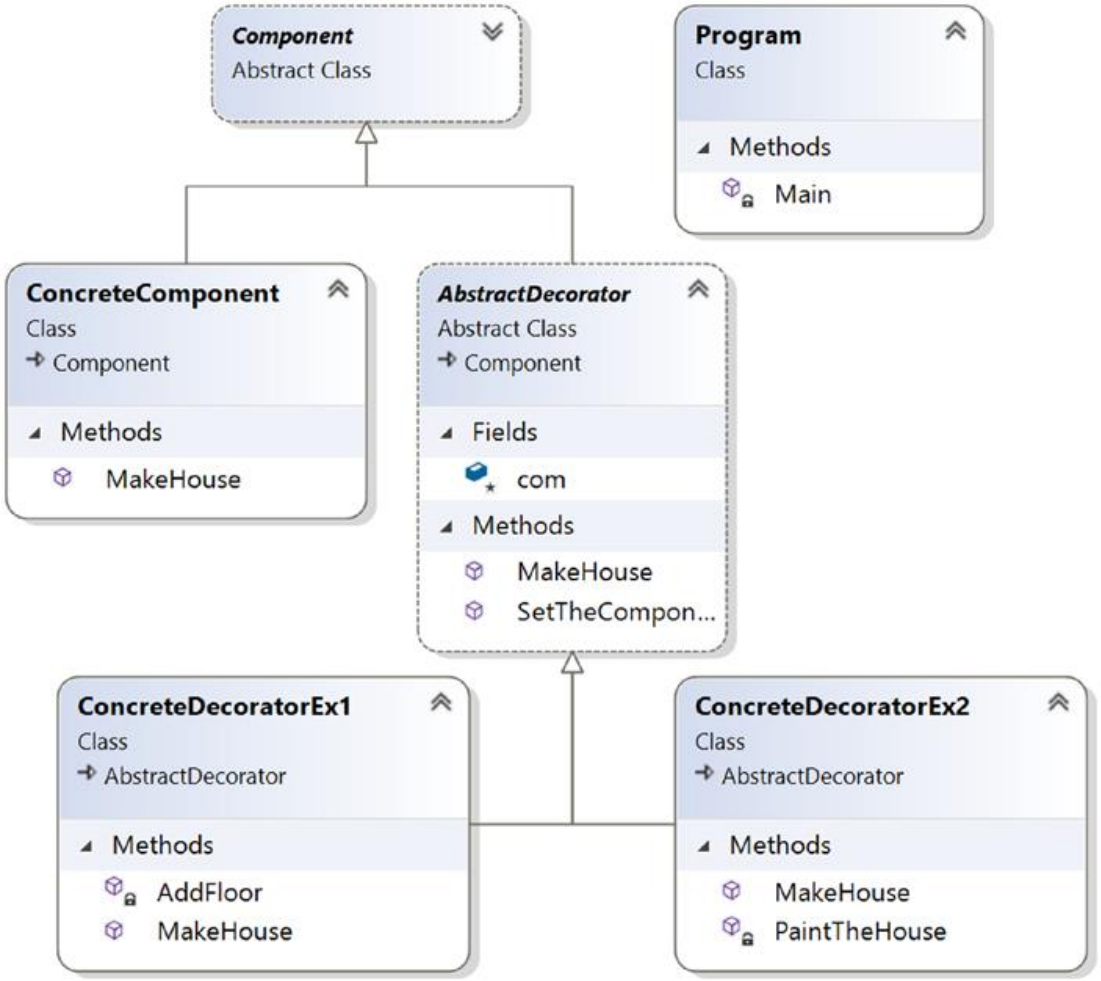
Decorator

- Suppose you own a single-story house and you decide to build a second floor on top of it.
- Obviously, you may not want to change the architecture of the ground floor.
- But you may want to change the design of the architecture for the newly added floor without affecting the existing architecture.

Decorator

- Suppose in a GUI-based toolkit you want to add some border properties.
- You could do this with inheritance, but that cannot be treated as an ultimate solution because you do not have absolute control over everything from the beginning.
- So, this technique is static in nature.
- Decorators offer a flexible approach. They promote the concept of dynamic choices.
- For example, you can surround the component in another object. The enclosing object is termed a *decorator*, and it must conform to the interface of the component that it decorates.
- It will forward the requests to the component, and it can perform additional operations before or after those requests. In fact, you can add an unlimited number of responsibilities with this concept.

Decorator



State Pattern

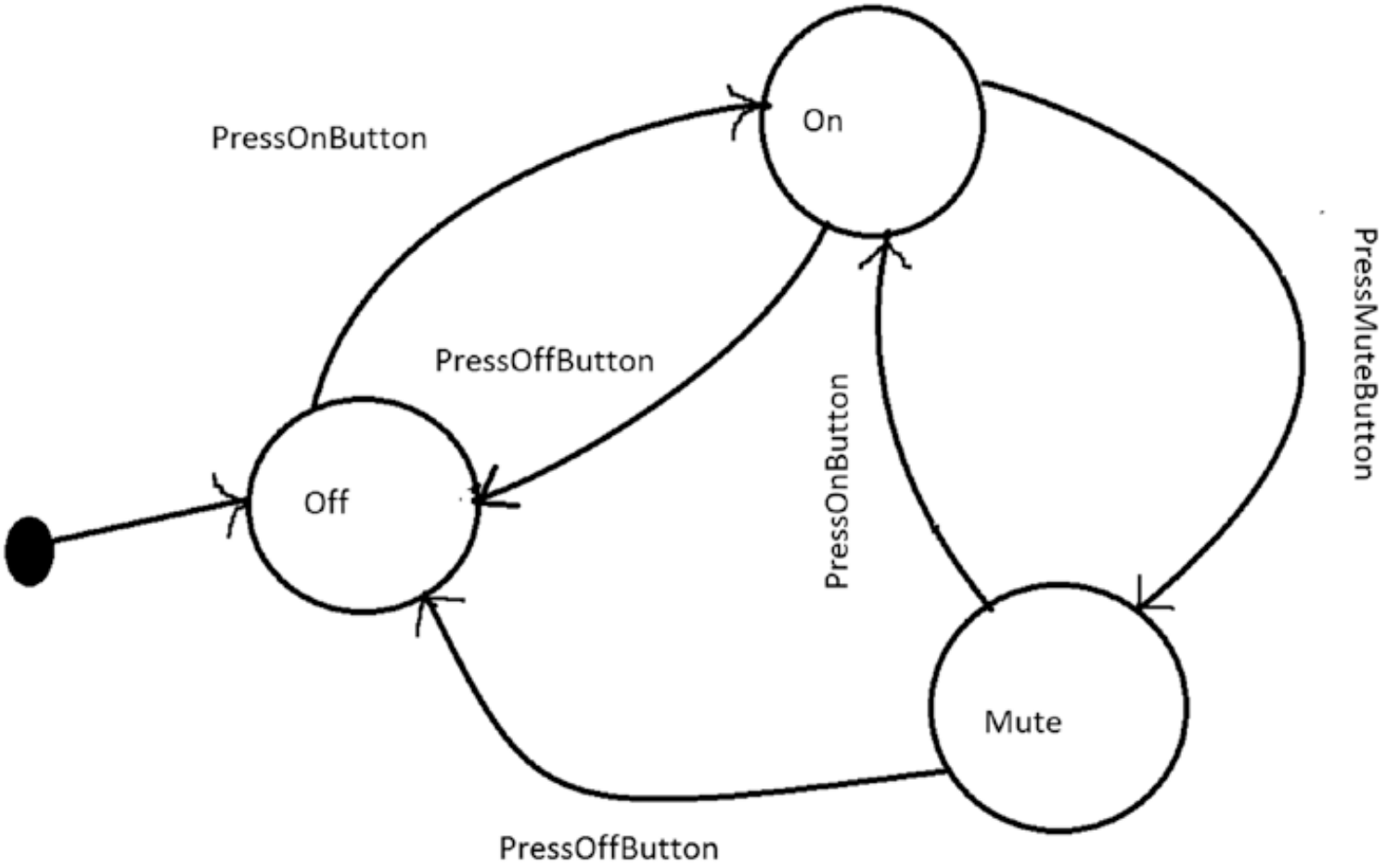
- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

- The concept is best described by the following examples.
- Consider the scenario of a network connection, say a TCP connection.
- An object can be in various states; for example, a connection might already be established, a connection might be closed, or the object has already started listening through the connection.
- When this connection receives a request from other objects, it responds as per its present state.

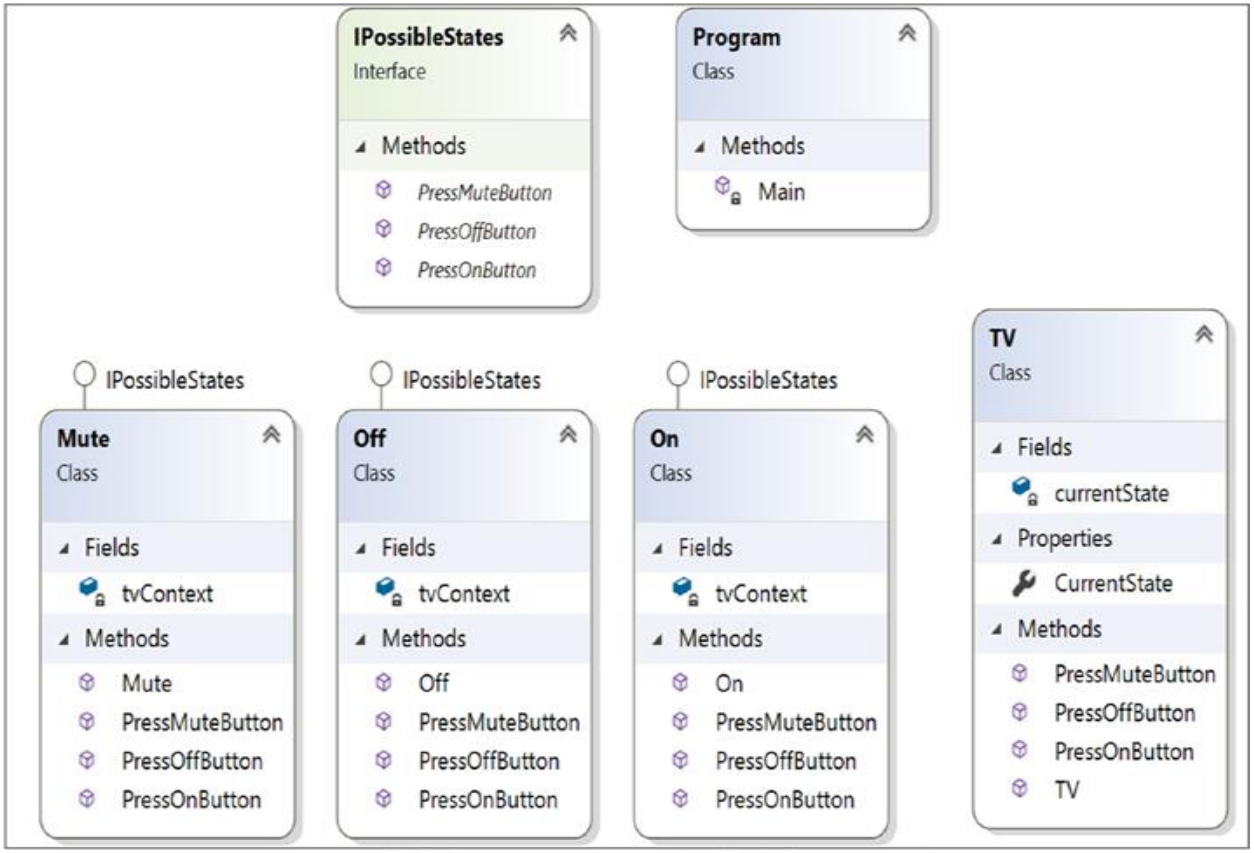
State Pattern

- Suppose you have a job-processing system that can process a certain number of jobs at a time.
- When a new job appears, either the system processes the job or it signals that the system is busy with the maximum number of jobs that it can process at a time.
- In other words, the system will send a busy signal if it gets another job-processing request when its total number of job-processing capabilities has been reached.

State Pattern



State Pattern



Visitor Pattern

- GoF Definition
- Represent an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor Pattern

- Concept
- With this pattern, you separate an algorithm from an object structure.
- So, you can add new operations without modifying the existing architecture.
- This pattern supports the open/close principle (which says extension is allowed but modification is disallowed for entities such as class, function, modules, and so on)..

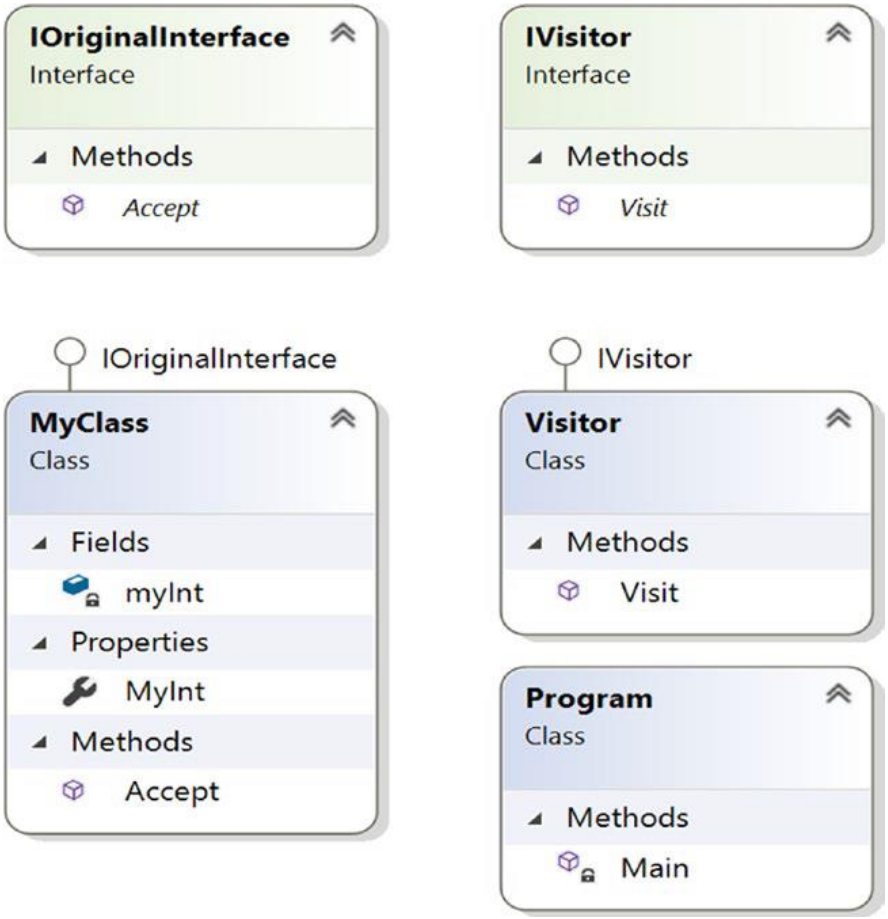
Visitor Pattern

- Think of a taxi-booking scenario.
- When the taxi arrives at your door and you enter the vehicle, the visiting taxi takes control of the transportation.
- It can take you to your destination through a new route that you are not familiar with, and in the worst case, it can alter the destination.

Visitor Pattern

- This pattern is useful when public APIs need to support *plug-in* operations.
- Clients can then perform their intended operations on a class (with the visiting class) without modifying the source.

Visitor Pattern



Questions

Module Summary

- Desing patterns
- EE Patterns
- How to use
- DSL
- Creational Pattern