1. Strategy pattern + Template Method pattern

Strategy

The Strategy pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable. It allows a client to choose an appropriate algorithm from a family of algorithms at runtime, without altering the code that uses the algorithm. The key idea is to separate the behavior of a class (or a set of related classes) from the class itself, making it more flexible and easily extensible.

https://refactoring.guru/design-patterns/strategy

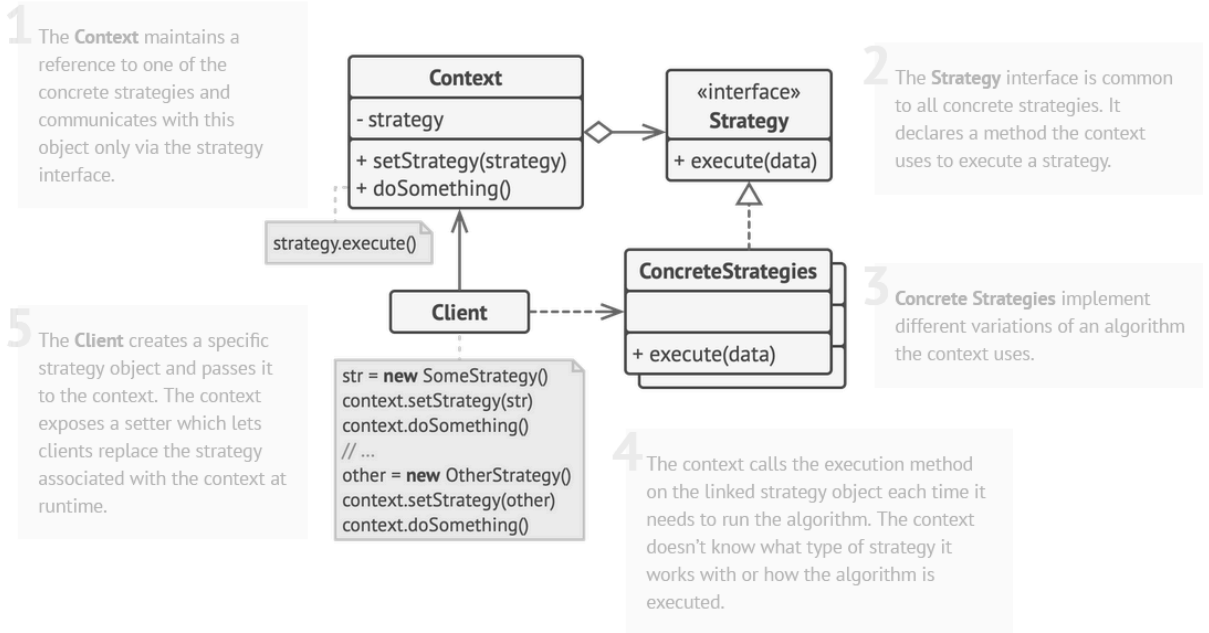
Characteristics

Encapsulation: The Strategy pattern promotes encapsulation by encapsulating the algorithm within a separate strategy class. This isolation helps to hide the implementation details from the context, making it easier to manage and understand.

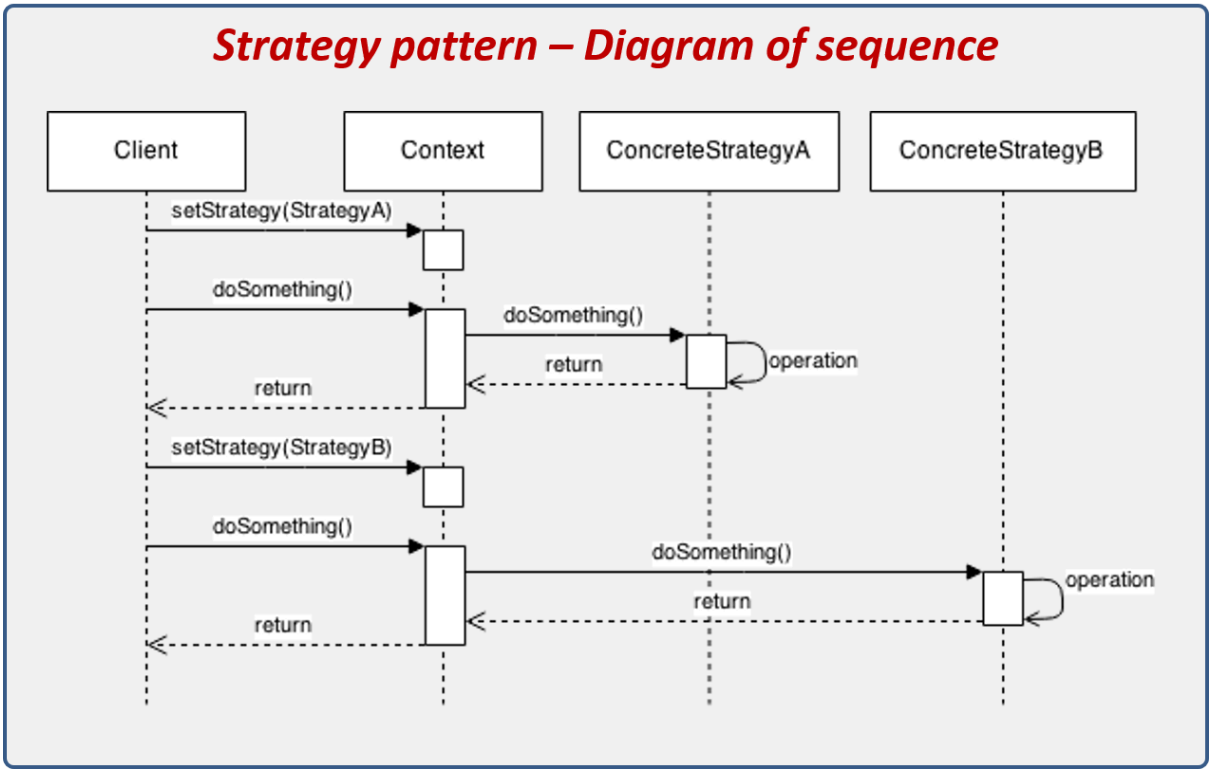
Flexibility and Extensibility: The Strategy pattern allows for easy switching between different algorithms **at runtime**. It provides a mechanism for adding or modifying algorithms without modifying the client code. This flexibility is particularly useful when dealing with a family of algorithms or when requirements change over time.

Open/Closed Principle: The Strategy pattern follows the Open/Closed Principle, which states that a class should be open for extension but closed for modification. You can introduce new strategies without modifying existing code, promoting code stability and reducing the risk of introducing bugs.

Class Diagram



Sequence Diagram

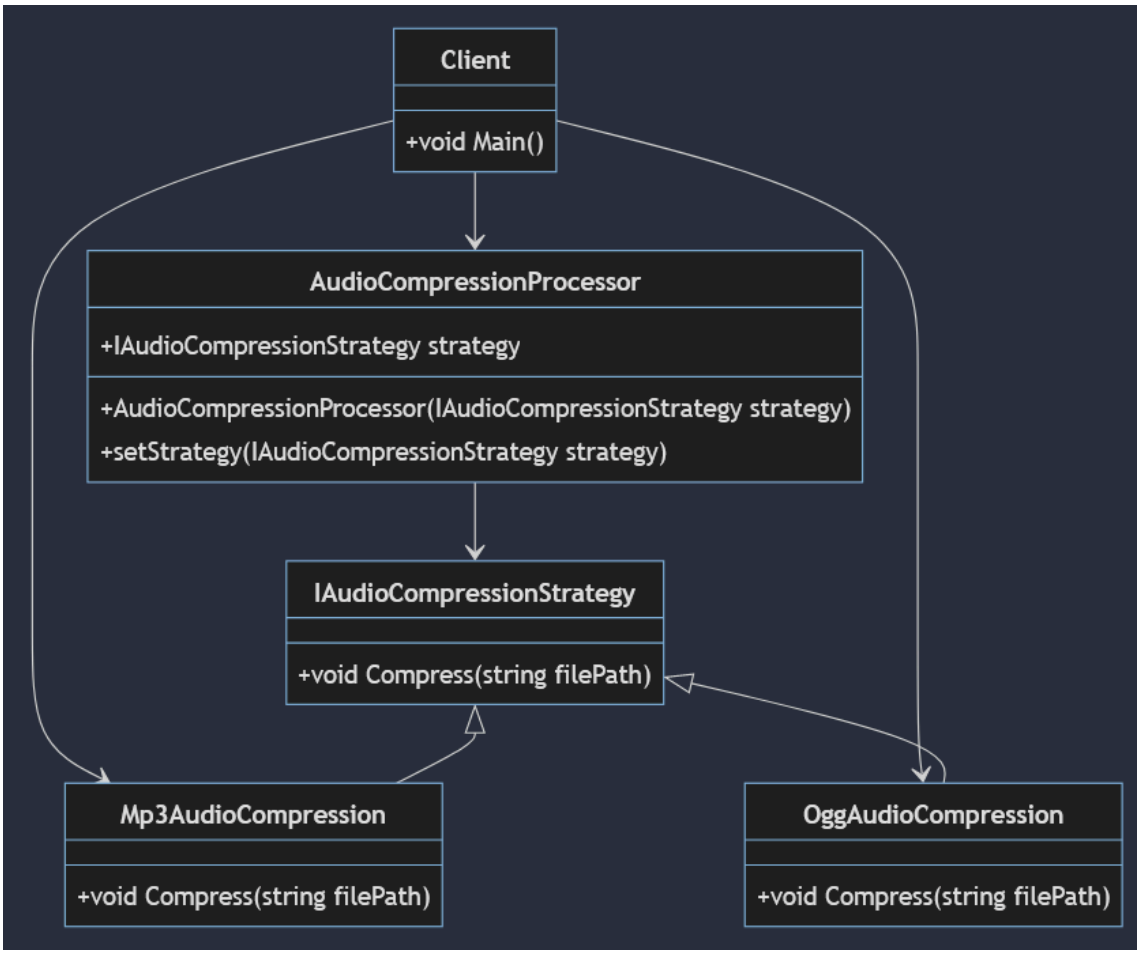


1. The client creates a new context and sets the StrategyA.
1. The client executes the doSomethingoperation.
2. Context hands over this responsibility to ConcreteStrategyA.
3. ConcreteStrategyA performs the operation and returns the result.
4. Context takes this result and hands it over to the client.
5. The client changes the Context strategy at runtime.
6. The client executes the doSomethingoperation again.
7. Context hands over this responsibility to ConcreteStrategyB.
8. ConcreteStategyB performs the operation and returns the result.
9. Context takes this result and hands it over to the client.

Example

There is audio compression to be implemented. We create IAudioCompressionStrategy interface which is implemented by Ogg and Mp3 compression classes. AudioCompressionProcessor is created with strategy and we can setStrategy differently if we want to. Client depends on the Processor and concrete strategies he wants to use.

Diagram



Code

```
namespace StrategyAudio.Strategy
{
    interface IAudioCompressionStrategy
    {
        void Compress(string filePath);
    }
}
```

```
namespace StrategyAudio.Strategy
{
    class Mp3AudioCompression : IAudioCompressionStrategy
    {
        public void Compress(string filePath)
        {
           Console.WriteLine($"Mp3 audio compression of this ->
{filePath}");
        }
    }
}

namespace StrategyAudio.Strategy
{
    class OggAudioCompression : IAudioCompressionStrategy
    {
        public void Compress(string filePath)
        {
            Console.WriteLine($"Ogg audio compression of this -> {filePath}");

        }
    }
}

using StrategyAudio.Strategy;

namespace StrategyAudio
{
    class AudioCompressionProcessor
    {
        IAudioCompressionStrategy strategy;

        public AudioCompressionProcessor(IAudioCompressionStrategy strategy) {

            this.strategy = strategy;
        }

        public void SetStrategy(IAudioCompressionStrategy strategy) {  this.strategy
= strategy; }


        public void CompressAudio(string filePath)
        {
            strategy.Compress(filePath);
        }
    }
}
using StrategyAudio;
```

```
using StrategyAudio.Strategy;

namespace Program
{
    class Client
    {
        public static void Main()
        {
            IAudioCompressionStrategy strategyMP3 = new Mp3AudioCompression();
            IAudioCompressionStrategy strategyOGG = new OggAudioCompression();

            AudioCompressionProcessor contextProcessor = new
AudioCompressionProcessor(strategyMP3);
            contextProcessor.CompressAudio("./files/audio1.wav");

            contextProcessor.SetStrategy(strategyOGG);
            contextProcessor.CompressAudio("./files/audio2.wav");
        }
    }
}
```

The given code follows SOLID principles:

1. **Single Responsibility Principle (SRP):** Each class has a single responsibility, e.g., `IAudioCompressionStrategy` for compression methods.
2. **Open/Closed Principle (OCP):** `AudioCompressionProcessor` is open for extension (via strategy injection) but closed for modification.
3. **Liskov Substitution Principle (LSP):** Strategies (`Mp3AudioCompression` and `OggAudioCompression`) can be substituted interchangeably.
4. **Interface Segregation Principle (ISP):** `IAudioCompressionStrategy` has only one method, and each strategy implements only what is needed.
5. **Dependency Inversion Principle (DIP):** High-level modules depend on abstractions (`IAudioCompressionStrategy`), allowing flexibility in choosing strategies.

Template Method

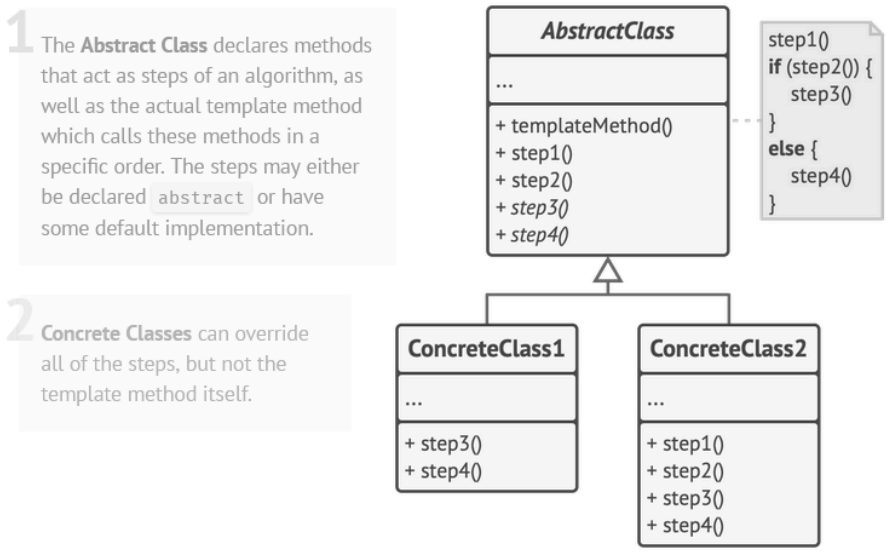
The Template Method pattern is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure. It enables the definition of a template for an algorithm while allowing variations in the implementation of certain steps by subclasses.
https://refactoring.guru/design-patterns/template-method

Characteristics

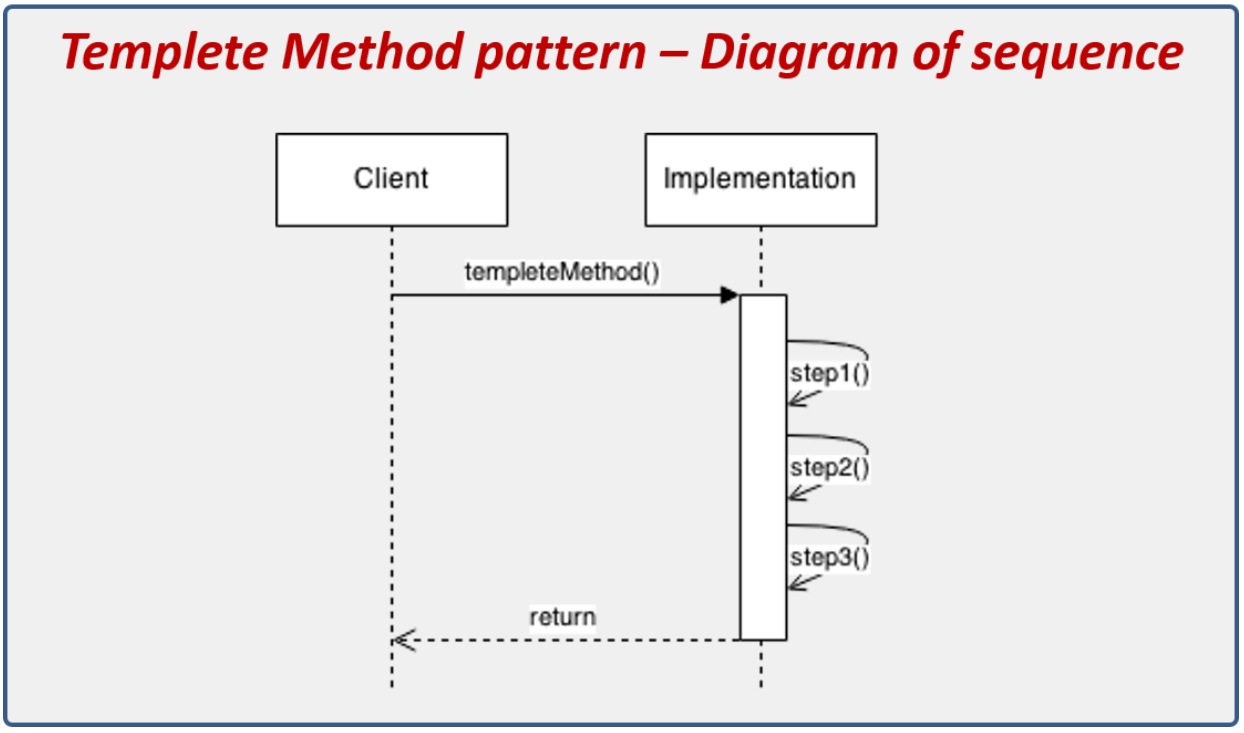
1. **Single Responsibility Principle (SRP):**
    - **Template Method Context:** The template method itself typically adheres to the SRP, as it is responsible for orchestrating the steps of an algorithm.
    - **Concrete Classes:** Concrete classes implementing specific steps are responsible for their respective tasks, aligning with the SRP.
2. **Open/Closed Principle (OCP):**

- **Template Method Context:** The Template Method pattern supports the OCP by providing an extensible framework. New functionality can be added by creating new subclasses without modifying the existing code in the superclass.
- **Concrete Classes:** Concrete classes can be added or extended without changing the existing template method.

Class Diagram



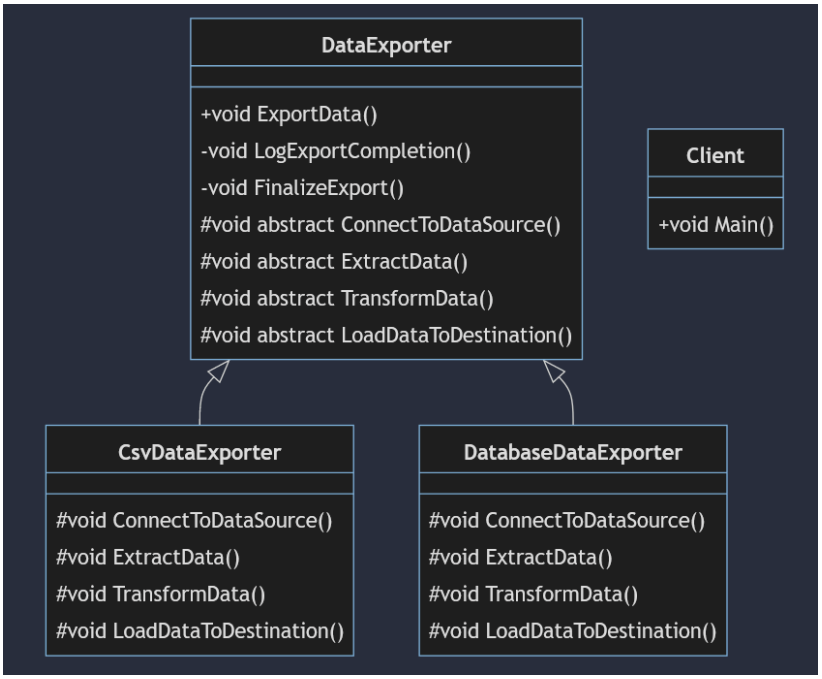
Sequence diagram



Example

There is a DataExporter class that has multiple abstract methods that will be overwritten by the subclasses. DataExporter has method ExportData() that invokes other private/public methods that are steps involving exporting data. Some methods are abstract so that they can be implemented by subclasses to make it function in a special way.

Diagram



Code

```
namespace DataExporterTemplateMethod
{
   public  abstract class DataExporter
    {
        public void ExportData()
        {
            ConnectToDataSource();
            ExtractData();
            TransformData();
            LoadDataToDestination();
            LogExportCompletion();
            FinalizeExport();
            Console.WriteLine("ExportData(): Data export has been completed.");
        }

        // Common methods implemented in the superclass
        private void LogExportCompletion()
        {
            Console.WriteLine("Common methods: Export process completed
successfully.");
        }

        private void FinalizeExport()
        {
            Console.WriteLine("Common methods: Finalizing export process");
            // Any additional steps needed for finalization
        }

        // Abstract methods implemented by subclasses
```

```
        protected abstract void ConnectToDataSource();
        protected abstract void ExtractData();
        protected abstract void TransformData();
        protected abstract void LoadDataToDestination();
    }
}

namespace DataExporterTemplateMethod
{
    public class DatabaseDataExporter : DataExporter
    {
        protected override void ConnectToDataSource()
        {
            Console.WriteLine("DatabaseDataExporter: Connecting to the data
source...");
        }

        protected override void ExtractData()
        {
            Console.WriteLine("DatabaseDataExporter: Extracting the data...");
        }

        protected override void LoadDataToDestination()
        {
            Console.WriteLine("DatabaseDataExporter: Loading data to the
destination...");
        }

        protected override void TransformData()
        {
            Console.WriteLine("DatabaseDataExporter: Transforming data...");
        }
    }
}


namespace DataExporterTemplateMethod
{
    public class CsvDataExporter : DataExporter
    {
        protected override void ConnectToDataSource()
        {
            Console.WriteLine("CsvDataExporter: Connecting to the data source...");
        }

        protected override void ExtractData()
        {
            Console.WriteLine("CsvDataExporter: Extracting the data...");
```

```
        }

        protected override void LoadDataToDestination()
        {
            Console.WriteLine("CsvDataExporter: Loading data to the destination...");
        }

        protected override void TransformData()
        {
            Console.WriteLine("CsvDataExporter: Transforming data...");
        }
    }
}

namespace DataExporterTemplateMethod
{
    class Client
    {
        public static void Main(string[] args)
        {
            DataExporter csvExporter = new CsvDataExporter();
            DataExporter dbExporter = new DatabaseDataExporter();

            Console.WriteLine("Launching ExportData() from CsvDataExporter:");
            csvExporter.ExportData();

            Console.WriteLine("\nLaunching ExportData() from DatabaseDataExporter:");
            dbExporter.ExportData();
        }
    }
}
```

- **Single Responsibility Principle (SRP):** `DataExporter` focuses on the template method for data export.
- **Open/Closed Principle (OCP):** The template method allows extension without modifying existing code.
- **Liskov Substitution Principle (LSP):** Subclasses (`CsvDataExporter` and `DatabaseDataExporter`) can be used interchangeably.
- **Interface Segregation Principle (ISP):** While not explicit, the template method enforces a form of interface segregation by providing a clear structure for required methods.
- **Dependency Inversion Principle (DIP):** High-level module (`DataExporter`) depends on abstractions (abstract methods), allowing flexibility in choosing specific steps.

2. Observer pattern

Observer

The Observer pattern is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents (observers) are notified and updated automatically. This pattern is also known as the "Publish-Subscribe" pattern.

Characteristics

There are two types of implementations of this pattern

Push Model:

- In the push model, the subject sends detailed information about the change to its observers when a state change occurs.
- Observers receive the actual data they need. **Observers get their data pushed.**

Pull Model:

- In the pull model, the subject sends a general notification to its observers about the change.
- Observers must then request the specific data they need from the subject. **Observers Pull the data after the“Update” function.**

Key components of the Observer pattern:

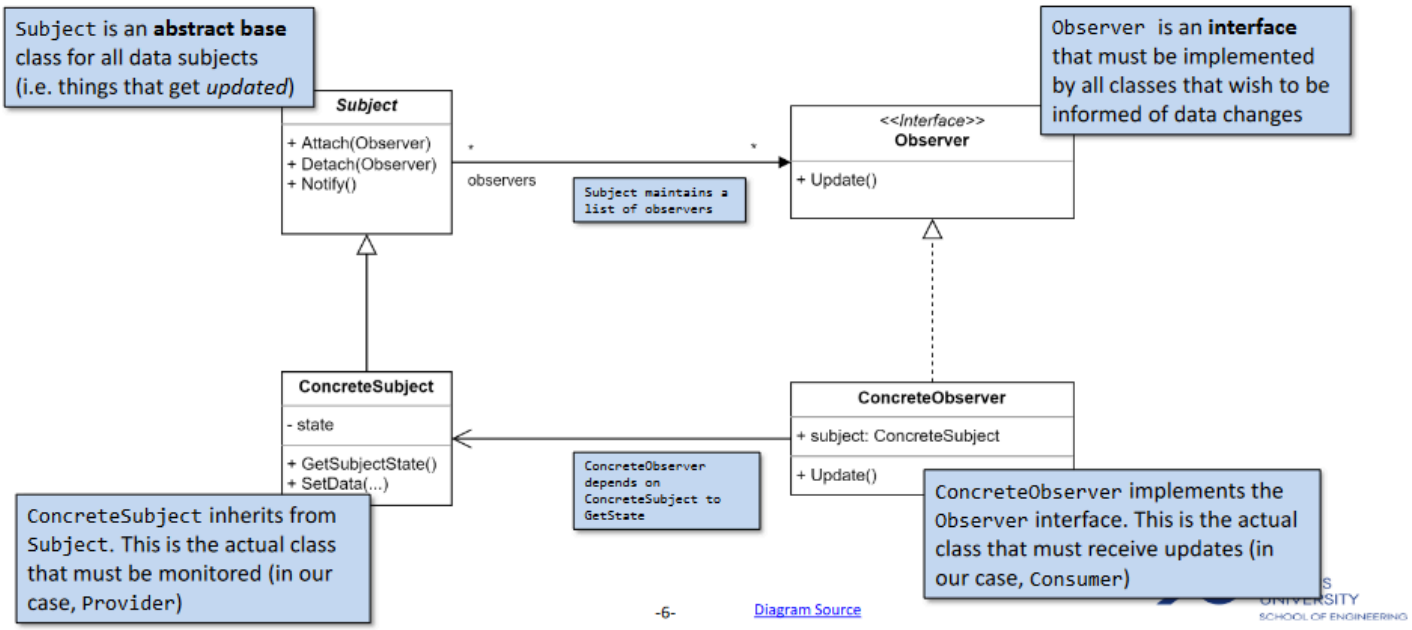
- **Subject (Observable):** This is the object that maintains a list of observers and notifies them of state changes. The subject can be any object that needs to be monitored for changes.

- **Observer:** This is the interface or abstract class that defines the contract for objects that should be notified of changes in the subject's state. Observers register themselves with the subject to receive updates.

- **ConcreteSubject:** This is the concrete implementation of the subject. It maintains the state and sends notifications to its observers when the state changes. It has observers’ list.

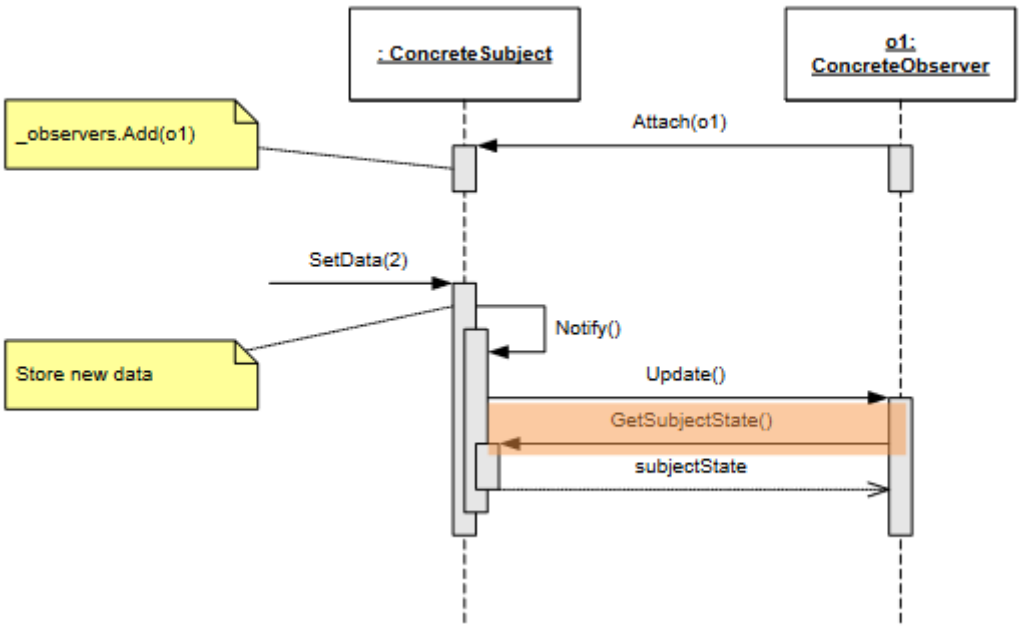
- **ConcreteObserver:** This is the concrete implementation of the observer. It implements the update method, which is called by the subject when its state changes.

Class Diagram

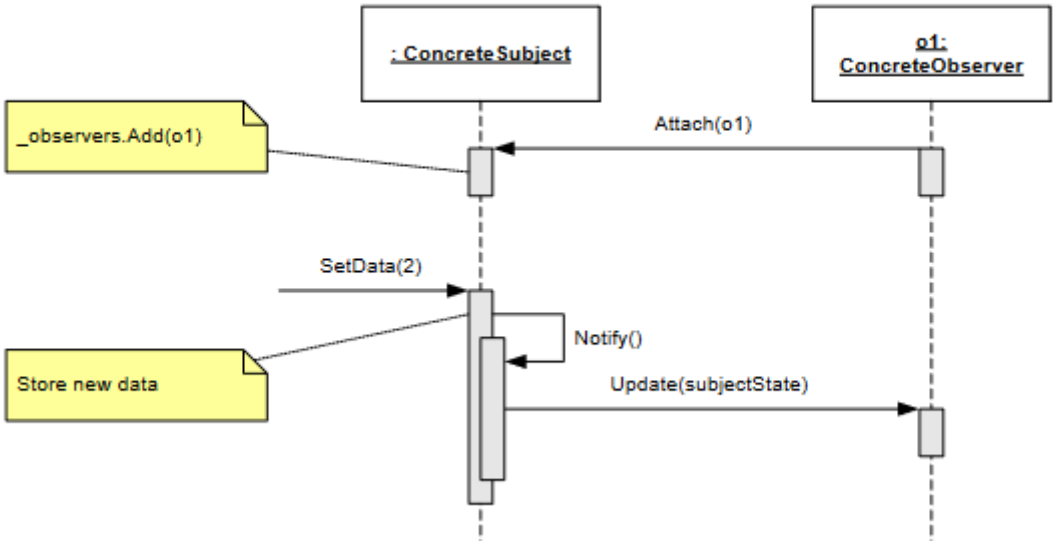


This one is "pull observer pattern" because ConcreteObserver Update method does not have any parameter

Sequence Diagram

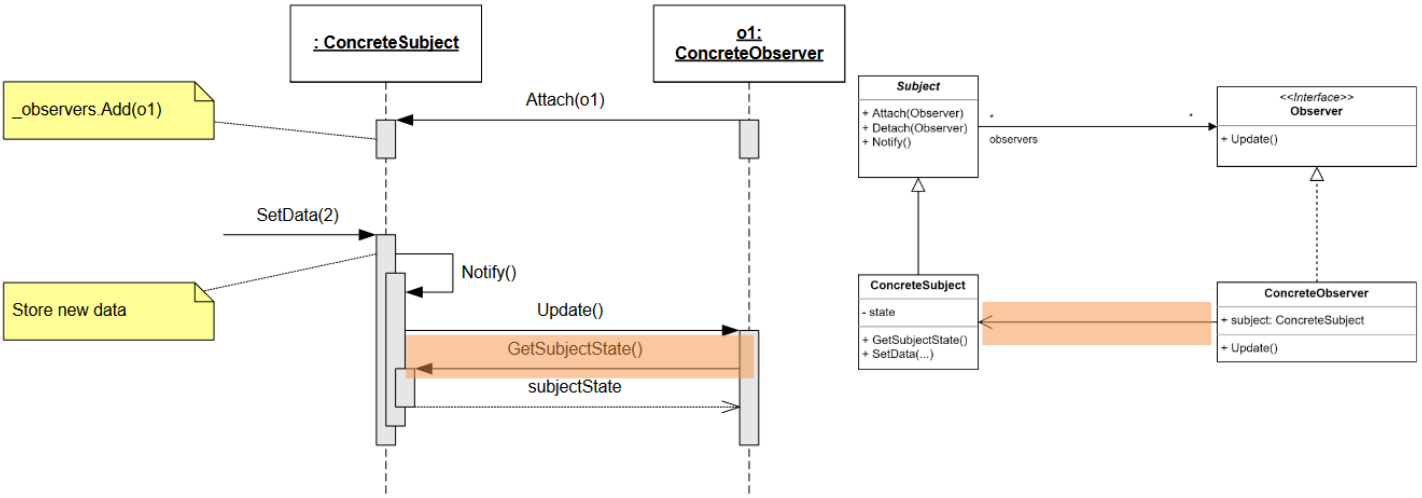


Sequence diagram of observer **pull** implementation.

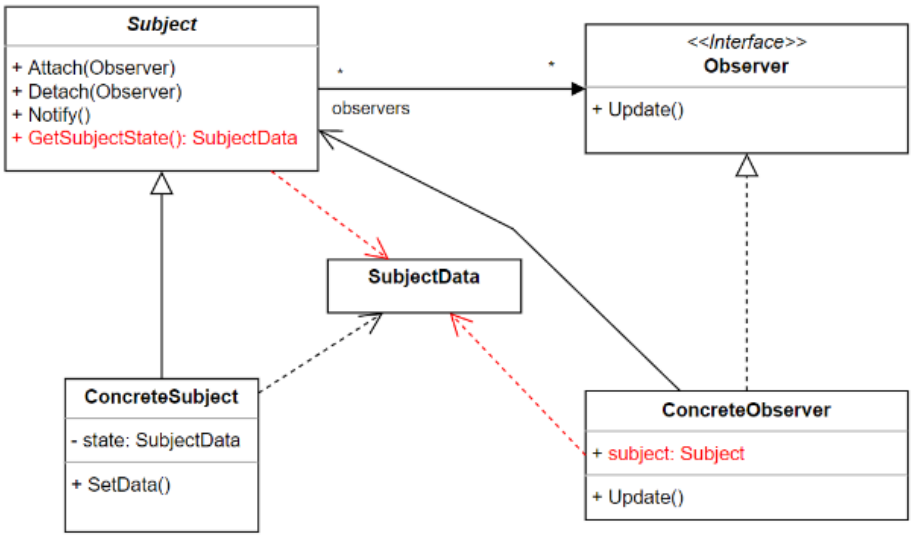


Sequence diagram of observer **push** implementation

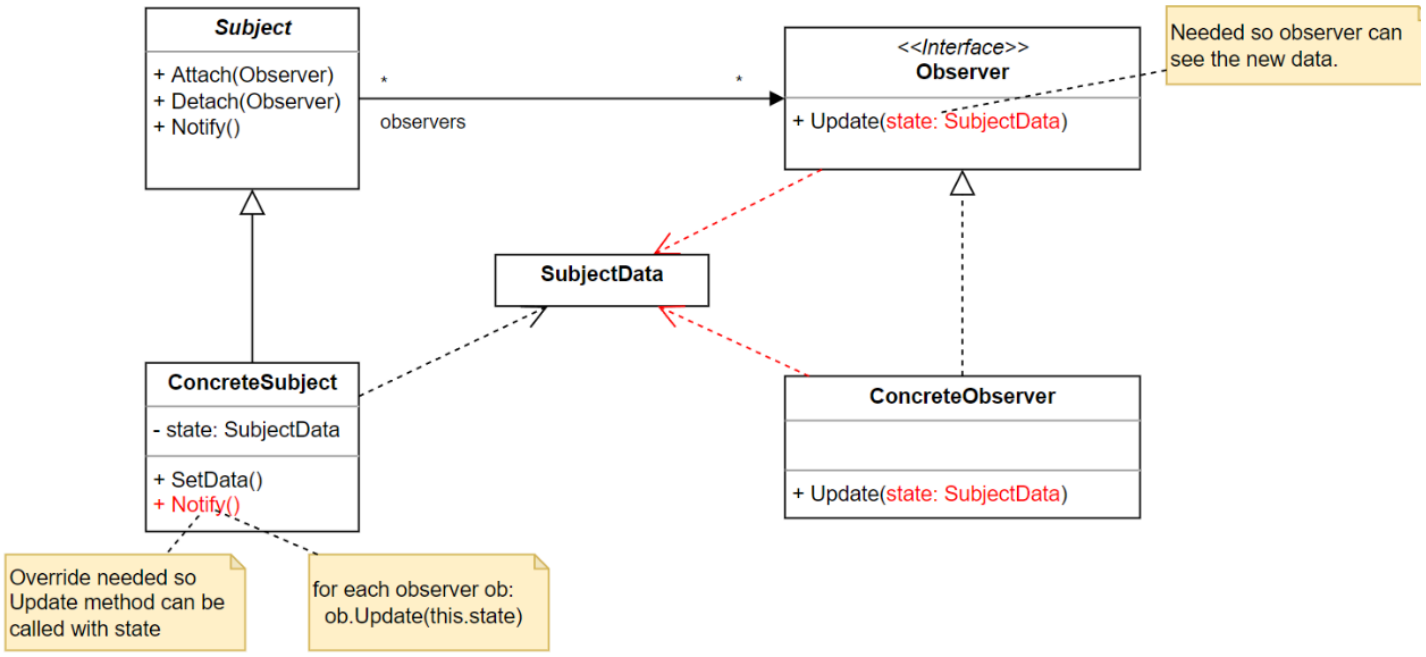
Problems of the ‘pull’ implementation



There is additional coupling between ConcreteSubject and ConcreteObserver



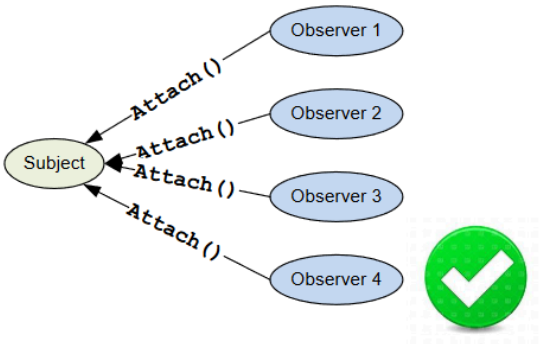
There are additional problems when adding the SubjectData class that wraps the data that the observer wants to be notified about.



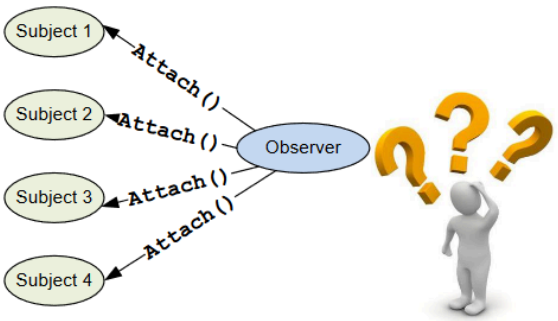
This is the push variant diagram. There is less coupling and ConcreteObserver has only generic 'Subject' passed - to attach itself to it.

Attaching to multiple subjects of THE SAME type

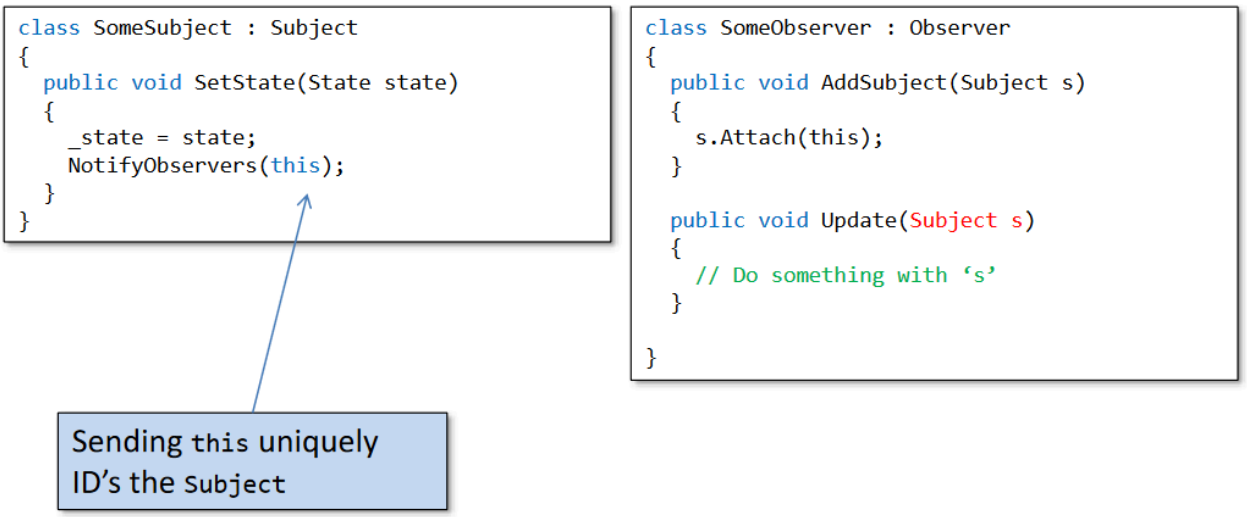
The variant of GoF Observer we have studied handles several observers registering on the *same* subject



How about one Observer attaching to several subjects of the **same** type?

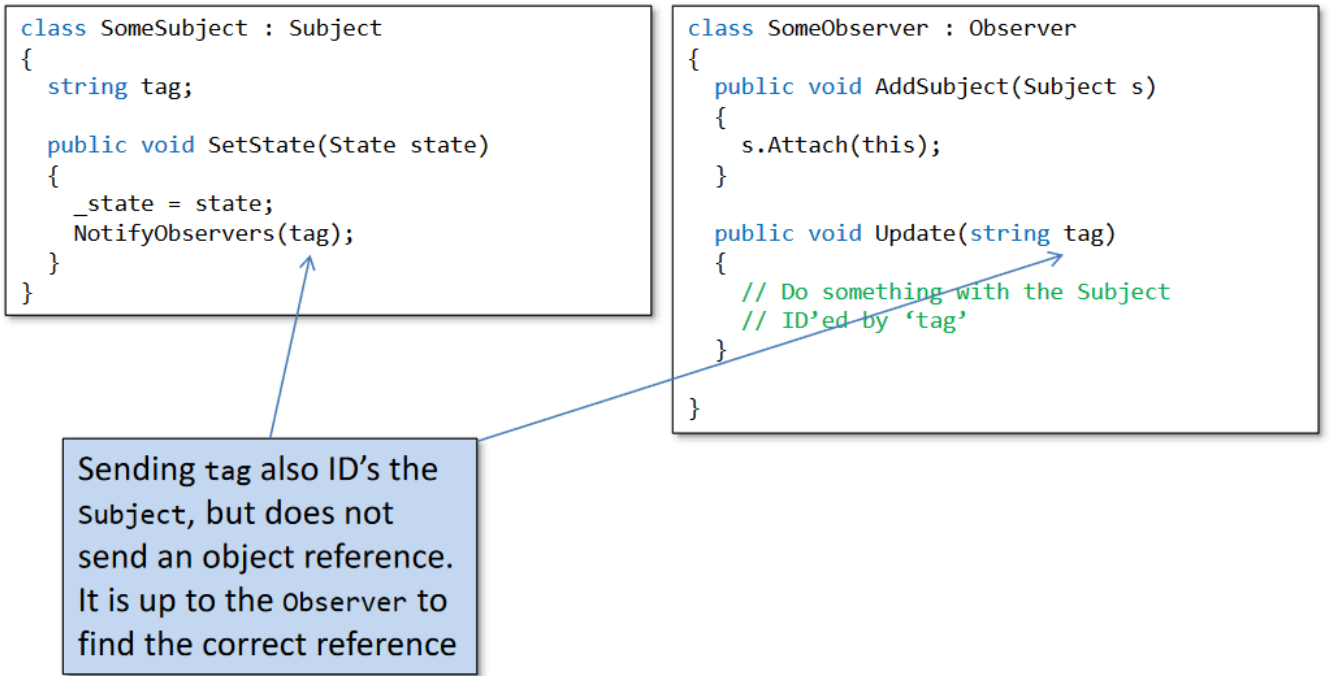


First idea:



NotifyObservers invokes Update for each of the observers Attached to the subject.

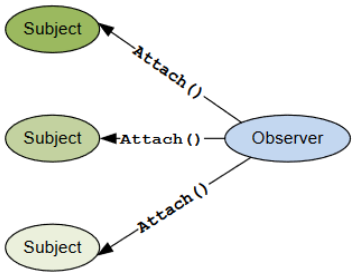
Second idea:



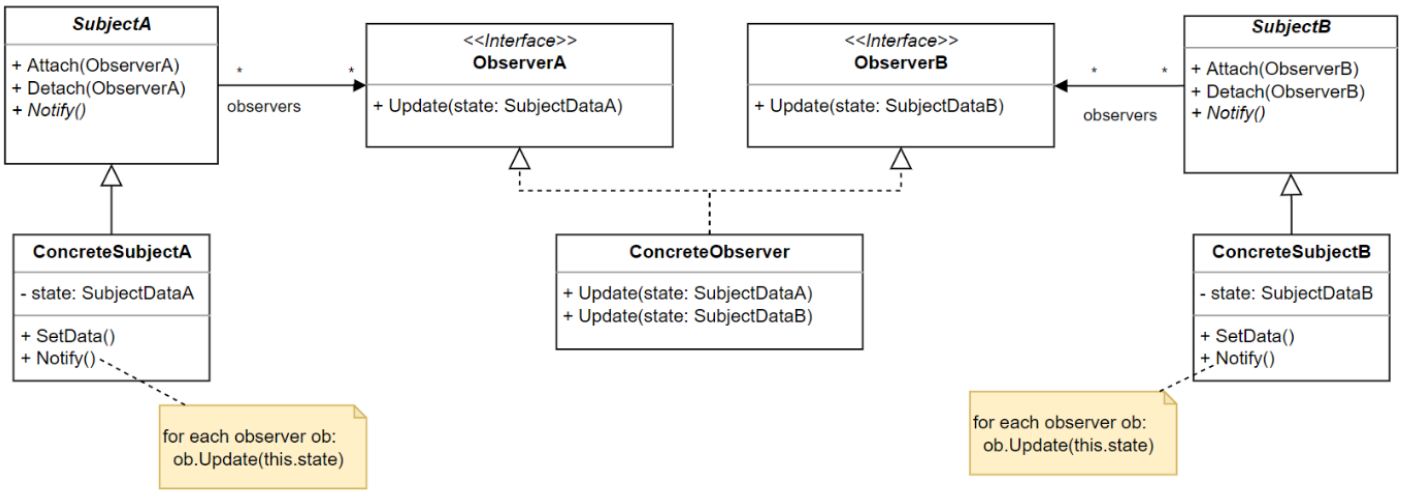
NotifyObservers invokes Update for each of the observers Attached to the subject.

Attaching to multiple subjects of DIFFERENT type

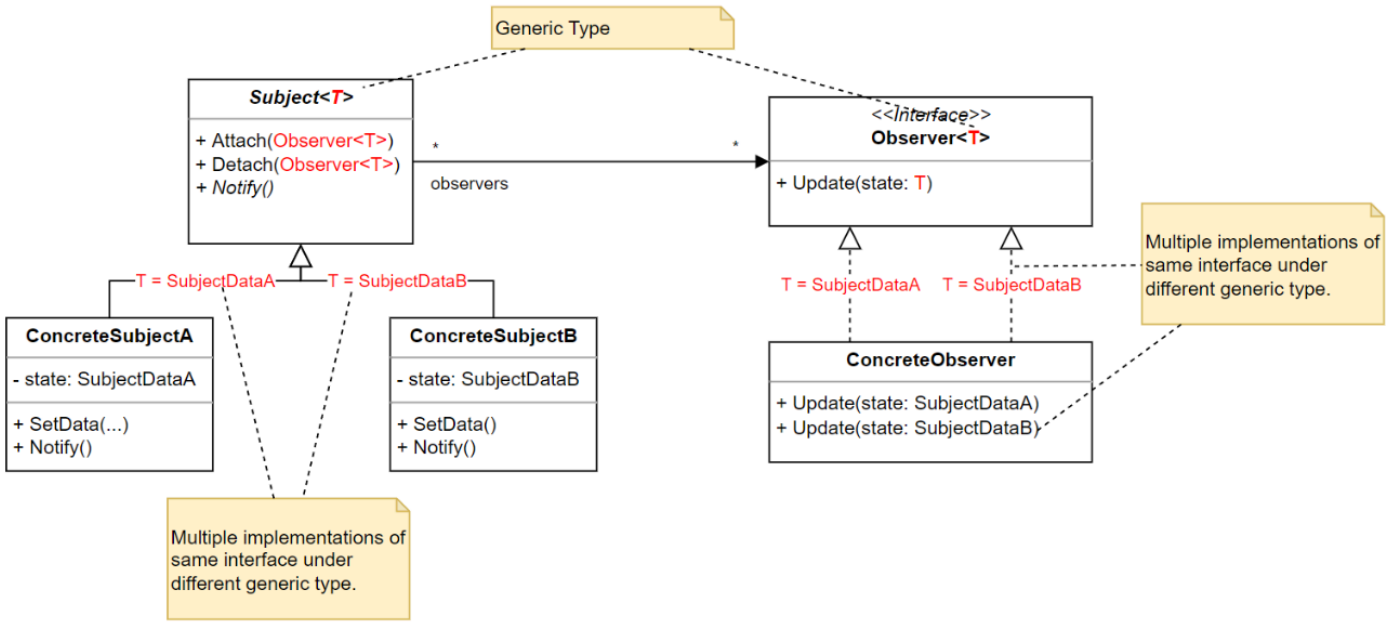
- How can we handle observers that connect to subjects of *different* types?
  - Again: Notice `Subject` is a generic **abstract base class**
  - `IObserver` as a generic **interface**!



GoF Observer – handling subjects of different types "manually"



This one is a bit bad, the upcoming one is better

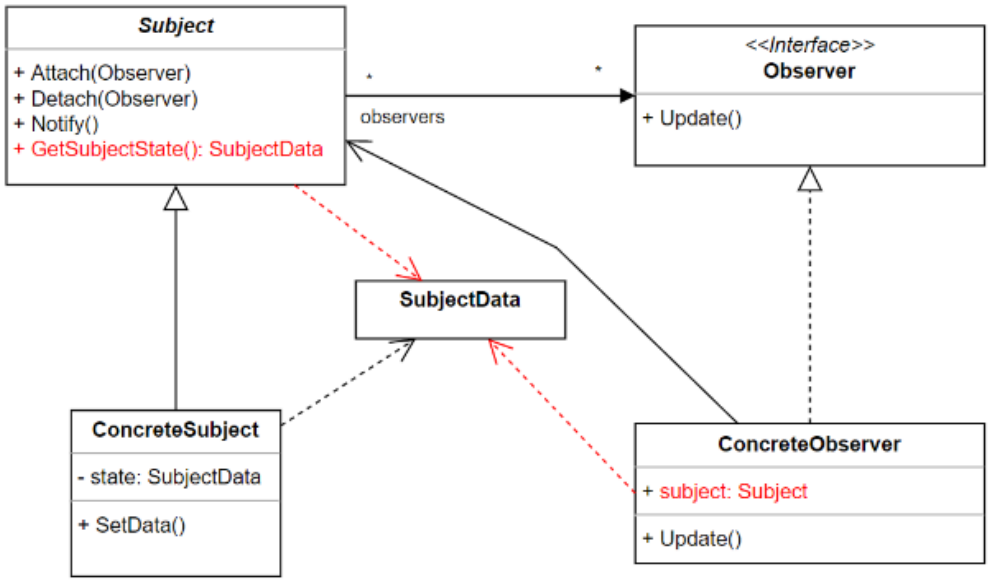


*Generic type is only to simplify attaching different observers to different topics with **different SubjectData** (which is the data that is related to the concrete subject). **Also Subject<T> is actually an interface, and it should be an interface because you don't know the SubjectData type in the generic Subject<T>**.*

Example

Pull version implementation

Diagram



Code

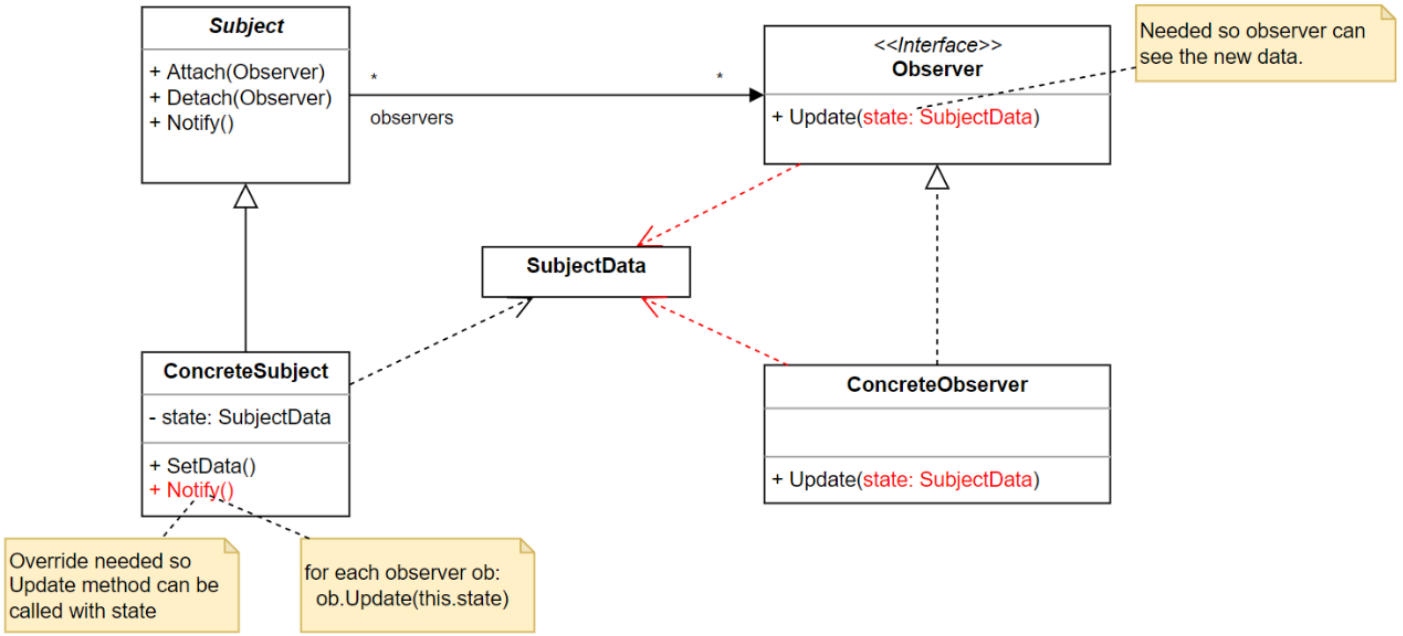
```
namespace ObserverPullVariant
{
    class SubjectData
    {
        public int Measurement { get; set; }
        public SubjectData(int measurement) {
             Measurement = measurement;
        }
    }
}
namespace ObserverPullVariant
{
    interface IObserver
    {
        public void Update();
    }
}


namespace ObserverPullVariant
```

```
{
    interface ISubject
    {
        public void Attach(IObserver observer);
        public void Detach(IObserver observer);
        public void NotifyAll();
    }
}



namespace ObserverPullVariant
{
    class ConcreteObserver : IObserver
    {
        // we are forced to use concrete subject type, because ISubject doesn't have
getter or getData method
        ConcreteSubject subject;
        SubjectData data ;
        int _id;
        public ConcreteObserver(ConcreteSubject subject, int id)
        {
            _id = id;
            this.subject = subject;
            subject.Attach(this);
            this.data = subject.Data;
        }
        public void Update()
        {
            //Pull data from the subject
            data = subject.Data;

            Console.WriteLine("Data has been changed for observer "+_id.ToString()+"
-> "+data.Measurement.ToString());

        }
    }
}
```

```
namespace ObserverPullVariant
{
    class ConcreteSubject : ISubject
    {
        List<IObserver> _observers;
        SubjectData _data;
        public SubjectData Data
        {
            get
            {
                return _data;
            }
            set
            {
                // after setting new data we notify all of the subscribers
                _data = value;
                NotifyAll();
            }
        }
        public ConcreteSubject() {

            _data = new SubjectData(0);
            _observers = new List<IObserver>();
        }

        public void Attach(IObserver observer)
        {
            _observers.Add(observer);
        }


        public void Detach(IObserver observer)
        {
           _observers.Remove(observer);
        }

        public void NotifyAll()
        {
            foreach(IObserver observer in _observers)
            {
                // we invoke update and observer pulls data with Data getter
                observer.Update();
            }
        }
    }
}
```

Push version implementation

Diagram



Code

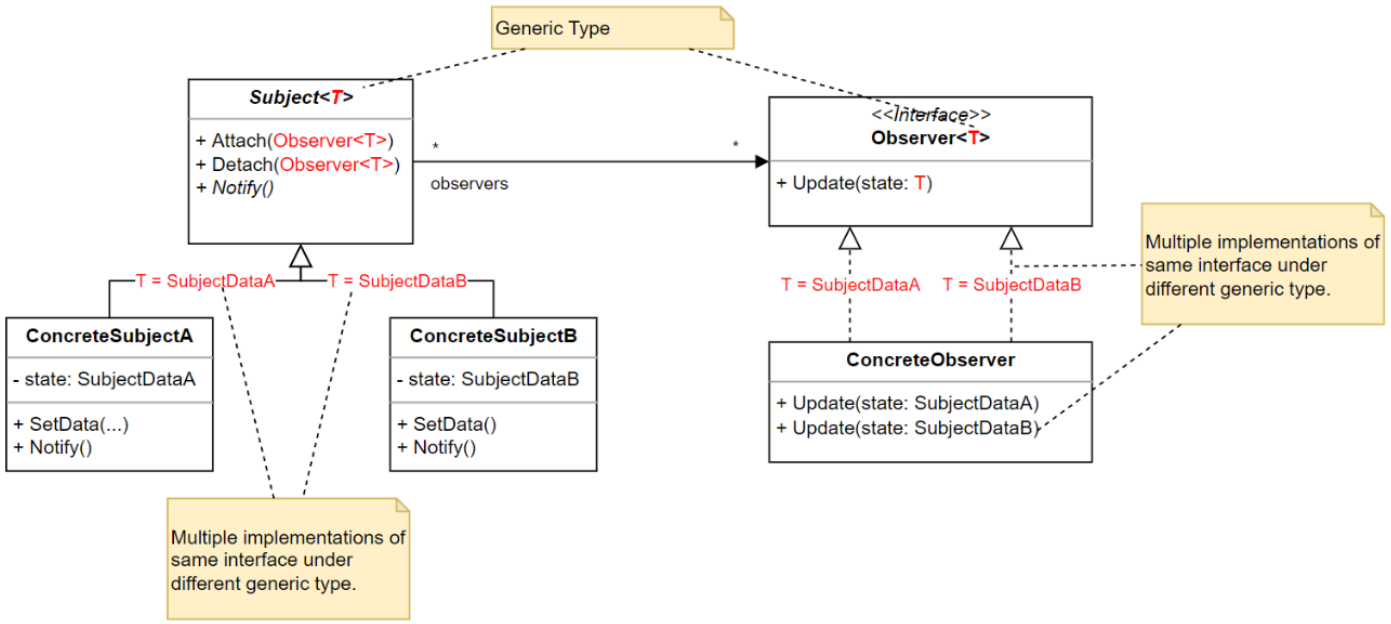
```
namespace ObserverPushVariant
{
    class SubjectData
    {
        public int Measurement { get; set; }
        public SubjectData(int measurement)
        {
            Measurement = measurement;
        }
    }
}

namespace ObserverPushVariant
{
    interface IObserver
    {
        void Update(SubjectData data);
    }
}

namespace ObserverPushVariant
{
    interface ISubject
    {
        void Attach(IObserver observer);
        void Detach(IObserver observer);
        void NotifyAll();
    }
```

```
}

namespace ObserverPushVariant
{
    class ConcreteSubject:ISubject
    {
        List<IObserver> _observers;

        SubjectData _data;

        public SubjectData Data
        {
            get
            {
                return _data;
            }
            set
            {
                // after setting new data we notify all of the subscribers
                _data = value;
                NotifyAll();
            }
        }
        public ConcreteSubject()
        {

            _data = new SubjectData(0);
            _observers = new List<IObserver>();
        }

        public void Attach(IObserver observer)
        {
          _observers.Add(observer);
        }

        public void Detach(IObserver observer)
        {
           _observers.Remove(observer);
        }

        public void NotifyAll()
        {
            // We "Push" data to the observer (we pass data as parameter from Subject
to Observer) that's the main difference between Pull vs Push variant- passing the
argument to Update function
            foreach(IObserver observer in _observers) {
                observer.Update(_data);
            }
```

```
        }
    }
}
namespace ObserverPushVariant
{
    class ConcreteObserver : IObserver
    {
        SubjectData _data;

        int _id;

        public ConcreteObserver(ISubject subject, int id)
        {
            subject.Attach(this);
            _id=id;
        }
        public void Update(SubjectData data)
        {
            _data = data;
            Console.WriteLine("Data has been changed for observer PUSH " +
_id.ToString() + " -> " + _data.Measurement.ToString());

        }
    }
}
```

Generic version implementation

Diagram



Code

What has changed:

- We use generics
- We have different SubjectData types
- We can make single ConcreteObserver subscribe to multiple Subjects of different SubjectData types
- ConcreteSubject implements particular SubjectData type by implementing ISubject<SubjectData> interface which is done with generics
- ConcreteObserver has to have additional methods to subscribe to a topic -> “subscribeToSubject”. If it didn’t have one we would have to modify the constructor (where we would pass Subjects) each time we want ConcreteObserver to subscribe to a new topic. We would break OCP. Now we don’t because we can just extend with another *subscribeToSubject* method

```
namespace ObserverGenericVariant
{
    public class SubjectDataA
    {
        public int Measurement { get; set; }

        public SubjectDataA(int data) {
        Measurement = data;
        }
    }
}
namespace ObserverGenericVariant
{
    class SubjectDataB
    {
        public string Measurement { get; set; }

        public SubjectDataB(string data) {

        Measurement = data;
        }

    }
}
namespace ObserverGenericVariant
{
    interface ISubject<T>
    {
        void Attach(IObserver<T> observer);
        void Detach(IObserver<T> observer);
        void NotifyAll();

    }
}
namespace ObserverGenericVariant
```

```
{
    interface IObserver<T>
    {
        public void Update(T data);
    }
}
namespace ObserverGenericVariant
{
    // the subject is for concrete ISubject implementation dependant on SubjectData T
    class ConcreteSubjectA : ISubject<SubjectDataA>
    {
        List<IObserver<SubjectDataA>> _observers = new
List<IObserver<SubjectDataA>>();
        SubjectDataA _data = new SubjectDataA(0);
        public SubjectDataA Data
        {
            get
            {
                return _data;
            }
            set
            {
                // after setting new data we notify all of the subscribers
                _data = value;
                NotifyAll();
            }
        }
        public void Attach(IObserver<SubjectDataA> observer)
        {
            _observers.Add(observer);
        }

        public void Detach(IObserver<SubjectDataA> observer)
        {
            _observers.Remove(observer);
        }

        public void NotifyAll()
        {
            foreach (IObserver<SubjectDataA> observer in _observers)
            {
                observer.Update(_data);
            }
        }
    }
}
namespace ObserverGenericVariant
{
```

```
    class ConcreteSubjectB : ISubject<SubjectDataB>
    {
        List<IObserver<SubjectDataB>> _observers = new
List<IObserver<SubjectDataB>>();
        SubjectDataB _data = new SubjectDataB("0");
        public SubjectDataB Data
        {
            get
            {
                return _data;
            }
            set
            {
                // after setting new data we notify all of the subscribers
                _data = value;
                NotifyAll();
            }
        }
        public void Attach(IObserver<SubjectDataB> observer)
        {
            _observers.Add(observer);
        }

        public void Detach(IObserver<SubjectDataB> observer)
        {
            _observers.Remove(observer);
        }

        public void NotifyAll()
        {
            foreach (IObserver<SubjectDataB> observer in _observers)
            {
                observer.Update(_data);
            }
        }
    }
}

namespace ObserverGenericVariant
{
    // Observer is able to implement multiple IObserver interfaces and subscribe to
different topics
    // Attach method invokation has to be moved from constructor to different methods
to adhere to OCP
    // If we didn't move it we would break OCP cause when adding another type of
subjec we would have to
    // Modify the constructor instead of expanding the class
    class ConcreteObserverAB : IObserver<SubjectDataA>, IObserver<SubjectDataB>
```

```
    {
        SubjectDataA _dataA = new SubjectDataA(0);
        SubjectDataB _dataB = new SubjectDataB("0");

        int _id;
        public ConcreteObserverAB(int id)
        {
            _id = id;

        }
        // There must be different subscribtion for each of the concrete subjects
        public void subscribeToSubject(ConcreteSubjectA subject)
        {
            subject.Attach(this);
        }
        public void subscribeToSubject(ConcreteSubjectB subject)
        {
            subject.Attach(this);
        }

        public void Update(SubjectDataA data)
        {
            _dataA = data;
            Console.WriteLine("Data (SubjectDataA) has been changed for observer " +
_id.ToString() + " -> " + _dataA.Measurement.ToString());
        }

        public void Update(SubjectDataB data)
        {

            _dataB = data;
            Console.WriteLine("Data (SubjectDataB) has been changed for observer " +
_id.ToString() + " -> " + _dataB.Measurement.ToString());
        }
    }
}
```

3. Factory patterns

Factories

The general goals of factories are:

- to separate the creation of an object from its use – SRP
- Make creation code open for extension but closed for modification (OCP)

Factories come in many different flavors – we will look at two classics:

- **GoF Factory Method** - Define an interface for creating an object, but let the classes that implement the interface decide which object to instantiate.
- **GoF Abstract Factory -** Define an interface for creating families of related or dependent objects without specifying their concrete classes

Factory Method

Characteristics

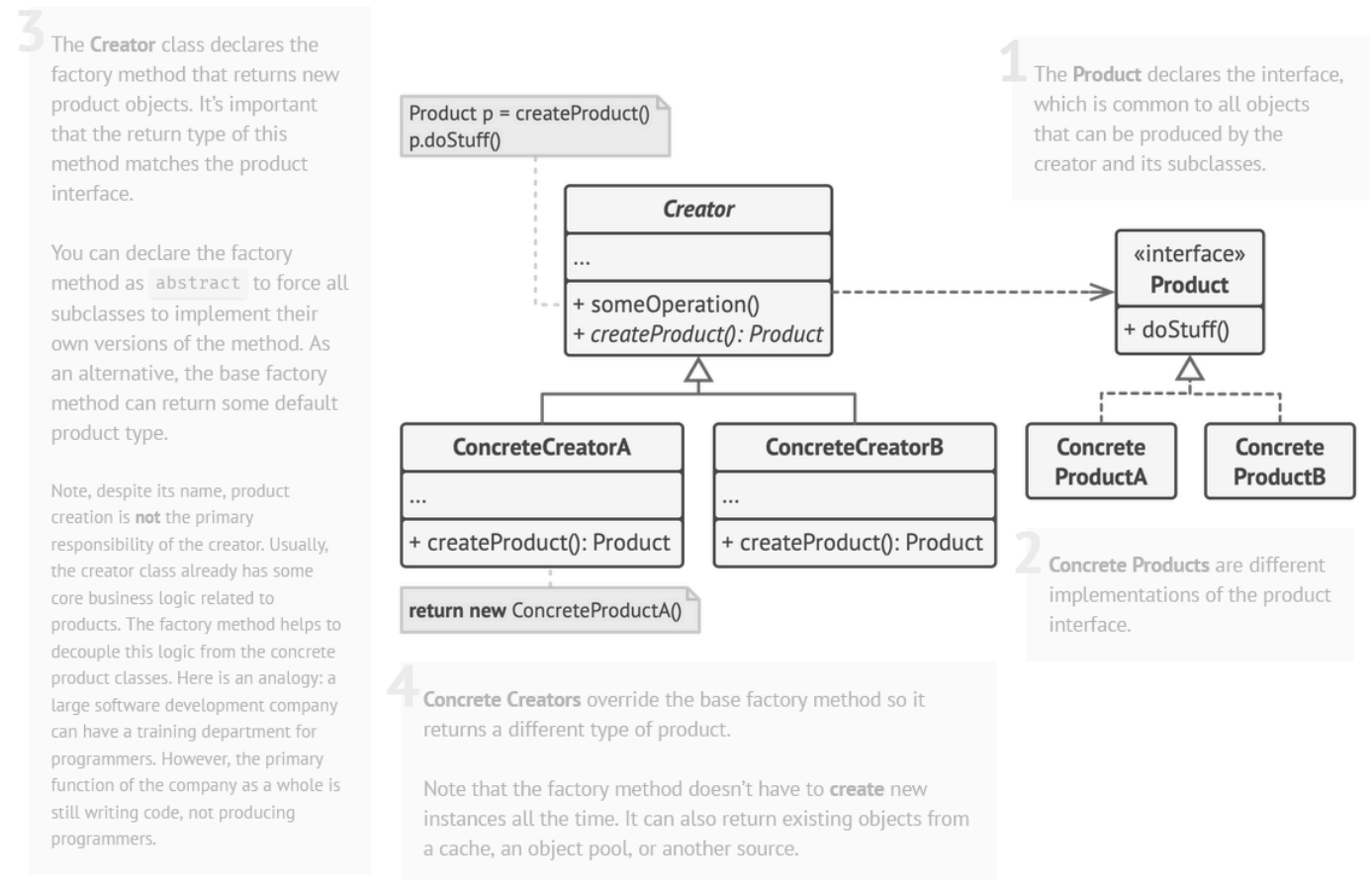
OCP and Factory Method:

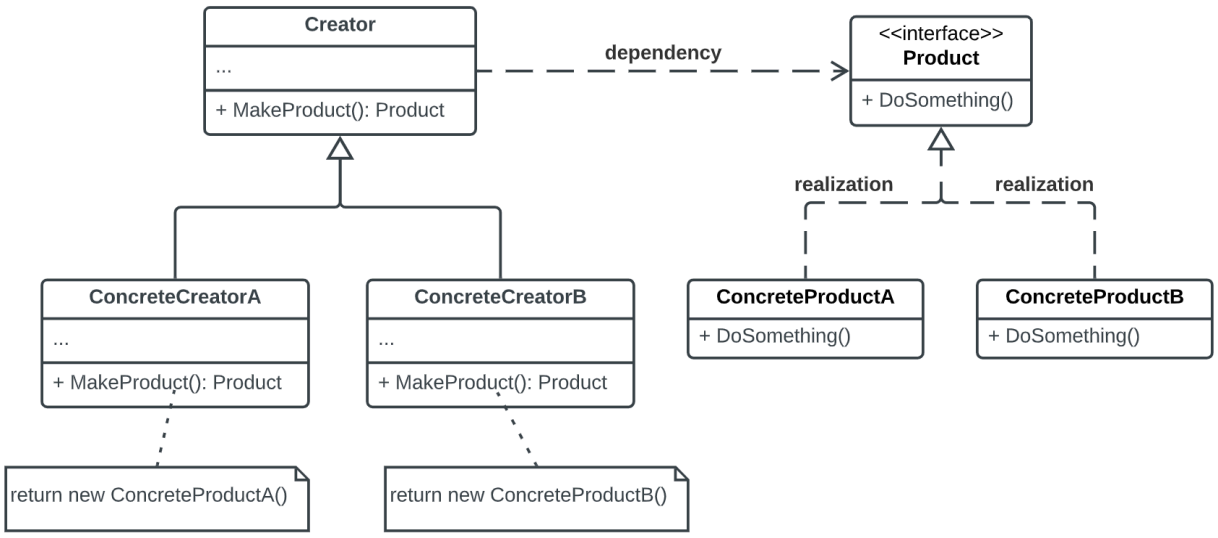
The Factory Method pattern is often used to adhere to the Open/Closed Principle. By relying on a factory method to create objects, you can introduce new types of objects (extensions) without modifying existing code. Each subclass can provide its own implementation of the factory method, allowing the system to be open for extension.

SRP and Factory Method:

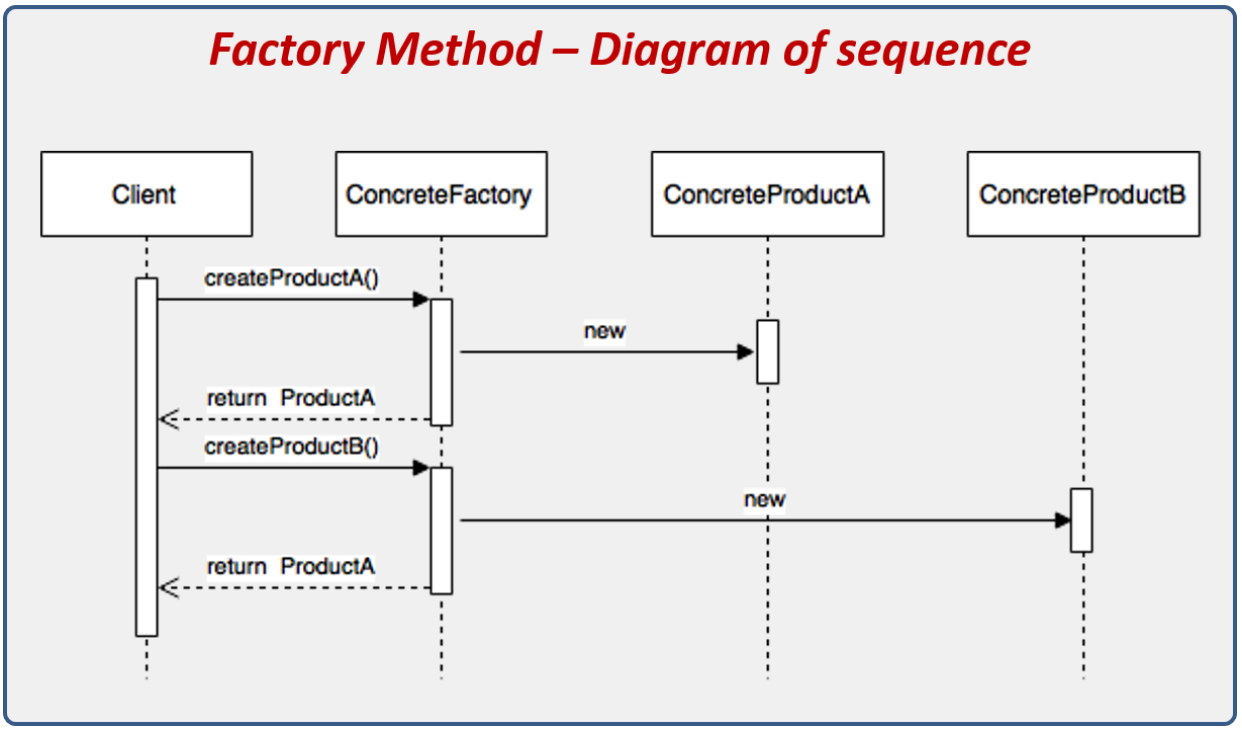
The Factory Method Pattern can help in adhering to the Single Responsibility Principle by encapsulating the object creation process in a separate class or method. This promotes separation of concerns, where one class is responsible for creating instances (the factory), and another class is responsible for the main business logic. Each class has a single responsibility.

Class diagram





Sequence diagram

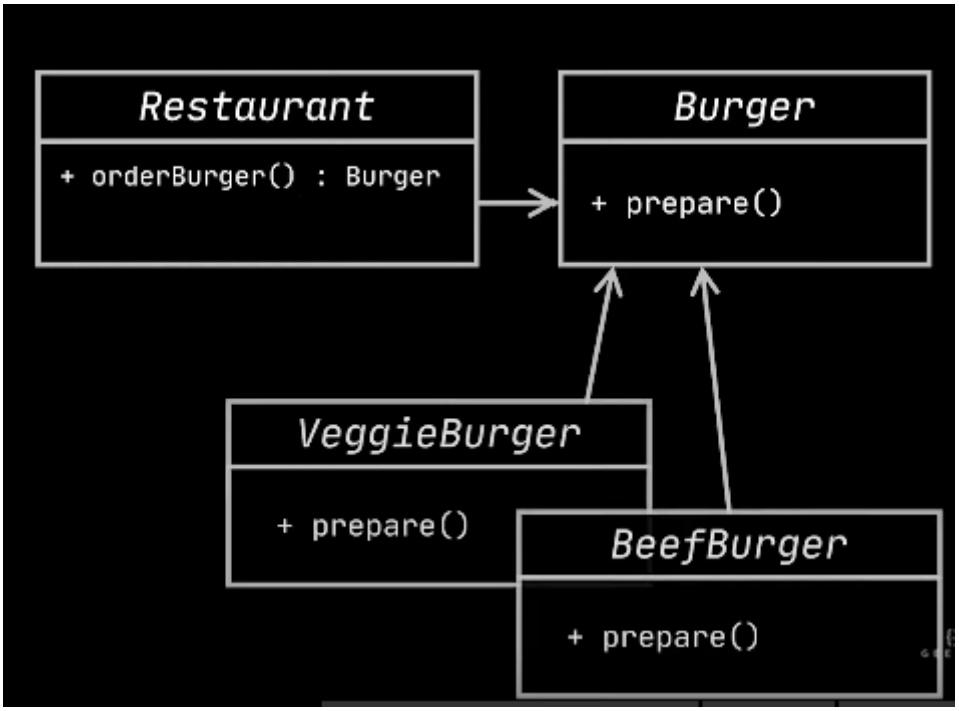


Example

Problem

Let’s consider a restaurant that makes different types of burgers.

Diagram

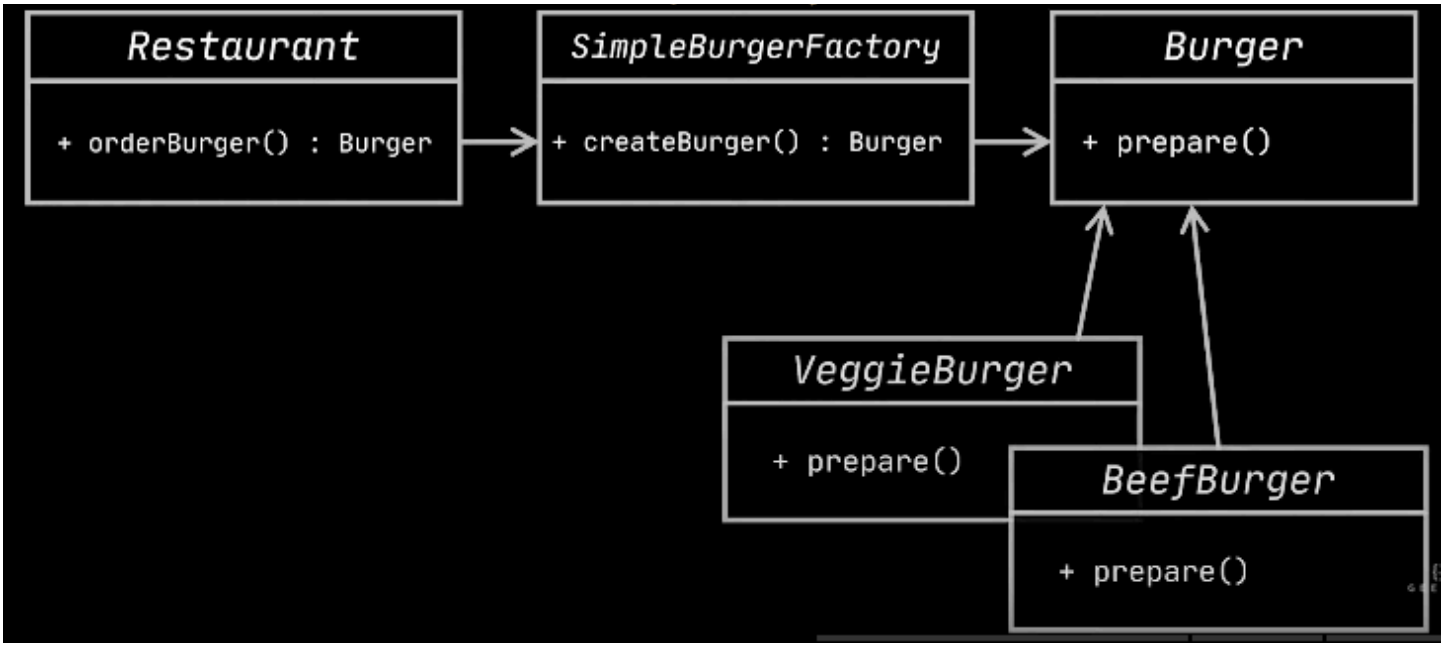


Code

```
namespace BurgerProblem
{
    public abstract class AbstractBurger
    {
        int ProductId;
        string Addons;

        public AbstractBurger(int id, string addons) {
            ProductId = id;
            Addons = addons;

        }
       public abstract void prepare();
    }
}


namespace BurgerProblem
{
    public class BeefBurger:AbstractBurger
    {
        bool HasAngus;

        public BeefBurger(int id, string addons, bool hasAngus):base(id,addons)
        {
            HasAngus = hasAngus;
```

```
        }

        public override void prepare()
        {
            Console.WriteLine("Preparing BeefBurger!");
        }
    }
}

namespace BurgerProblem
{
    public class VeggieBurger : AbstractBurger
    {
        bool HasOlives;

        public VeggieBurger(int id, string addons, bool hasOlives) : base(id, addons)
        {
            HasOlives = hasOlives;

        }

        public override void prepare()
        {
            Console.WriteLine("Preparing VeggieBurger!");
        }
    }
}



namespace BurgerProblem
{
    public class Restaurant
    {
        // This class has two responsibilites (creating and ordering burgers)
        // It also breaks OCP, because each time we add new burger type we need to
modify it
        public AbstractBurger? OrderBurger(string burgerName)
        {
            AbstractBurger? burger = null;

            if(burgerName == "BEEF") {

                burger = new BeefBurger(1, "beef and stuff", true);

            }
            else if(burgerName == "VEGGIE")
            {
```

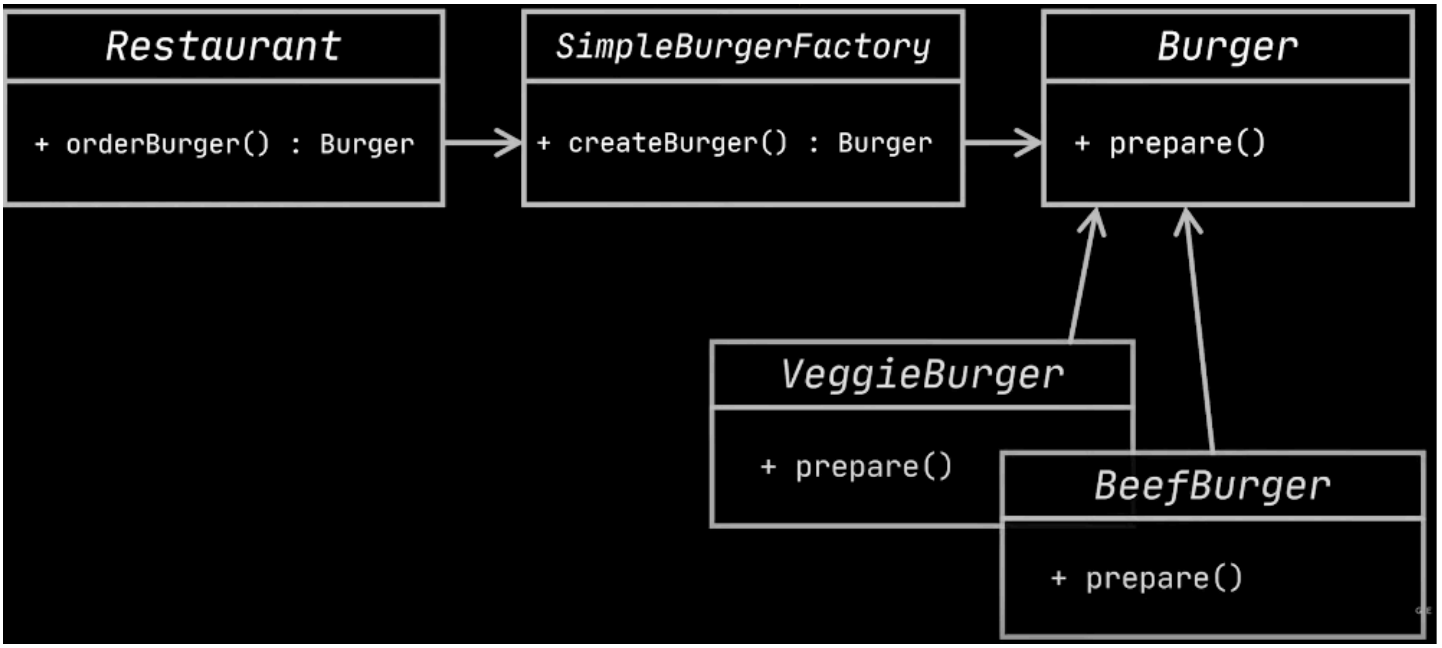
```
                burger = new VeggieBurger(2, "veggies and stuff", true);
            }

            burger?.prepare();

            return burger;
        }
    }
}

namespace BurgerProblem
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Restaurant restaurant = new Restaurant();
            restaurant.OrderBurger("BEEF");
            restaurant.OrderBurger("VEGGIE");
        }

    }
}
```



Restaurant class breaks OCP because every time you add a new type of Burger you need to modify Restaurant class (despite the use of AbstractBurger). Also we break SRP because we both create and prepare the burger in the Restaurant class. Therefore we can try to create a simple factory (this is not FactoryMethod pattern yet).

Factory Idiom

Diagram



Code

```
namespace BurgerFactoryIdiom
{
    public abstract class AbstractBurger
    {
        int ProductId;
        string Addons;

        public AbstractBurger(int id, string addons) {
            ProductId = id;
            Addons = addons;
        }

       public abstract void prepare();
    }
}
namespace BurgerFactoryIdiom
{
    public class BeefBurger:AbstractBurger
    {
        bool HasAngus;

        public BeefBurger(int id, string addons, bool hasAngus):base(id,addons)
        {
            HasAngus = hasAngus;

        }

        public override void prepare()
```

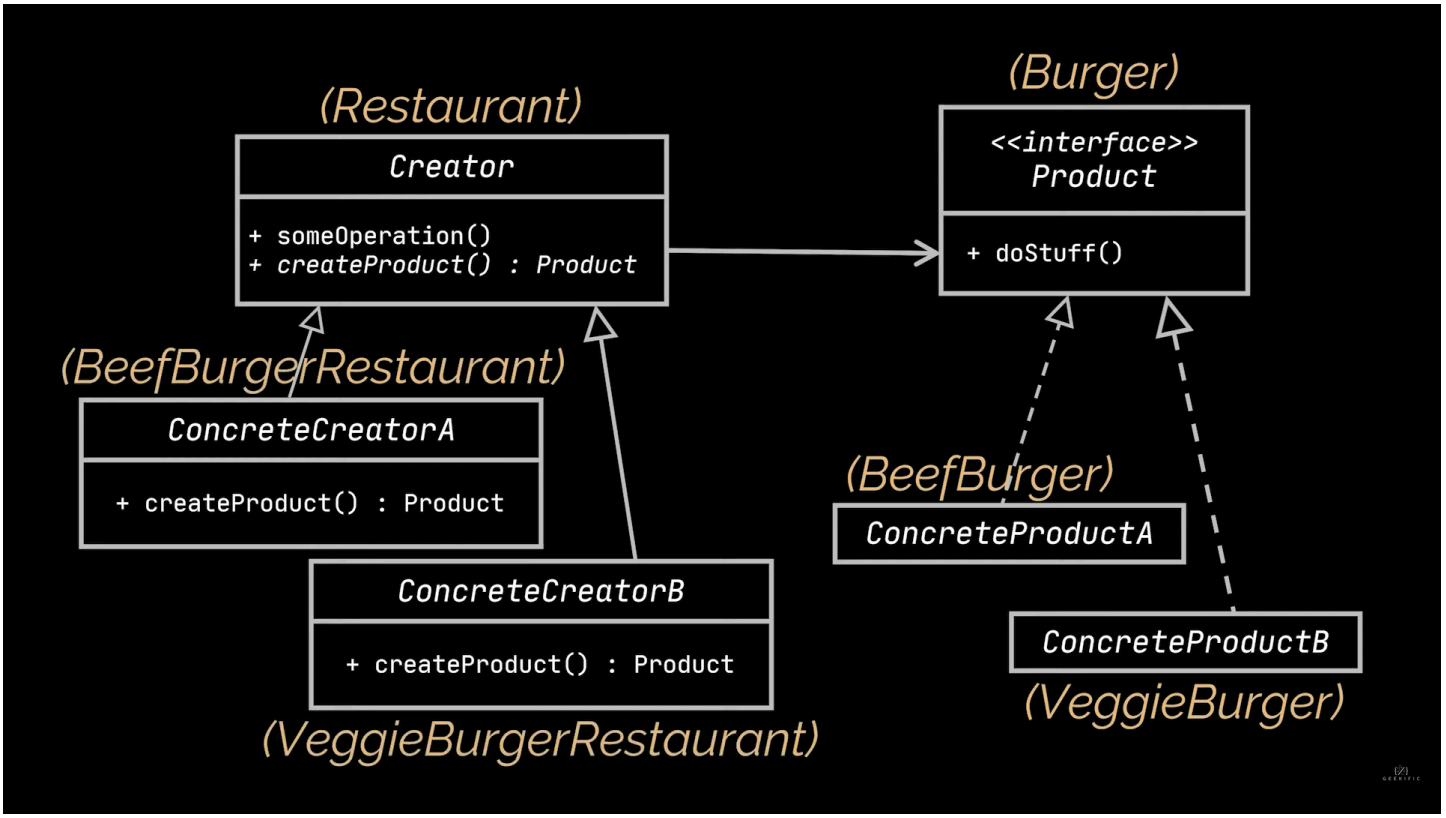
```
        {
            Console.WriteLine("Preparing BeefBurger!");
        }
    }
}
namespace BurgerFactoryIdiom
{
    public class VeggieBurger : AbstractBurger
    {
        bool HasOlives;

        public VeggieBurger(int id, string addons, bool hasOlives) : base(id, addons)
        {
            HasOlives = hasOlives;

        }

        public override void prepare()
        {
            Console.WriteLine("Preparing VeggieBurger!");
        }
    }
}
using System;

namespace BurgerFactoryIdiom
{
    public class BurgerFactory
    {

        // Breaks OCP, because each time you add new burger type, you need to edit
this method
        public AbstractBurger CreateBurger(string burgerName)
        {
            AbstractBurger burger = null;

            if (burgerName == "BEEF")
            {

                burger = new BeefBurger(1, "beef and stuff", true);

            }
            else if (burgerName == "VEGGIE")
            {
                burger = new VeggieBurger(2, "veggies and stuff", true);
            }

            burger?.prepare();
```

```
            return burger;
        }
    }
}
namespace BurgerFactoryIdiom
{
    public class Restaurant
    {
        // This class adheres to SRP as the rest of the classes
        // But BurgerFactory still breaks OCP
        public AbstractBurger? OrderBurger(string burgerName)
        {
            BurgerFactory burgerFactory = new BurgerFactory();
            AbstractBurger burger  = burgerFactory.CreateBurger(burgerName);

            return burger;
        }
    }
}

namespace BurgerFactoryIdiom
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Restaurant restaurant = new Restaurant();
            restaurant.OrderBurger("BEEF");
            restaurant.OrderBurger("VEGGIE");
        }

    }
}
```

We still break OCP in BurgerFactory.

Factory method

Diagram



Code

```
namespace BurgerFactoryMethod
{
    public abstract class AbstractBurger
    {
        int ProductId;
        string Addons;

        public AbstractBurger(int id, string addons) {
            ProductId = id;
            Addons = addons;
        }
       public abstract void prepare();
    }
}
namespace BurgerFactoryMethod
{
    public class BeefBurger:AbstractBurger
    {
        bool HasAngus;
        public BeefBurger(int id, string addons, bool hasAngus):base(id,addons)
        {
             HasAngus = hasAngus;
        }
```

```
        public override void prepare()
        {
            Console.WriteLine("Preparing BeefBurger!");
        }
    }
}
namespace BurgerFactoryMethod
{
    public class VeggieBurger : AbstractBurger
    {
        bool HasOlives;
        public VeggieBurger(int id, string addons, bool hasOlives) : base(id, addons)
        {
            HasOlives = hasOlives;
        }
        public override void prepare()
        {
            Console.WriteLine("Preparing VeggieBurger!");
        }
    }
}
namespace BurgerFactoryMethod
{
    public abstract class AbstractRestaurant
    {
        public AbstractBurger? OrderBurger()
        {
            AbstractBurger burger  = CreateBurger();
            burger.prepare();
            return burger;
        }

        protected abstract AbstractBurger CreateBurger();
    }
}

namespace BurgerFactoryMethod
{
    class BeefBurgerRestaurant : AbstractRestaurant
    {
        protected override AbstractBurger CreateBurger()
        {
            return new BeefBurger(1,"beef and stuff",true);
        }
    }
}
```

```
namespace BurgerFactoryMethod
{
    class VeggieBurgerRestaurant : AbstractRestaurant
    {
        protected override AbstractBurger CreateBurger()
        {
            return new VeggieBurger(2,"veggies and stuff",true);
        }
    }
}

namespace BurgerFactoryMethod
{
    public class Program
    {
        public static void Main(string[] args)
        {
            AbstractRestaurant beefBurgerRestaurant = new BeefBurgerRestaurant();
            beefBurgerRestaurant.OrderBurger();

            VeggieBurgerRestaurant veggieBurgerRestaurant = new
VeggieBurgerRestaurant();
            veggieBurgerRestaurant.OrderBurger();

        }

    }
}
```

Instead of Restaurant we have AbstractRestaurant with OrderBurger method and CreateBurger abstract method that is implemented by subclasses. Both OrderBurger and CreateBurger methods do not involve passing burgerType and checking whether it's "Beef" or "Veggie" like in previous examples. We adhere to SRP and to OCP. Every time we add a new burgerType we just create it by inheriting from the AbstractBuger class and we create a new BurgerRestaurant for serving it. Also look at the Main invocation.

Abstract Factory

Define an interface for creating families of related or dependent objects without specifying their concrete classes.

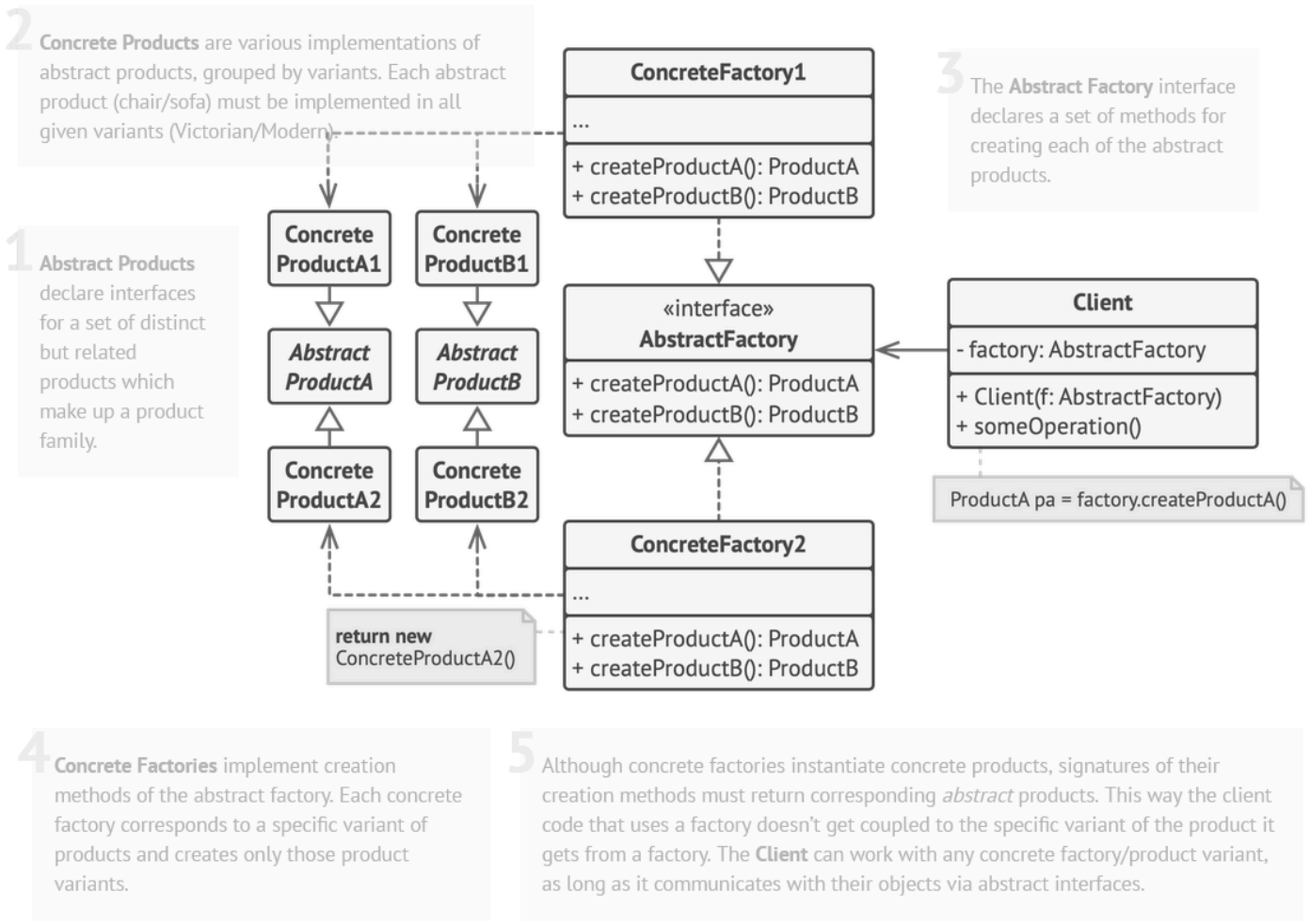
Characteristics

1. **Single Responsibility Principle (SRP):**
    - Abstract Factory helps in adhering to the SRP by encapsulating the object creation process in a separate set of factory classes. Each concrete factory has the responsibility of creating a family of related products (e.g., characters and equipment), which ensures that each class has a single responsibility.
2. **Open/Closed Principle (OCP):**

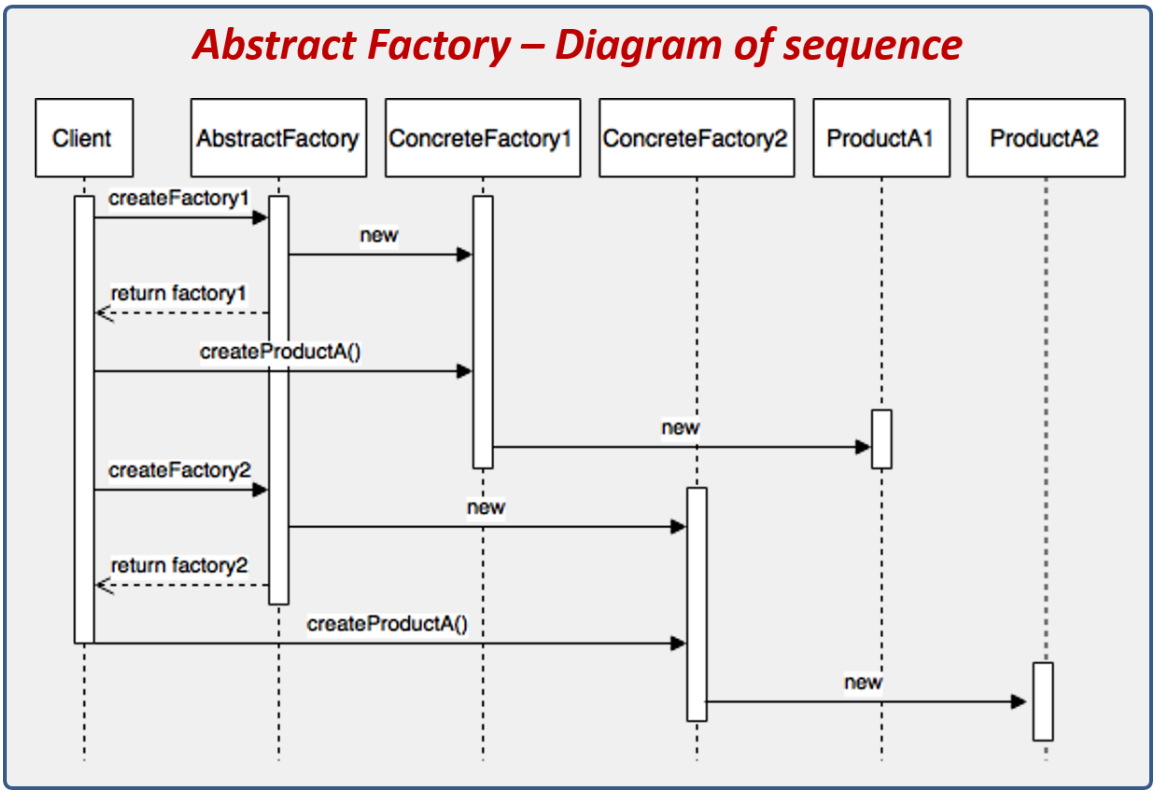
- Abstract Factory supports the OCP by allowing the introduction of new types of objects (extensions) without modifying existing code. Each concrete factory can provide its own implementation of the factory methods, enabling the system to be open for extension. When a new family of products is needed, you can create a new concrete factory without altering existing client code.

3. **Interface Segregation Principle (ISP):**
   - Abstract Factory tends to adhere to the ISP by providing separate interfaces or abstract classes for creating different families of products. Clients can choose to implement and use only the interfaces relevant to their needs. For example, a client might be interested in creating characters and may not need to deal with the details of creating equipment.
4. **Dependency Inversion Principle (DIP):**
   - Abstract Factory aligns with the DIP by allowing client code to depend on abstractions (interfaces or abstract classes) rather than concrete implementations. Clients interact with the abstract factory and product interfaces, promoting a high-level of abstraction and decoupling between the client and the concrete product implementations.

Class diagram



Sequence diagram



Example 1

Let's continue the example with FactoryMethod, BurgerRestaurant and Burgers

Let's say we would like to add another type of product that might be created by our Restaurant. Except for classical burgers we want to add Italian burgers.

We will try to do the following:

- Change BeefBurger to BeefClassicBurger
- Add ItalianBeefBurger to the BeefBurgerRestaurant

The Problem

```
namespace BurgerFactoryMethod
{
    class BeefBurgerRestaurant : AbstractRestaurant
    {
        protected override AbstractBurger CreateBurger(string burgerType)
        {      // We now again break OCP
            if (burgerType == "classic")
            {
                 return new BeefClassicBurger(1,"beef and stuff",true);
            }
            else if(burgerType == "italian")
            {
                return new BeefItalianBurger(1, "beef and stuff", true);
            }
            return null;
```

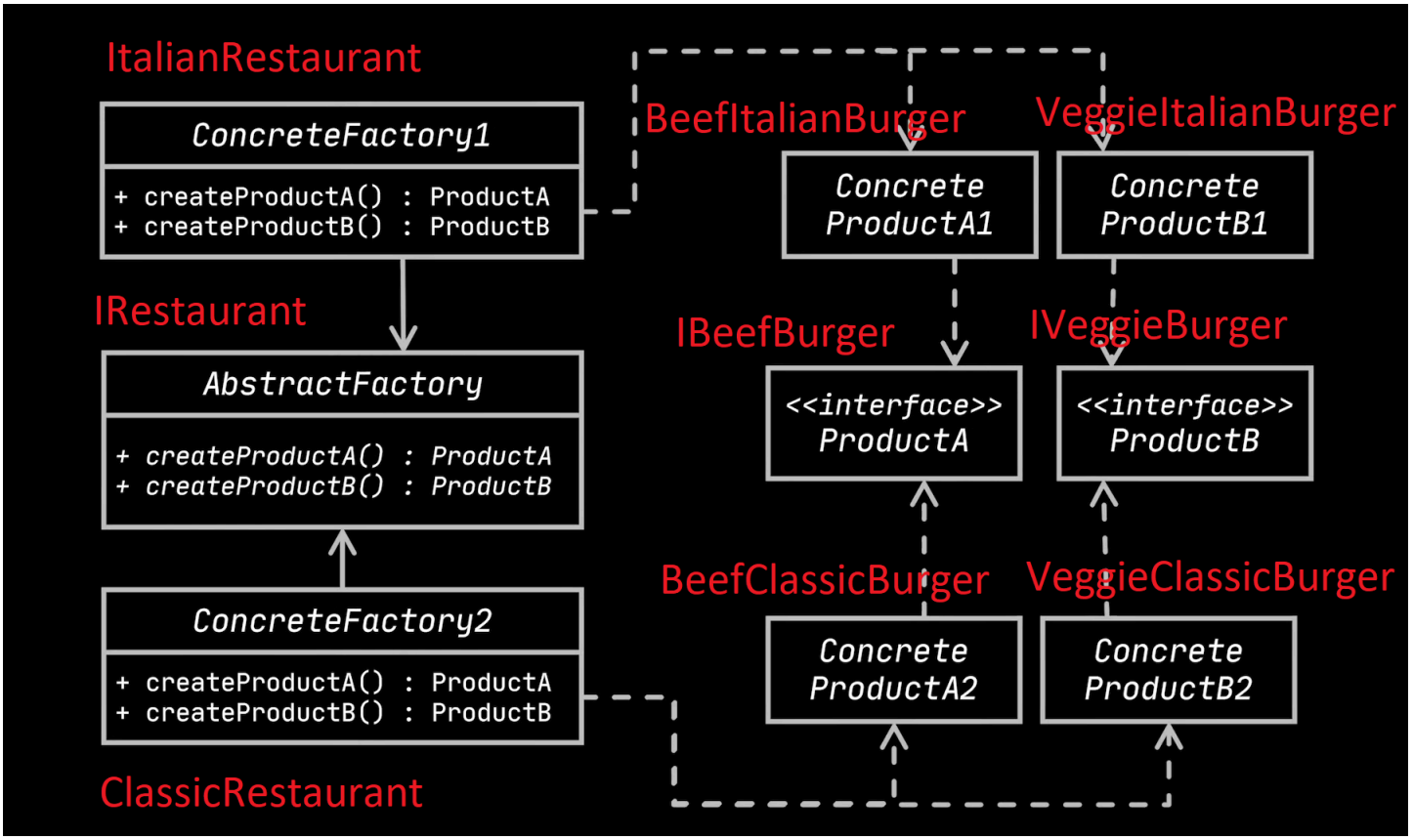
```
        }
    }
}
```

The Solution Diagram

We will try to do the following:

- Create interface for each of types of burgers IBeefBurger and IVeggieBurger

We end up with:



The Solution Code

```
public interface IBeefBurger
{
    public void Prepare();
}

public interface IVeggieBurger
{
    public void Prepare();
}

public class BeefClassicBurger : IBeefBurger
 {
     int Id;
```

```
    string Addons;
    bool HasAngus;


    public BeefClassicBurger(int id, string addons, bool hasAngus)
    {
        Id = id;
        Addons = addons;
        HasAngus = hasAngus;

    }

    public void Prepare()
    {
        Console.WriteLine("Preparing BeefClassicBurger!");
    }
}

public class BeefItalianBurger : IBeefBurger
{
    int Id;
    string Addons;
    bool HasOlives;


    public BeefItalianBurger(int id, string addons, bool hasOlives)
    {
        Id = id;
        Addons = addons;
        HasOlives = hasOlives;

    }

    public void Prepare()
    {
        Console.WriteLine("Preparing BeefItalianBurger!");
    }
}

public class VeggieClassicBurger : IVeggieBurger
  {
      int Id;
      string Addons;
      bool HasOlives;


      public VeggieClassicBurger(int id, string addons, bool hasOlives)
      {
```

```
        Id = id;
        Addons = addons;
        HasOlives = hasOlives;

    }

    public void Prepare()
    {
        Console.WriteLine("Preparing VeggieClassicBurger!");
    }
}

    public class VeggieItalianBurger : IVeggieBurger
  {
      int Id;
      string Addons;
      bool HasAngus;


      public VeggieItalianBurger(int id, string addons, bool hasAngus)
      {
          Id = id;
          Addons = addons;
          HasAngus = hasAngus;

      }

      public void Prepare()
      {
          Console.WriteLine("Preparing VeggieItalianBurger!");
      }
  }
```

```
public interface IRestaurant
{
      public IBeefBurger CreateBeefBurger();
      public IVeggieBurger CreateVeggieBurger();

}

public class ClassicRestaurant : IRestaurant
    {
        public IBeefBurger CreateBeefBurger()
        {
            return new BeefClassicBurger(1, "beef classic and stuff", false);
        }

        public IVeggieBurger CreateVeggieBurger()
        {
            return new VeggieClassicBurger(2, "veggies classic and stuff", true);
        }
    }

public class ItalianRestaurant : IRestaurant
  {
      public IBeefBurger CreateBeefBurger()
      {
          return new BeefItalianBurger(1, "beef and italian addons and stuff", true);
      }

      public IVeggieBurger CreateVeggieBurger()
      {
          return new VeggieItalianBurger(2, "veggies and italian addons and stuff",
true);
      }
  }

public class Program
 {
     public static void Main(string[] args)
     {
         IRestaurant classicFactory = new ClassicRestaurant();
         IRestaurant italianFactory = new ItalianRestaurant();

         IBeefBurger beefClassicBurger = classicFactory.CreateBeefBurger();
         beefClassicBurger.Prepare();

         IBeefBurger beefItalianBurger = classicFactory.CreateBeefBurger();
         beefItalianBurger.Prepare();
```

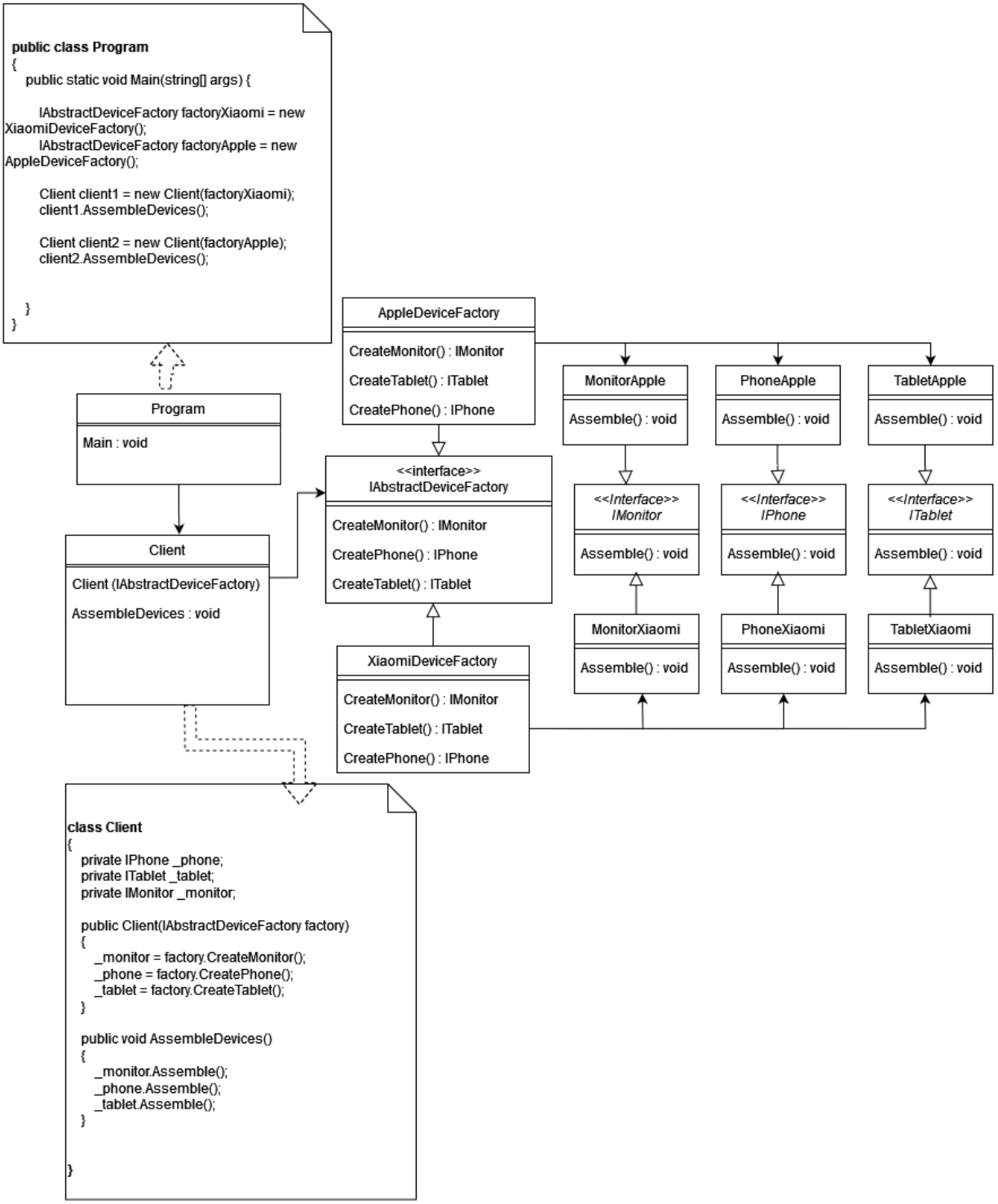
```
        IVeggieBurger veggieClassicBurger = classicFactory.CreateVeggieBurger();
        veggieClassicBurger.Prepare();

        IVeggieBurger veggieItalianBurger = classicFactory.CreateVeggieBurger();
        veggieClassicBurger.Prepare();

    }

}
```

- For each new product (BeefBurger, ChickenBurger) we should make a new interface
- We have concrete restaurants (concrete factories) that are able to produce different items of the same genre
- By creating two methods for creating each of the Products (Yoghurt and Cake) CreateBeefBurger and CreateVeggie we managed make it OCP compliant
- We can now create new flavors, new products and when modifying we just need to add new method here for each of new products
- Implementing new product however forces it to be implemented in all of the ConcreteFactories

Example 2

Let’s imagine we have two producers of the same products. For producers let them be Xiaomi and Apple companies and for products Tablets, Phones and Monitors. Each of the companies produces all three of these. How do we model this with an abstract factory?

Diagram



This example is better than previous one:

- We changed AbstractFactory to be an interface
- We have 2 lines of 3 different products
- Main creates factories and can inject different factories, because client class takes AbstractFactory
- We are OCP and SRP compliant
- The code is extensible

Code

```
namespace AbstractFactoryDifferentExample.Products
{
```

```
    interface IMonitor
    {
        public void Assemble();
    }
}
namespace AbstractFactoryDifferentExample.Products
{
    interface IPhone
    {
        public void Assemble();
    }
}
namespace AbstractFactoryDifferentExample.Products
{
    interface IPhone
    {
        public void Assemble();
    }
}

namespace AbstractFactoryDifferentExample.Products
{
    class MonitorApple : IMonitor
    {
        public void Assemble()
        {
            Console.WriteLine("Assembling Apple Monitor");
        }
    }
}

namespace AbstractFactoryDifferentExample.Products
{
    class MonitorXiaomi : IMonitor
    {
        public void Assemble()
        {
            Console.WriteLine("Assembling Xiaomi Monitor");
        }
    }
}

namespace AbstractFactoryDifferentExample.Products
{
    class PhoneApple : IPhone
    {
        public void Assemble()
        {
```

```
            Console.WriteLine("Assembling Apple Phone");
        }
    }
}

namespace AbstractFactoryDifferentExample.Products
{
    class PhoneXiaomi : IPhone
    {
        public void Assemble()
        {
            Console.WriteLine("Assembling Xiaomi Phone");
        }
    }
}

namespace AbstractFactoryDifferentExample.Products
{
    class TabletApple : ITablet
    {
        public void Assemble()
        {
            Console.WriteLine("Assembling Apple Tablet");
        }
    }
}

namespace AbstractFactoryDifferentExample.Products
{
    class TabletXiaomi : ITablet
    {
        public void Assemble()
        {
            Console.WriteLine("Assembling Xiaomi Tablet");
        }
    }
}
using AbstractFactoryDifferentExample.Products;

namespace AbstractFactoryDifferentExample.Factories
{
    interface IAbstractDeviceFactory
    {
        public IMonitor CreateMonitor();
        public IPhone CreatePhone();
        public ITablet CreateTablet();
    }
}
```

```
using AbstractFactoryDifferentExample.Products;

namespace AbstractFactoryDifferentExample.Factories
{
    public class AppleDeviceFactory : IAbstractDeviceFactory
    {
        IMonitor IAbstractDeviceFactory.CreateMonitor()
        {
            return new MonitorApple();
        }

        IPhone IAbstractDeviceFactory.CreatePhone()
        {
            return new PhoneApple();
        }

        ITablet IAbstractDeviceFactory.CreateTablet()
        {
            return new TabletApple();
        }
    }
}
using AbstractFactoryDifferentExample.Products;

namespace AbstractFactoryDifferentExample.Factories
{
    public class XiaomiDeviceFactory : IAbstractDeviceFactory
    {
        IMonitor IAbstractDeviceFactory.CreateMonitor()
        {
            return new MonitorXiaomi();
        }

        IPhone IAbstractDeviceFactory.CreatePhone()
        {
            return new PhoneXiaomi();
        }

        ITablet IAbstractDeviceFactory.CreateTablet()
        {
            return new TabletXiaomi();
        }
    }
}
using AbstractFactoryDifferentExample.Factories;
using AbstractFactoryDifferentExample.Products;
```

```
namespace AbstractFactoryDifferentExample
{
    class Client
    {
        private IPhone _phone;
        private ITablet _tablet;
        private IMonitor _monitor;

        public Client(IAbstractDeviceFactory factory)
        {
            _monitor = factory.CreateMonitor();
            _phone = factory.CreatePhone();
            _tablet = factory.CreateTablet();
        }

        public void AssembleDevices()
        {
            _monitor.Assemble();
            _phone.Assemble();
            _tablet.Assemble();
        }


    }
}

using AbstractFactoryDifferentExample.Factories;

namespace AbstractFactoryDifferentExample
{
    public class Program
    {
        public static void Main(string[] args) {

            IAbstractDeviceFactory factoryXiaomi = new XiaomiDeviceFactory();
            IAbstractDeviceFactory factoryApple = new AppleDeviceFactory();

            Client client1 = new Client(factoryXiaomi);
            client1.AssembleDevices();

            Client client2 = new Client(factoryApple);
            client2.AssembleDevices();


        }
    }
}
```

4. State Machine patterns

State machines

There are three common implementations of a state machine:

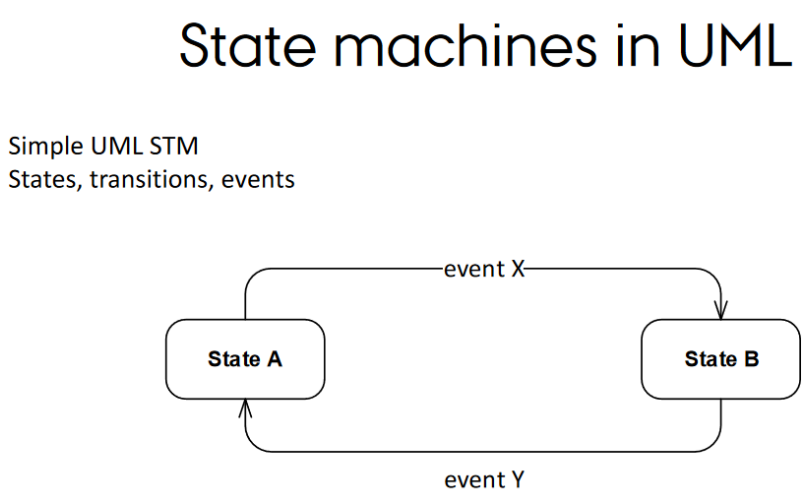
1. Switch-case
2. State/event tables
3. GoF State Pattern

• Each implementation maps the STM to code
• Each has advantages and drawbacks

Characteristics

A state machine (STM) is a model of a special kind of a system.At any time, the STM is in one of a finite set of states:

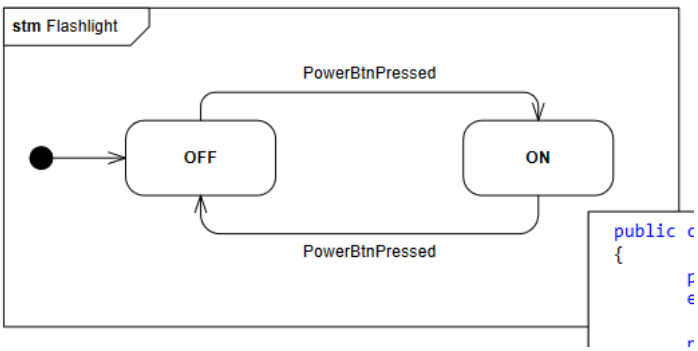
- A light switch: { ON | OFF }
- A process: {READY | RUNNING | BLOCKED }
- The STM can transition between states when events occur



Switch case variant

Example 1 - simple light switch

Diagram



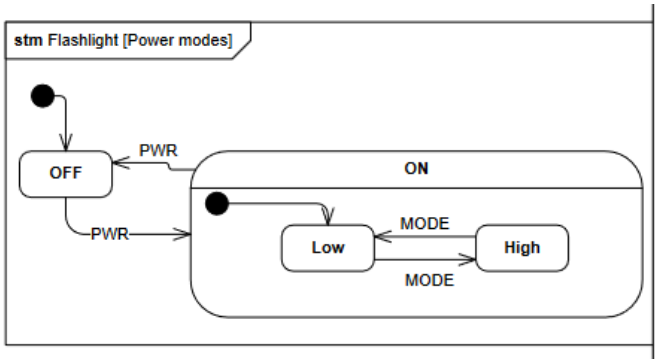
Code

```
namespace LightExampleSwitchCase
{
    enum LightState
    {
        On, Off
    }
    enum LightEvent
    {
        Pressed
    }

    class Light
    {
        private LightState _currentLightState;

        public Light()
        {
            _currentLightState = LightState.Off;
        }

        public void HandleEvent(LightEvent lightEvent)
        {
            switch (_currentLightState)
            {
                case LightState.Off:
                    HandleOffState(lightEvent);
                    break;
                case LightState.On:
                    HandleOnState(lightEvent);
                    break;
                default:
                    throw new Exception("Invalid state");

            }


        }

        private void HandleOnState(LightEvent lightEvent)
        {
            switch (lightEvent) {

                case LightEvent.Pressed:
                    Console.WriteLine("Turning the light off");
                    _currentLightState = LightState.Off;
```

```
                    break;
                default:
                    Console.WriteLine("The current state was On and the event was
TurnOn");
                    break;

            }

        }

        private void HandleOffState(LightEvent lightEvent)
        {
            switch (lightEvent)
            {
                case LightEvent.Pressed:
                    Console.WriteLine("Turning the light on");
                    _currentLightState = LightState.On;
                    break;
                default:
                    Console.WriteLine("The current state was Off and the event was
TurnOff");
                    break;

            }

        }


    }
}
namespace LightExampleSwitchCase
{
    class Program
    {
        static void Main(string[] args) {

            Light light = new Light();
            light.HandleEvent(LightEvent.Pressed);
            light.HandleEvent(LightEvent.Pressed);
            light.HandleEvent(LightEvent.Pressed);

        }
    }
}
```

Example 2 - complex light switch

Diagram



Code

```
namespace LightExampleSwitchCase
{

    enum LightIntensityState
    {
        High, Low
    }
    enum LightPowerState
    {
        On, Off
    }
    enum LightEvent
    {
        ClickPower,ClickMode
    }

    class Light
    {

        private LightPowerState _currentPowerState;
        private LightIntensityState _currentIntensityState;
        public Light()
        {
            _currentPowerState = LightPowerState.Off;
            _currentIntensityState = LightIntensityState.Low;
        }

        public void PrintCurrentState()
        {
            Console.WriteLine("Current state of light: ");
            Console.WriteLine("POWER: " + _currentPowerState.ToString());
            Console.WriteLine("INTENSITY: " + _currentIntensityState.ToString());
            Console.WriteLine();
```

```
        }
        public void HandleEvent(LightEvent lightEvent)
        {
            switch (_currentPowerState)
            {
                case LightPowerState.Off:
                    HandleOffState(lightEvent);
                    break;
                case LightPowerState.On:
                    HandleOnState(lightEvent);
                    break;
                default:
                    throw new Exception("Invalid _currentPowerState");

            }


        }


        private void HandleOnState(LightEvent lightEvent)
        {
            switch (lightEvent) {

                case LightEvent.ClickPower:
                    Console.WriteLine("Turning the light off and changing intensity
to Low");
                    _currentPowerState = LightPowerState.Off;
                    _currentIntensityState = LightIntensityState.Low;
                    break;
                case LightEvent.ClickMode:
                    Console.WriteLine("The light is on and you clicked MODE");
                    HandleIntensityState();
                        break;
                default:
                    Console.WriteLine("The currentPowerState was On and there is
error. If you see this message sth is wrong. Nothing happens...");
                    break;

            }

        }

        private void HandleIntensityState()
        {
            switch (_currentIntensityState)
            {
```

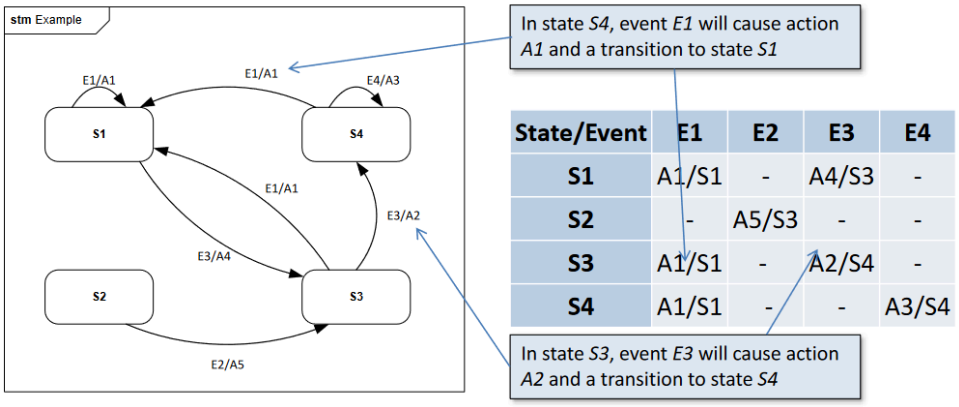
```
                case LightIntensityState.Low:
                    Console.WriteLine("Changing intensity to High");
                    _currentIntensityState = LightIntensityState.High;

                    break;
                case LightIntensityState.High:
                    Console.WriteLine("Changing intensity to Low");
                    _currentIntensityState = LightIntensityState.Low;
                    break;
                default:
                    throw new Exception("Invalid _currentIntensityState ");

            }
        }

        private void HandleOffState(LightEvent lightEvent)
        {
            switch (lightEvent)
            {
                case LightEvent.ClickPower:
                    Console.WriteLine("Turning the light on");
                    _currentPowerState = LightPowerState.On;
                    break;

                default:
                    Console.WriteLine("The currentPowerState was Off and the event
was not power On. Nothing happens... ");
                    break;

            }

        }




    }
}
namespace LightExampleSwitchCase
{
    class Program
    {
        static void Main(string[] args) {

            Light light = new Light();
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickMode);
            light.PrintCurrentState();
```

```
            light.HandleEvent(LightEvent.ClickPower);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickMode);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickMode);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickPower);
            light.PrintCurrentState();

            light.HandleEvent(LightEvent.ClickPower);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickMode);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickPower);
            light.PrintCurrentState();



        }
    }
}
```

State/Event Tables variant

Example 1 - simple light switch

Diagram



State/Event	E1	E2	E3	E4
S1	A1/S1	-	A4/S3	-
S2	-	A5/S3	-	-
S3	A1/S1	-	A2/S4	-
S4	A1/S1	-	-	A3/S4

General diagram and table from slides

State/Event	***Button Pressed (E1)***
PowerOff (S1)	*TurnOn (A1/S2)*
PowerOn (S2)	*TurnOff (A2/S1)*

Table for simple switch for turning on light.

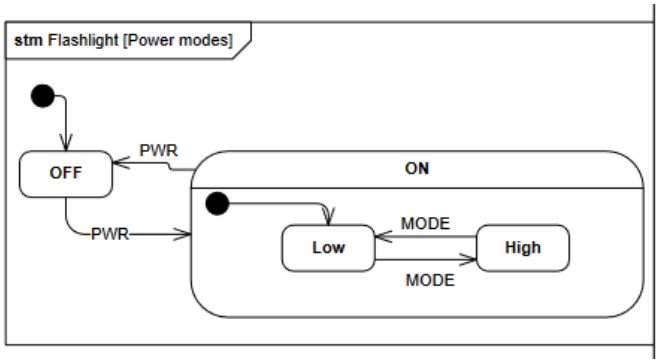
Code

```
namespace LightExampleStateEventTables
{
    enum LightState
    {
        On, Off
    }
    enum LightEvent
    {
        Pressed
    }
    class Light
    {


        private LightState _currentLightState;
        private Dictionary<(LightState, LightEvent), Action> _stateTransitionTable;
        public Light()
        {
            _currentLightState = LightState.Off;
            _stateTransitionTable = new Dictionary<(LightState, LightEvent), Action>
            {
                {(LightState.Off,LightEvent.Pressed), TurnLightOn },
                {(LightState.On, LightEvent.Pressed), TurnLightOff}

            };
        }

        public void HandleEvent(LightEvent lightEvent)
        {
            (LightState, LightEvent) key = (_currentLightState, lightEvent);
            if (_stateTransitionTable.TryGetValue(key, out var transition))
            {
                transition.Invoke();
            }
            else
            {
                throw new Exception("Invalid state transition");
            }
        }
        private void TurnLightOn()
        {
            Console.WriteLine("Turning the light On");
            _currentLightState = LightState.On;
        }
```

```
        private void TurnLightOff()
        {
            Console.WriteLine("Turning the light Off");
            _currentLightState = LightState.Off;
        }
    }

    }

namespace LightExampleStateEventTables
{
    class Program
    {
        static void Main(string[] args) {

            Light light = new Light();
            light.HandleEvent(LightEvent.Pressed);
            light.HandleEvent(LightEvent.Pressed);
            light.HandleEvent(LightEvent.Pressed);
        }
    }
}
```

Example 2 - complex StateEventTables

Diagram



We can shrink the states to one with 4 options (S2 will not be considered a valid state)

State/Event	***Power Button Pressed (E1)***	***Mode Button Pressed (E2)***
PowerOff with Low Intensity (S1)	*A1/S3*	*-*
PowerOff with High Intensity (S2)	*-*	*-*
PowerOn with Low Intensity (S3)	*A2/S1*	*A3/S4*
PowerOn with High Intensity (S4)	*A2/S3*	*A4/S3*

A1 - Make Power on
A2 - Make Power off

A3 - Switch to High intensity
A4 - Switch to Low intensity

Code

```
namespace LightComplicatedExampleStateEventTables
{
    enum LightState
    {
        HighOn, LowOn, Off
    }

    enum LightEvent
    {
        ClickPower, ClickMode
    }

    class Light
    {
        private LightState _currentLightState;
        private Dictionary<(LightState, LightEvent), Action> _stateTransitionTable;
        public Light()
        {
            _currentLightState = LightState.Off;
            _stateTransitionTable = new Dictionary<(LightState, LightEvent), Action>
            {
                { (LightState.Off,LightEvent.ClickPower), TurnLightOn},
                { (LightState.LowOn,LightEvent.ClickPower), TurnLightOff},
                { (LightState.HighOn,LightEvent.ClickPower), TurnLightOff},
                { (LightState.Off,LightEvent.ClickMode), ()=>{
Console.WriteLine("Cannot change mode when off. Nothing happens..."); } },
                { (LightState.LowOn,LightEvent.ClickMode),TurnIntensityUp },
                { (LightState.HighOn,LightEvent.ClickMode),TurnIntensityDown },
            };
        }

        public void HandleEvent(LightEvent lightEvent)
        {
            (LightState, LightEvent) key = (_currentLightState, lightEvent);
            if (_stateTransitionTable.TryGetValue(key, out var transition))
            {
                transition.Invoke();
            }
            else
            {
                throw new Exception("Invalid state transition");
            }
        }
```

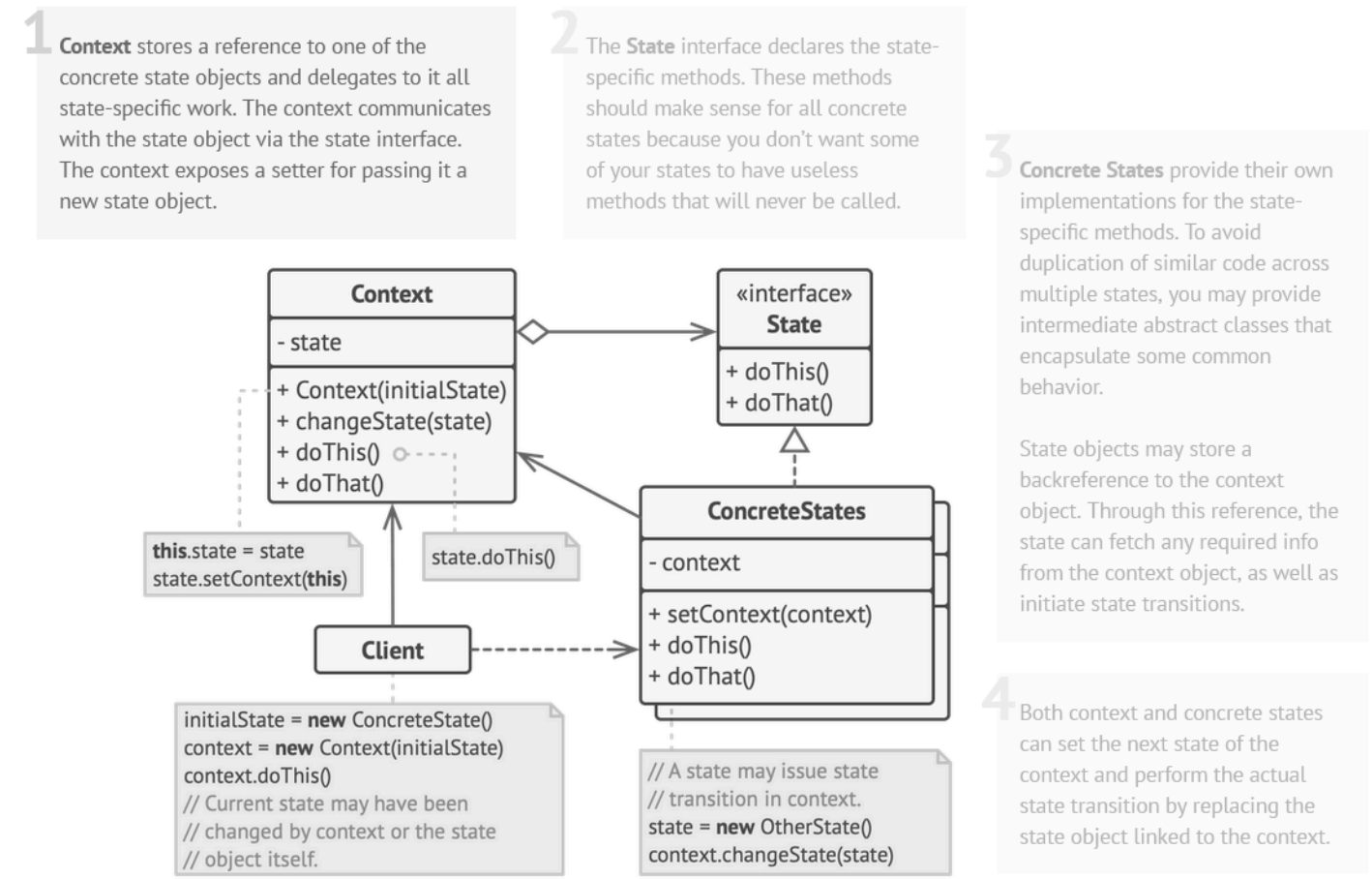
```
        public void PrintCurrentState()
        {
            Console.WriteLine("Current state of light: ");
            Console.WriteLine("State: " + _currentLightState.ToString());

            Console.WriteLine();

        }
        private void TurnLightOn()
        {
            Console.WriteLine("Turning the light On");
            _currentLightState = LightState.LowOn;
        }
        private void TurnLightOff()
        {
            Console.WriteLine("Turning the light Off");
            _currentLightState = LightState.Off;
        }

        private void TurnIntensityUp()
        {
            Console.WriteLine("Turning the intensity Up");
            _currentLightState = LightState.HighOn;
        }
        private void TurnIntensityDown()
        {
            Console.WriteLine("Turning the intensity Down");
            _currentLightState = LightState.LowOn;
        }
    }
}


namespace LightComplicatedExampleStateEventTables
{
    class Program
    {
        public static void Main(string[] args) {

            Light light = new Light();
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickMode);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickPower);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickMode);
            light.PrintCurrentState();
```

```
            light.HandleEvent(LightEvent.ClickMode);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickPower);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickPower);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickMode);
            light.PrintCurrentState();
            light.HandleEvent(LightEvent.ClickPower);
            light.PrintCurrentState();

        }
    }
}
```

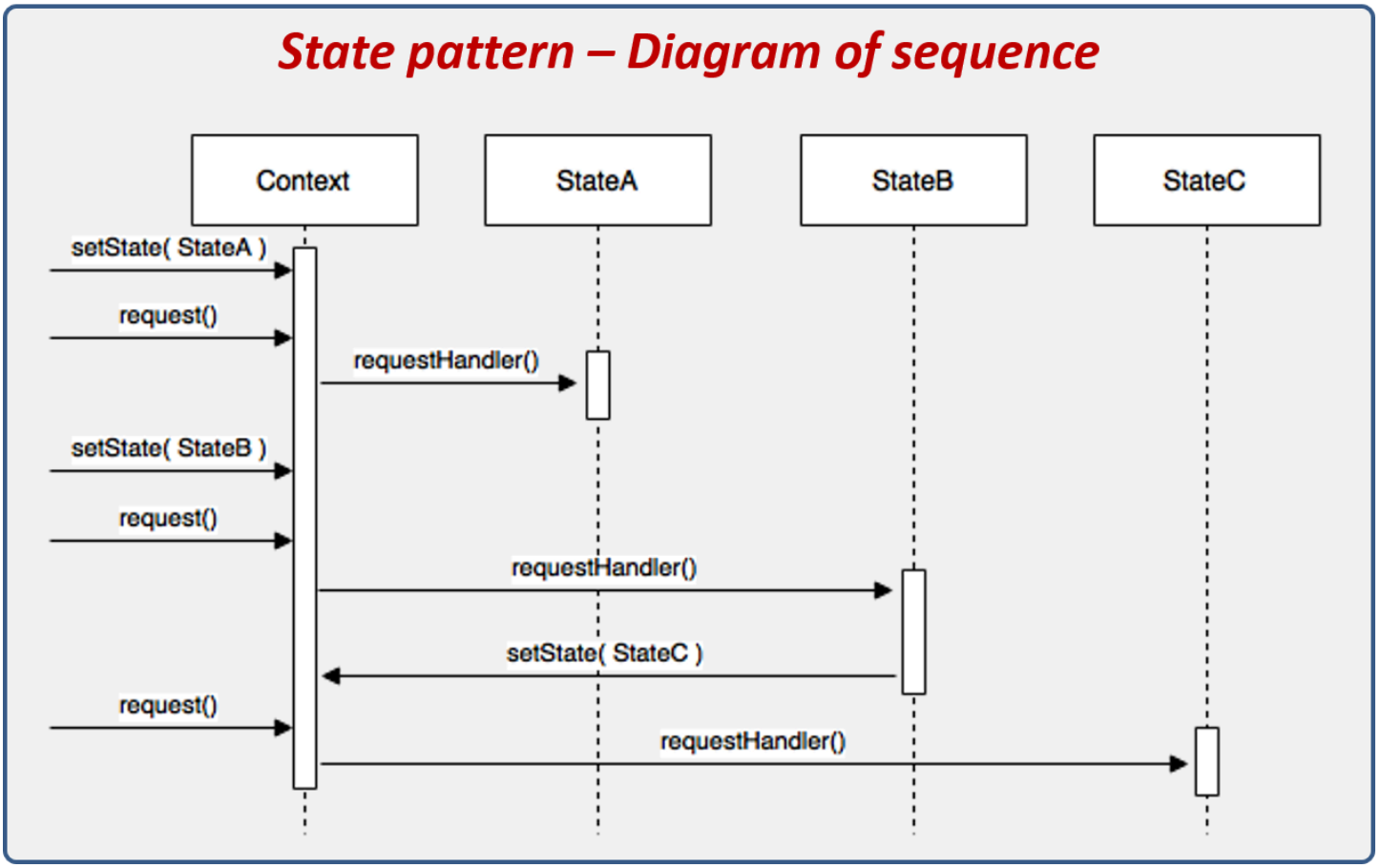
State pattern variant

The GoF State Pattern is especially neat when used for complex (nested, orthogonal) state machines.

Class Diagram

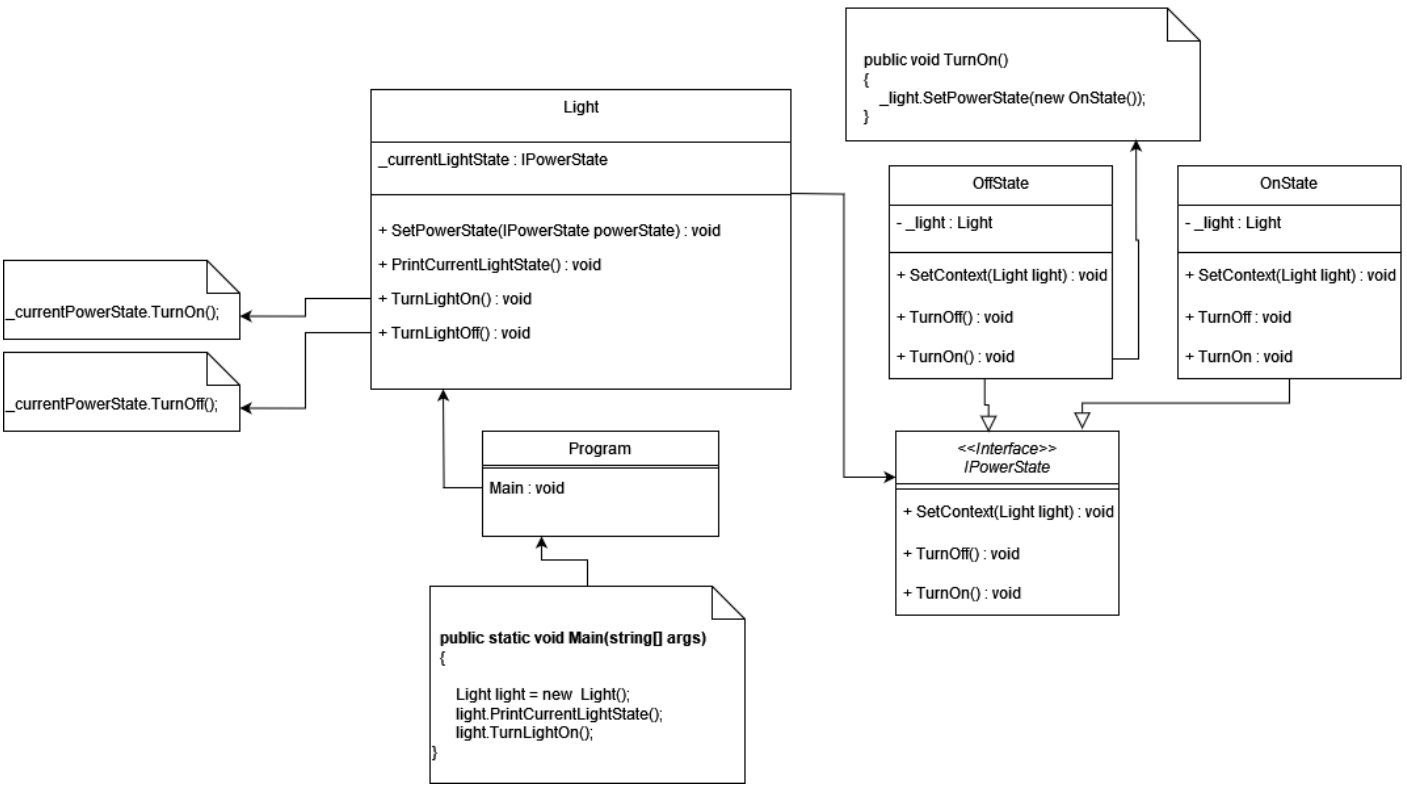


Sequence Diagram



Example 1 - simple light switch

Diagram



We have the Light class which is the context that depends on the IPowerState interface which is an interface for our light On/Off concrete states. The OffState and OnState classes use the Light context, and the

SetContext method is invoked in the Light constructor and in SetPowerState() (because we return a new ConcreteState object and we need to reassign the context).

User workflow:

1. Create the Light object
2. The context gets set to the ConcreteState object
3. Invoke TurnLightOn() method
4. TurnLightOn() invokes _currentLightState.TurnOn(); in the Light object
5. The Light object invokes the method SetPowerState with new ConcreteState object _light.SetState(new OnState());
6. In the Light class the passed object gets assigned to the _currentPowerState
7. The context gets reassigned in the Light class to ‘this’, because the reference changed due to the new object creation

Code

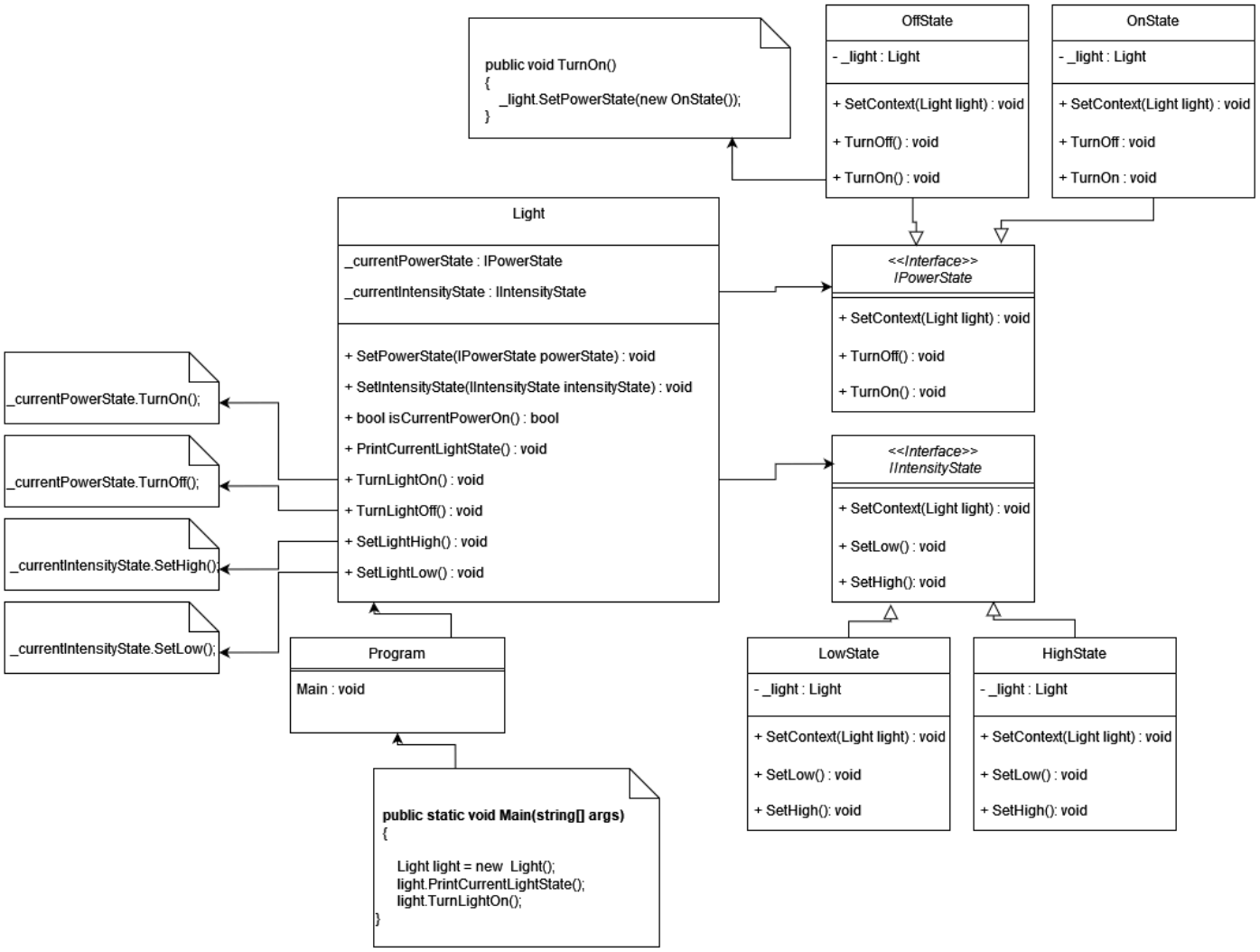
```
namespace LightExampleGofState
{
    interface ILightState
    {
        public void SetContext(Light light);
        public void TurnOn();
        public void TurnOff();

    }
}

namespace LightExampleGofState
{
    class OffState : ILightState
    {
        private Light _light;
        public void SetContext(Light light)
        {
            _light = light;
        }

        public void TurnOff()
        {
            Console.WriteLine("This state is already Off. Nothing happens..");
        }

        public void TurnOn()
        {
            Console.WriteLine("Turning light On");
            _light.SetState(new OnState());
        }
    }
}
```

```
namespace LightExampleGofState
{
    class OnState : ILightState
    {
        private Light _light;
        public void SetContext(Light light)
        {
            _light = light;
        }

        public void TurnOff()
        {
            Console.WriteLine("Turning light Off");
            _light.SetState(new OffState());

        }

        public void TurnOn()
        {
            Console.WriteLine("The state is already on. Nothing happens...");

        }
    }
}


namespace LightExampleGofState
{

    class Light
    {
        private ILightState _currentLightState;

        public Light()
        {
            _currentLightState = new OffState();
            _currentLightState.SetContext(this);
        }

        public void SetState(ILightState state)
        {
            Console.WriteLine($"Invoking SetState -> Changing state to
{state.GetType().Name}");
            _currentLightState = state;
            _currentLightState.SetContext(this);
        }
```

```
        public void TurnLightOn()
        {
            _currentLightState.TurnOn();
        }

        public void TurnLightOff() {
             _currentLightState.TurnOff();
        }

        public void PrintCurrentLightState()
        {
           Console.WriteLine("Current state is: "+
_currentLightState.GetType().Name);
           Console.WriteLine();
        }


    }
}

namespace LightExampleGofState
{
    class Program
    {
        public static void Main(string[] args) {

            Light light = new Light();
            light.PrintCurrentLightState();
            light.TurnLightOn();
            light.PrintCurrentLightState();
            light.TurnLightOn();
            light.PrintCurrentLightState();
            light.TurnLightOff();
            light.PrintCurrentLightState();
            light.TurnLightOff();
            light.PrintCurrentLightState();


        }
    }
}
```

Example 2 - complex light switch

Diagram



Now we have two states - one for power and one for light intensity regulation. Not much has changed with adding another state except for:

1. Methods in the Light class to set intensity to High and Low
2. Method isCurrentPowerOn() in the Light class that is used by LowState and HighState classes to check conditionally whether the Light is powered and only if it is change state to Low/High

Code

```
namespace LightComplicatedExampleGofState.PowerState
{
    interface IPowerState
    {
        public void SetContext(Light light);
        public void TurnOn();
        public void TurnOff();
    }
}
namespace LightComplicatedExampleGofState.PowerState
{
```

```
    class OffState : IPowerState

    {

        private Light _light;
        public void SetContext(Light light)
        {
            _light = light;
        }


        public void TurnOff()
        {
            Console.WriteLine("This state is already Off. Nothing happens..");
        }

        public void TurnOn()
        {
            _light.SetPowerState(new OnState());
        }
    }
}
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LightComplicatedExampleGofState.PowerState
{
    class OnState : IPowerState
    {

        private Light _light;
        public void SetContext(Light light)
        {
            _light = light;
        }

        public void TurnOff()
        {
            _light.SetPowerState(new OffState());
        }

        public void TurnOn()
        {
            Console.WriteLine("This state is already On. Nothing happens..");
        }
```

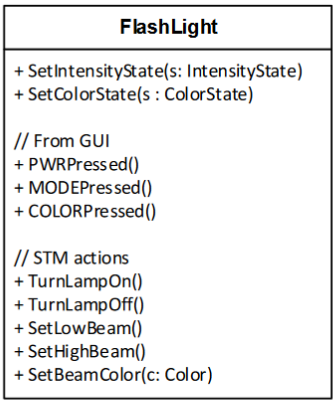
```
    }
}
namespace LightComplicatedExampleGofState.IntensityState
{
    interface IIntensityState
    {
        public void SetContext(Light light);

        public void SetHigh();
        public void SetLow();
    }
}
namespace LightComplicatedExampleGofState.IntensityState
{
    class HighState : IIntensityState
    {
        private Light _light;
        public void SetContext(Light light)
        {
            _light = light;
        }

        public void SetHigh()
        {
            if (!_light.isCurrentPowerOn())
            {
                Console.WriteLine("Power is not on, cannot modify intensity");
                return;
            }
            Console.WriteLine("Intensity state is already set High. Nothing has
changed...");
        }

        public void SetLow()
        {
            if (!_light.isCurrentPowerOn())
            {
                Console.WriteLine("Power is not on, cannot modify intensity");
                return;
            }
            _light.SetIntensityState(new LowState());
        }
    }
}
namespace LightComplicatedExampleGofState.IntensityState
{
    class LowState : IIntensityState
    {
```

```
        private Light _light;
        public void SetContext(Light light)
        {
            _light = light;
        }

        public void SetHigh()
        {

            {
                Console.WriteLine("Power is not on, cannot modify intensity");
                return;
            }
            _light.SetIntensityState(new HighState());
        }

        public void SetLow()
        {
            if (!_light.isCurrentPowerOn())
            {
                Console.WriteLine("Power is not on, cannot modify intensity");
                return;
            }
            Console.WriteLine("Intensity state is already set Low. Nothing has
changed...");
        }
    }
}


using LightComplicatedExampleGofState.IntensityState;
using LightComplicatedExampleGofState.PowerState;

namespace LightComplicatedExampleGofState
{
    class Light
    {
        IPowerState _currentPowerState;
        IIntensityState _currentIntensityState;
        public Light() {

            _currentPowerState = new OffState();
            _currentIntensityState = new LowState();

            _currentPowerState.SetContext(this);
            _currentIntensityState.SetContext(this);
```

```
        }

        public void SetPowerState(IPowerState powerState)
        {
            Console.WriteLine($"Invoking SetPowerState -> Changing state to
{powerState.GetType().Name}");
            _currentPowerState = powerState;
            _currentPowerState.SetContext(this);
        }

        public void SetIntensityState(IIntensityState intensityState)
        {
            Console.WriteLine($"Invoking SetState -> Changing state to
{intensityState.GetType().Name}");
            _currentIntensityState = intensityState;
            _currentIntensityState.SetContext(this);
        }

        public bool isCurrentPowerOn()
        {
            return _currentPowerState is OnState;
        }

        public void PrintCurrentLightState()
        {
            Console.WriteLine("Current powerState is: " +
_currentPowerState.GetType().Name);
            Console.WriteLine("Current intensityState is: " +
_currentIntensityState.GetType().Name);
            Console.WriteLine();
        }


        public void TurnLightOn()
        {
            _currentPowerState.TurnOn();
        }

        public void TurnLightOff() {
            _currentPowerState.TurnOff();

        }

        public void SetLightHigh()
        {
            _currentIntensityState.SetHigh();

        }
```

```
        public void SetLightLow()
        {
            _currentIntensityState.SetLow();

        }

    }
}



namespace LightComplicatedExampleGofState
{
    class Program
    {
        public static void Main(string[] args)
        {

            Light light = new Light();
            light.PrintCurrentLightState();
            light.TurnLightOn();
            light.PrintCurrentLightState();
            light.SetLightHigh();
            light.PrintCurrentLightState();
            light.SetLightHigh();
            light.PrintCurrentLightState();
            light.SetLightLow();
            light.PrintCurrentLightState();
            light.TurnLightOff();
            light.PrintCurrentLightState();
            light.TurnLightOff();
            light.PrintCurrentLightState();



        }
    }
}
```

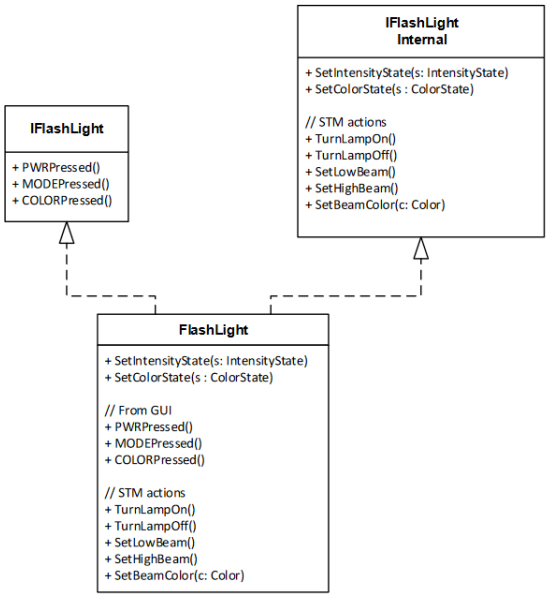
The evil client



What if a client to the context writes code which sets the state directly?

Or turns on/off the lamp, and thus bypasses the statemachine?

ISP to the rescue



Segregate the interface to the context.

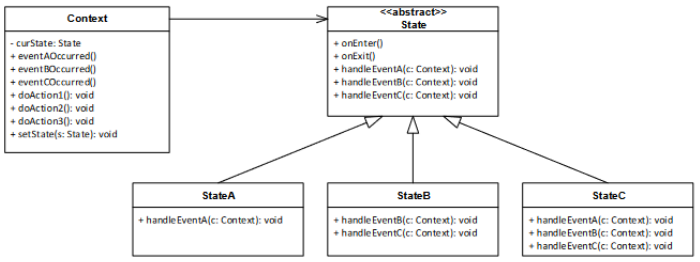
One interface to be used by clients of the context.

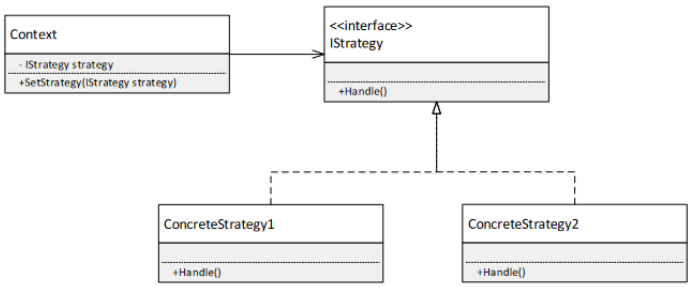
Another interface to be used by the state machine implementation

These slides are about adding two interfaces to Light state class in order to not let the user modify the state directly. The methods in IFlashLightInternal need to be public, because they are used by ConcreteStates, but you can give the user only relevant methods (here IFlashLight) that are used to manipulate so he doesn't accidentally set the whole state.

State vs. Strategy

The structure of GoF State and GoF Strategy are quite similar.





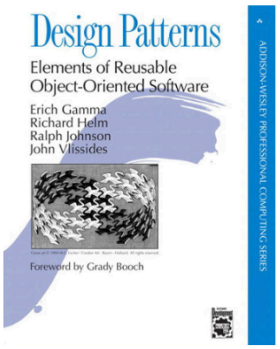
State vs. Strategy

GoF State

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

GoF Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



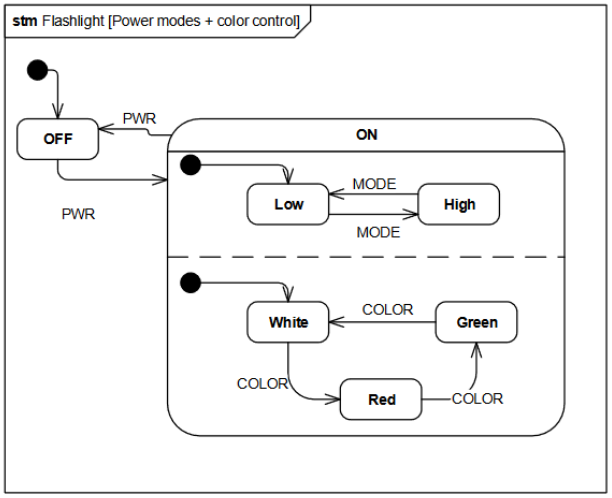
Orthogonal states problem

Orthogonal states

- Two implementation strategies:

  1) Collapse into a single state machine: Low-White, Low-Green, Low-Red, High-White, High-Green, High-Red.

  2) Create two separate state machines and let the context hold a reference to both.



States shrinking is presented here in the states table I made.

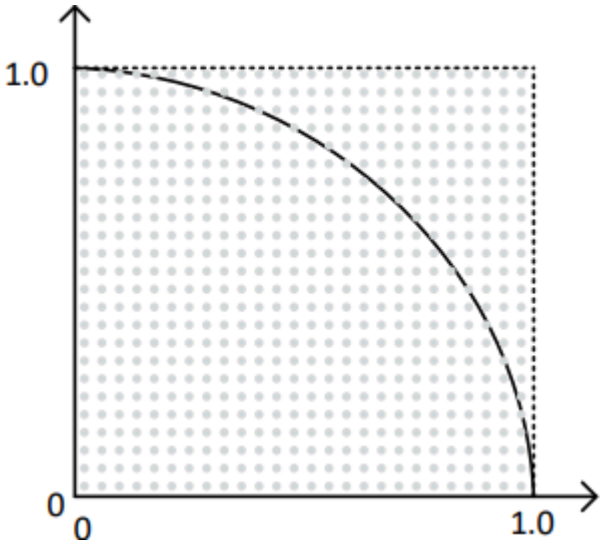
5. Parallel Aggregation + MapReduce

Parallel Aggregation

When you compute a large number of inputs and you get a large number of outputs, sometimes you don't need to store all of them - their sum or mean is just enough. Aggregation is the action of collecting items to form a total quantity like such.

In C# (but also generally) there are some ways to perform computation required for obtaining a large number of results to aggregate. Let's have an example of calculating pi with Dartboard algorithm.

The idea is that you draw an arch (¼ of a circle) with a radius=1. then we put it in a square and draw a points one close to another with an equal spacing, so let's say its *k x k* points. Like that:

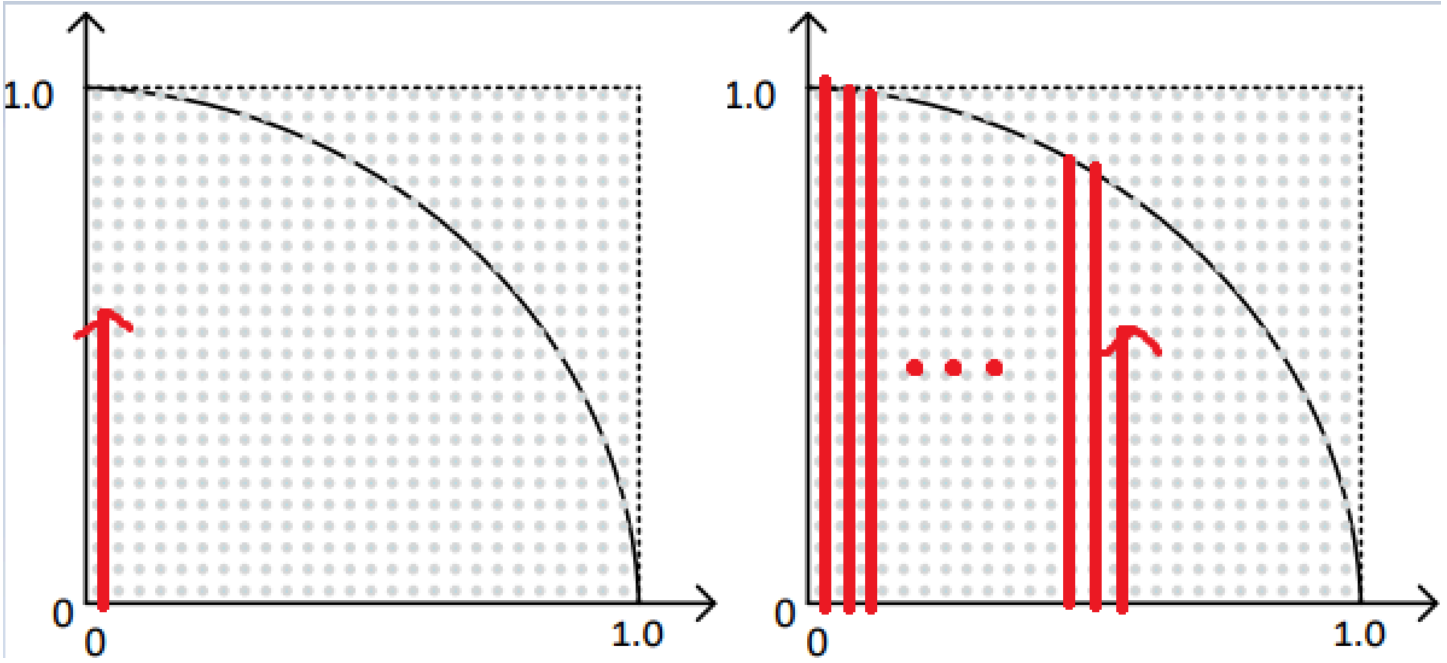


$$\pi \cong \frac{4 \cdot n_{circle}}{n}$$

Then, we check which points are within the radius and which are not. Next, we just put it in the formula above. Note that *n = k*k.*
The algorithm (pseudocode) goes as follows:

```
sumInside = 0
for i in 0…k:
        for j in 0…k:
                if not isInside(point(i, j)): break
                else: sumInside++

pi = 4*sumInside / k*k
```

Where *isInside(Point p)* checks if the point is inside the radius.
In this algorithm we're basically going column by column - left to right, bottom to top until we reach the first point that's out of the radius scope.



The aggregation is here the *sumInside++* part, where we just sum the number of points inside the radius, instead of storing all their values.

Approach 1:

The most simple way would be just to run the algorithm as-is. This approach here is however linear, because all iterations are performed sequentially - so may not be time-wise optimal.

Observation: the calculations for each i (column) are independent.

Approach 2 (intermediate step):

We should try to parallelize it (Parallel.For) and just aggregate the results to a common variable (remembering about locking it).

```
sumInside = 0
Parallel.For i in 0…k:
        for j in 0…k:
                if not isInside(point(i, j)): break
                else: lock(sumInside++)

pi = 4*sumInside / k*k
```

Now, we parallelized the calculations - so a thread just takes a column (i) and makes calculations for each point in this column. Every time a point is inside the radius, there is a lock put on the common summing variable - which results in many, many lockings. This doesn't seem too efficient.

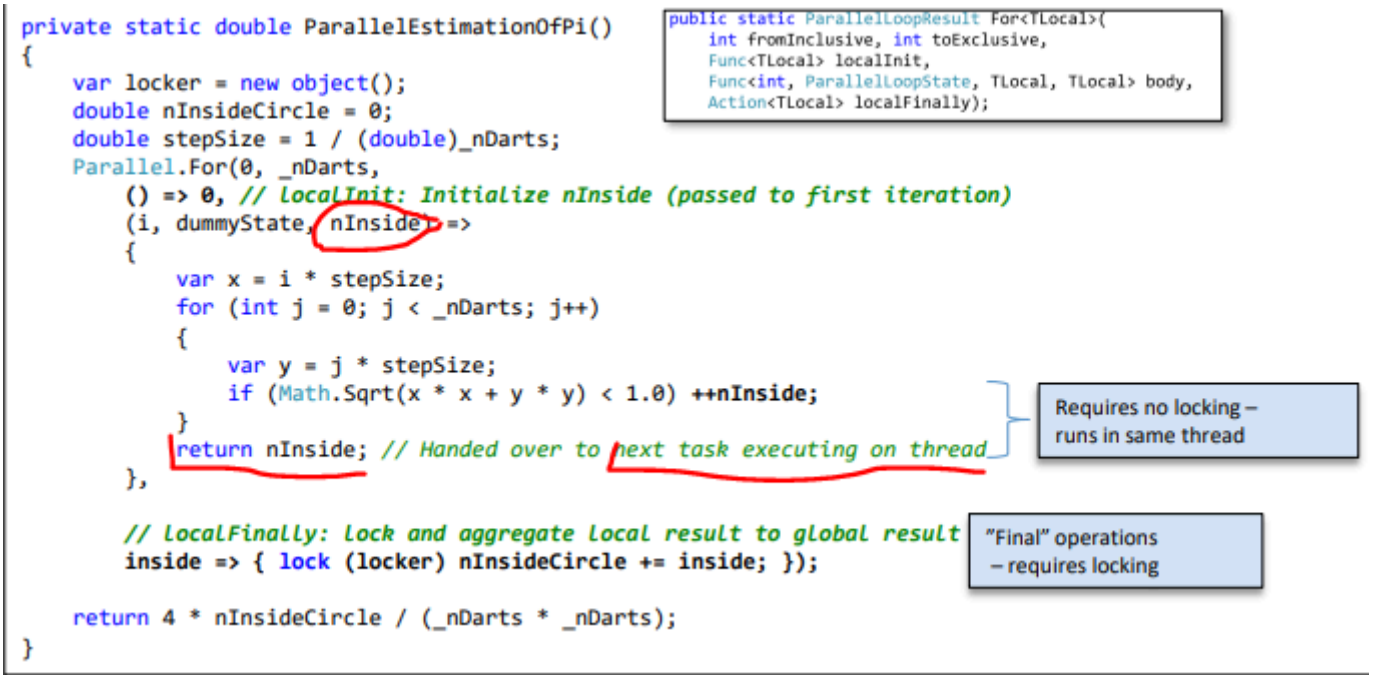
So maybe, for operations done in the scope of one thread, we should create a local aggregation variable and just add it to a common summing variable when the thread exits?

Approach 3:

So, we could now for each column (i) done in parallel create a local summing variable and just add it to a common summing variable after computing the whole column - that would noticeably reduce the number of locks (from *sumInside* to *i* times):

```
sumInside = 0
Parallel.For i in 0…k:
        localSum = 0
        for j in 0…k:
                if not isInside(point(i, j)): break
                else: localSum++
        lock(sumInside) += localSum

pi = 4*sumInside / k*k
```

…but in C# there's even better option. There's an overloaded Parallel.For version, that allows for passing some return argument from one iteration to the next done on the same thread - so in that case you only need to use lock *numThreads* times.



Approach 3+:

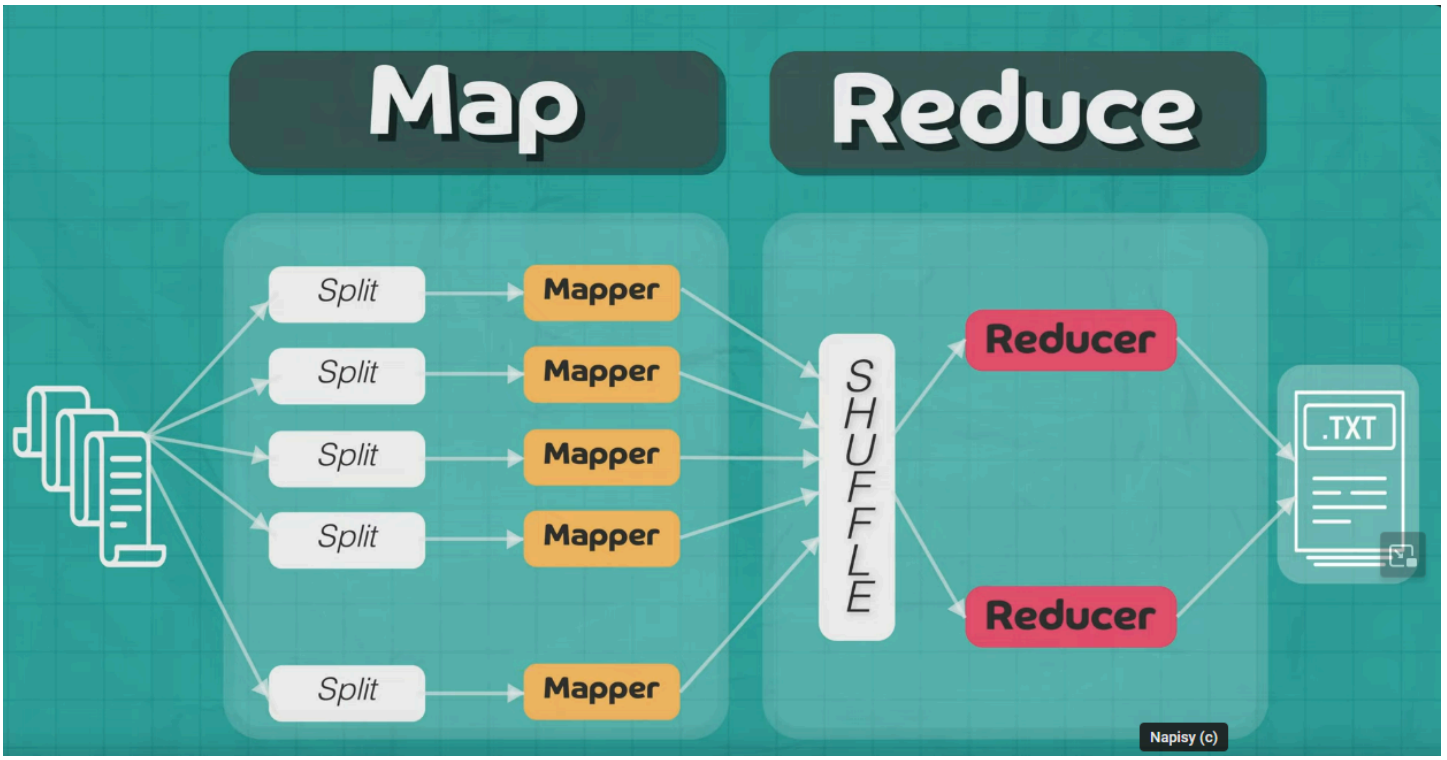
We can go even further with improving this. Currently, the partitioning is based upon columns, and they are distributed in some unspecified manners. Maybe ve could do some supersmart partitioning, so we could make an optimal division of columns between threads?
In C# there's an overloaded Parallel.ForEach using a *Partitioner* to do the splitting for optimal chunks for us.

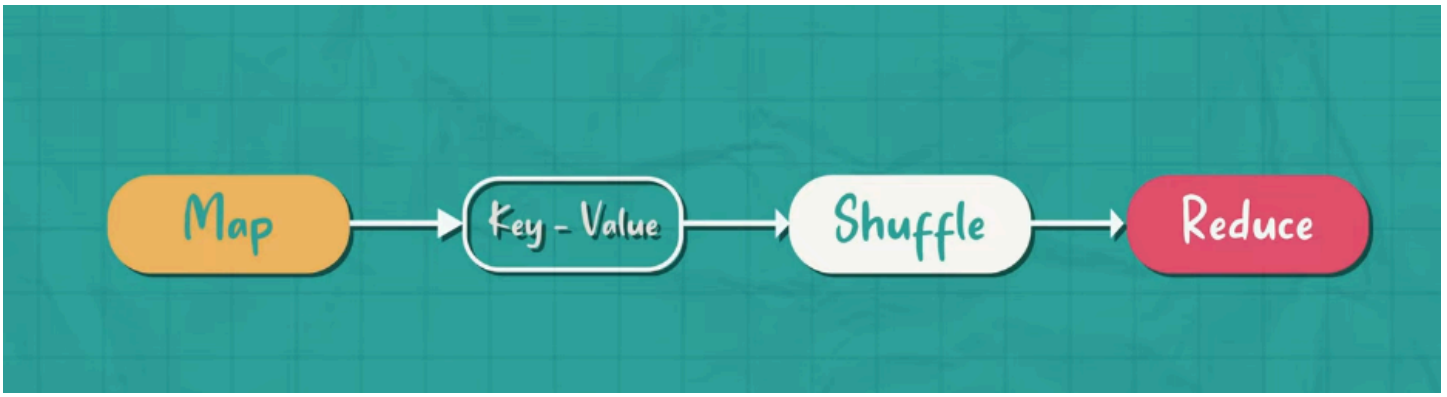
MapReduce

https://youtu.be/cHGaQz0E7AU

MapReduce:

- Map transforms data into key-value pairs
- Reduce - reduces the data values





1. We have distributed file system shared by nodes with data living on them
2. We do not move data, map works locally on every node
3. Then we try to merge similar data from different nodes (Shuffle, reorganization)
4. Input to reduce functions

Example - unique word counting

We have 2 input files with sentences:

- this is an apple - 1st file
- apple is red in color - 2nd file

We distribute these files to two nodes. On the nodes mapper works:

First:

- this:1
- is:1
- an:1
- apple:1

Second:

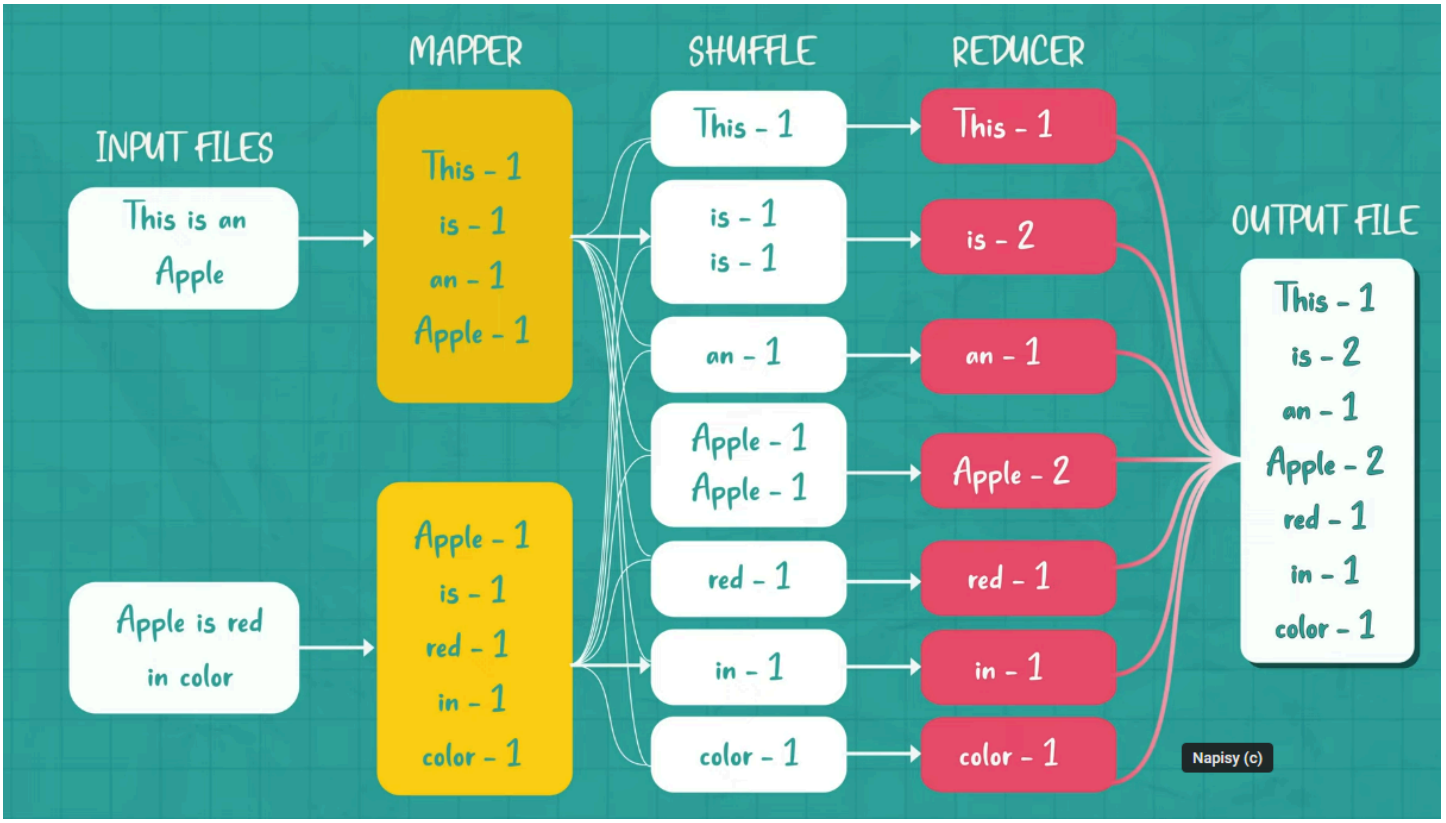
- apple:1
- is:1
- red:1
- in:1

- color:1

Shuffle

Grouping the same keys from different nodes.

Reducer
Applying reduction function SUM to count the same keys

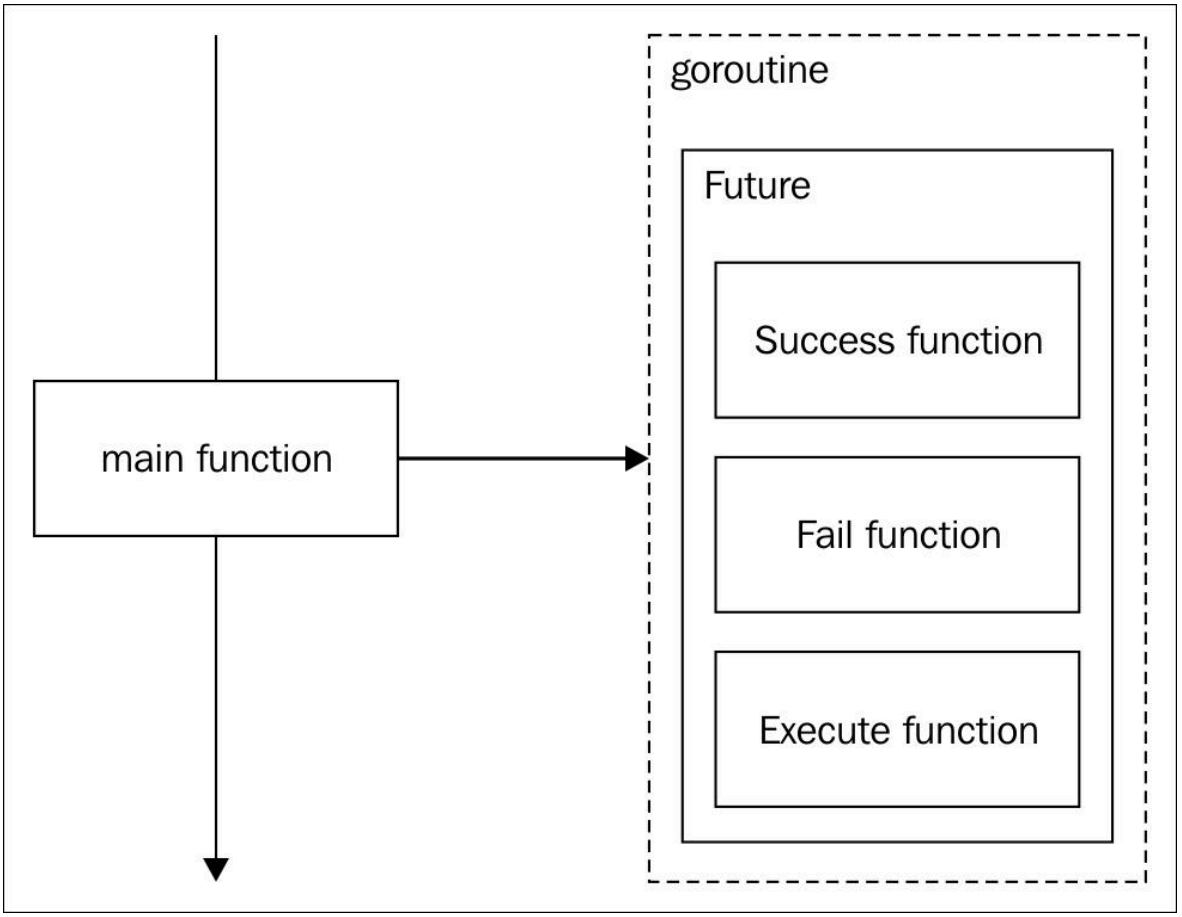


6. Futures + Pipelines

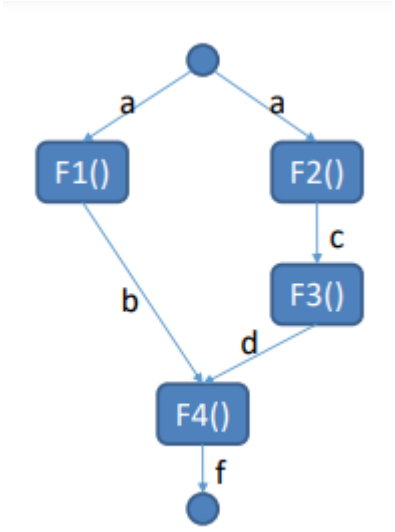
Futures

A future is a stand-in for a computational result that is initially unknown but becomes available at a later time. Futures are used when we want to parallelize code with data dependencies, so when the later task needs to take as an input some computed value from the previous task.

Diagram



Parallel tasks -> async actions (no return value)
Futures -> async functions (returning a value)



For such flow graph, the parallelized implementation could be:

Code

```
class Program
{
    static async Task Main()
    {
        var a = "A";
```

```
        // Compute F1(a) asynchronously
        Task<int> futureB = Task.Run(() => F1(a));

        // Compute F2(a) synchronously
        var c = F2(a);

        // Compute F3(c) synchronously
        var d = F3(c);

        // Wait for Task to complete and then compute F4(result, d) synchronously
        int resultB = await futureB;
        var f = F4(resultB, d);

        // Print the final result
        Console.WriteLine("FINAL RESULT {0}",f);
    }

    static int F1(string input)
    {
        Console.WriteLine($"Start: Computing F1({input}) asynchronously");
        // Simulate a time-consuming operation
        Task.Delay(3500).Wait();
        Console.WriteLine($"Stop: Computing F1({input}) asynchronously");
        return input.Length;
    }

    static string F2(string input)
    {
        Console.WriteLine($"Start: Computing F2({input}) synchronously");
        Task.Delay(900).Wait();
        Console.WriteLine($"Stop: Computing F2({input}) synchronously");
        return input.ToUpper();
    }

    static double F3(string input)
    {
        Console.WriteLine($"Start: Computing F3({input}) synchronously");
        Task.Delay(900).Wait();
        Console.WriteLine($"Stop: Computing F3({input}) synchronously");
        return input.Length * 1.5;
    }

    static string F4(int resultB, double resultD)
    {
        Console.WriteLine($"Start: Computing F4({resultB}, {resultD})
synchronously");

        return $"Result: {resultB + resultD:F2}";
```

```
    }
}
```

By the time we query futureB for the result (in order to compute var f):
• If task running F1() has already finished: futureB.Result is ready and immediately returned.
• If task running F1() is running but has not yet finished: Calling thread **blocks** until futureB.Result is available.
• If task running F1() hasn't started yet: The task will be executed inline in the current thread context, if possible.

Pipelines

The Pipeline pattern uses parallel tasks and concurrent queues to process a sequence of input values. Each task implements a stage of the pipeline, and the queues act as buffers that allow the stages of the pipeline to execute concurrently, even though the values are processed in order - so the output of *i-th* stage is an input for *i+1-th* stage (and so on).
You can think of software pipelines as analogous to assembly lines in a factory, where each item in the assembly line is constructed in stages. The partially assembled item is passed from one assembly stage to another. The outputs of the assembly line occur in the same order as that of the inputs.

A pipeline is composed of a series of producer/consumer stages, each one depending on the output of its predecessor. Pipelines allow you to use parallelism in cases where there are too many dependencies to use a parallel loop - so tasks cannot be performed independently.

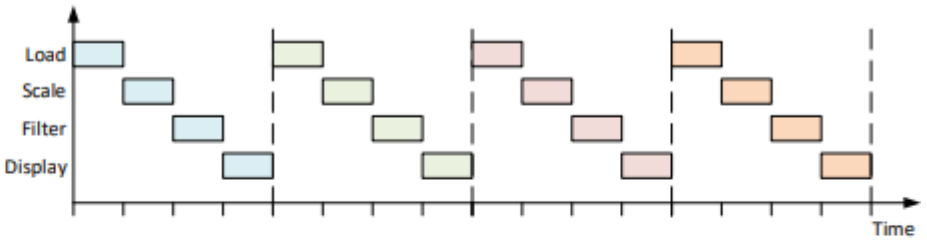
There are many ways to use pipelines. Pipelines are often useful when the data elements are received from a real-time event stream, such as values on stock ticker tapes, user-generated mouse click events, and packets that arrive over the network. Pipelines are also used to process elements from a data stream, as is done with compression and encryption, or processing streams of video frames. In all of these cases, it's important that the data elements are processed in sequential order.

In the .NET Framework, the buffers that connect stages of a software pipeline are usually based on the **BlockingCollection<T>** class, which is a thread-safe collection class providing the following features:

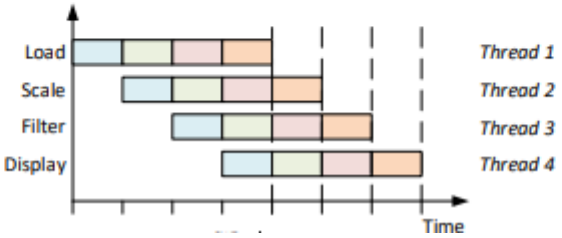
- An implementation of the Producer-Consumer pattern.
- Concurrent adding and taking of items from multiple threads.
- Optional maximum capacity.
- Insertion and removal operations that block when collection is empty or full.
- Insertion and removal "try" operations that do not block or that block up to a specified period of time.
- Encapsulates any collection type that implements IProducerConsumerCollection<T>
- Cancellation with cancellation tokens.
- Two kinds of enumeration with *foreach*
    - Read-only enumeration.
    - Enumeration that removes items as they are enumerated (**GetConsumingEnumerable()**). Using GetConsumingEnumerable() is important, so no two threads consume the same item.

So the idea of pipeline pattern is to divide one, big job (like some complex image processing) into sequential tasks that treat output of former as input to latter. This introduces parallelization, as each task may now (and even needs to) work on a different part of data.

Sequential
(4 images)

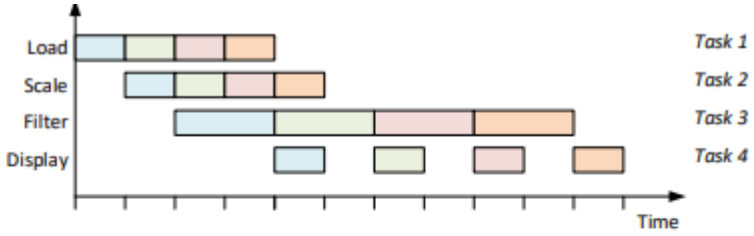


Pipelined –
throughput * 4

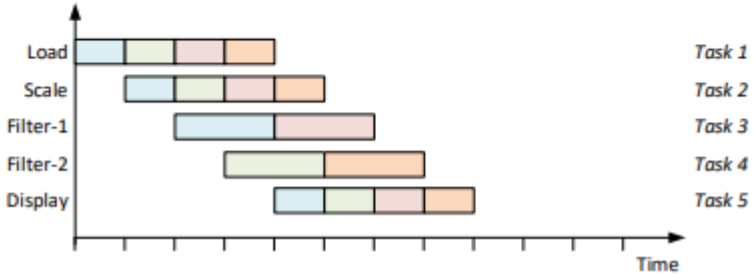


Now, it is just perfect when all tasks are of (more or less) the same length. But what if they're not?

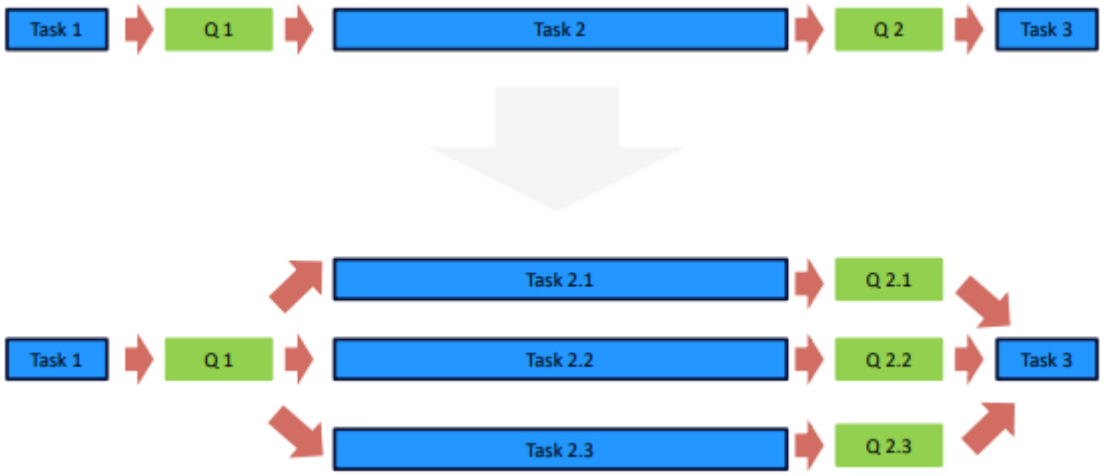
Problem: Filter stage
becomes bottleneck



Solution: Several Filter
stages to feed Display



A solution would be just to multiply tasks running the longer stage. BlockingCollection will handle it.



The parallel tasks just consume items from one blocking collection and it's all fine. But Task 3 needs to observe all collections from the previous stage. For this, there is a TakeFromAny() method of BlockingCollection.

```
private void ToLowerCase(BlockingCollection<string>[] inputs, BlockingCollection<string> output)
{
   var str = "";
   while (!inputs.All(bc => bc.IsCompleted))
   {
      BlockingCollection<string>.TakeFromAny(inputs, out str);
      str = str.ToLower();
      output.Add(str);
   }
   output.CompleteAdding();
}
```

Bear in mind, that this method will throw an exception if all collections it takes from are marked as *Complete*. It is probably because collections are marked Complete when the pipeline gets cancelled - so the method throws the exception not to cause any deadlock. But I'm not sure.

Now, how to test if actually any stage is longer?

- Test and time the filters individually – the design is very easy to test.
- Check the queue lengths during runtime from main to find the bottleneck(s).
- Replace the other filters with dummies only forwarding data, and time the whole thing.

Cancelling the pipe

In order to have the possibility to cancel the pipeline - so to cancel all the tasks - all of them should have a reference to a cancellation token. Each task should check with it if the cancellation has been requested - and, if so, gracefully stop performing.

```
void DoStage(BlockingCollection<T> input, BlockingCollection<T> output, CancellationToken token)
 try
 {
   foreach (var item in input.GetConsumingEnumerable())
   {
     if (token.IsCancellationRequested) break;
     var result = ...
     output.Add(result, token);
   }
 }
 catch (OperationCanceledException) { }
 finally
 {
   output.CompleteAdding();
 }
}
```

The first usage simply tells the task to stop iterating if the pipeline is cancelled, pretty straightforward. The second usage (in *Add*) is less obvious - but imagine, that we checked at the beginning of the loop, that cancellation was not requested. While result is computed, a pipeline gets cancelled by some other tasks. The *Add* method was called on next task's input collection, but this one may have already processed the cancellation request and closed the collection. As *Add* method is blocking, it'll wait forever, hence deadlock. *BlockingCollection.Add* with a token will be also able to detect if there was a cancellation - and, instead of waiting forever, throw an exception that can be handled.

The *CompleteAdding()* method in *finally* will complete (close) the collection associated with the current task, to have a clean exit and (possibly) avoid memory leaks.

7. Software architecture 3 main topics – process, documentation, and styles – introduce all three, details on at least one

Process

What is software architecture?

- the decisions that you wish you could get right early
- the important stuff. Whatever that is.
- it's anything and everything related to the significant elements of a software system; from the
- structure and foundations of the code through to the successful deployment of that code into a live environment.
- The architectural decisions are those that you can't reverse without some degree of effort. Or, put simply, they're the things that you'd find hard to refactor in an afternoon

How to predict the future?

You can try to get requirements of the system:

- **Functional Requirements (login, searching mechanism, recommendation mechanism)**
    - a lot of what, little of how
    - probably small impact on the architecture
    - defined as user stories or use cases
    - ex. user should be able to search for video, user should be able to login

- **Non-functional requirements (regulatory, security, performance)**
    - closer to how, than the previous
    - many of these are sensitive to the architecture
        - if they change it might have big impact on the architecture
    - often defined as quantified quality attributes
    - ex. system shall play movies to 1000 users simultaneously

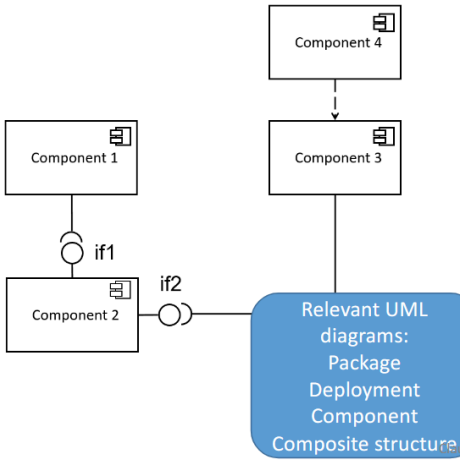
Quality attributes are also part of non functional requirements.

Architecture noun vs verb

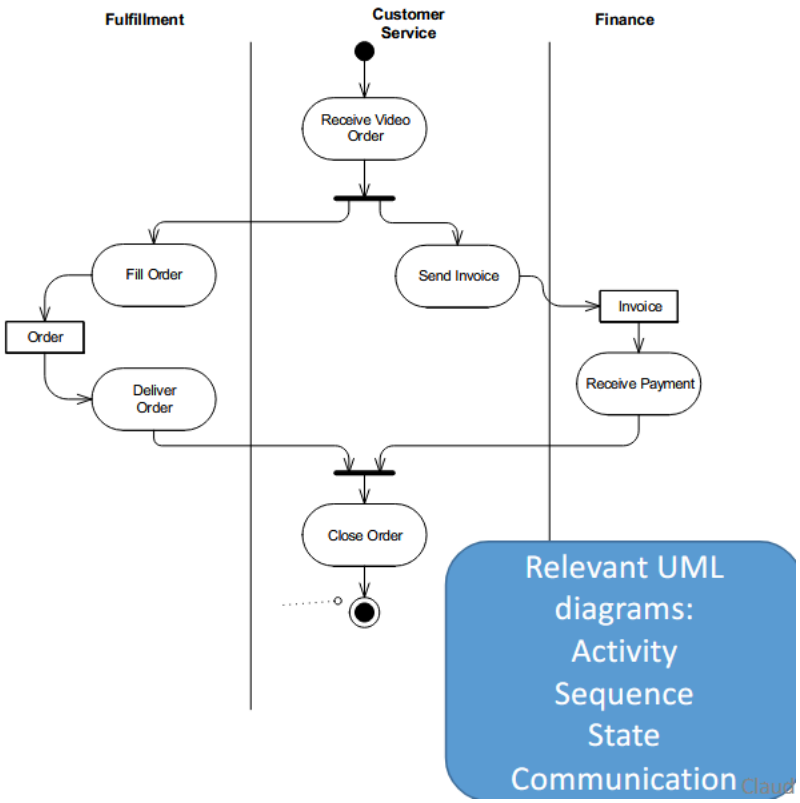
Noun

As noun the ‘architecture’ means **structure and behavior**

- What belongs together?
- What functionality depends on other functionality
- What should be exposed at the boundaries of the boxes?
- Examine the functional requirements and put functionality into ‘boxes’ or modules.
- The SOLID principles also apply at the architecture level. We’re just talking modules and not classes.
- High Cohesion, low coupling



- It’s important also to model flow and software behavior, not only the structure
  - How does data flow between the modules?
  - What is the flow through the software?

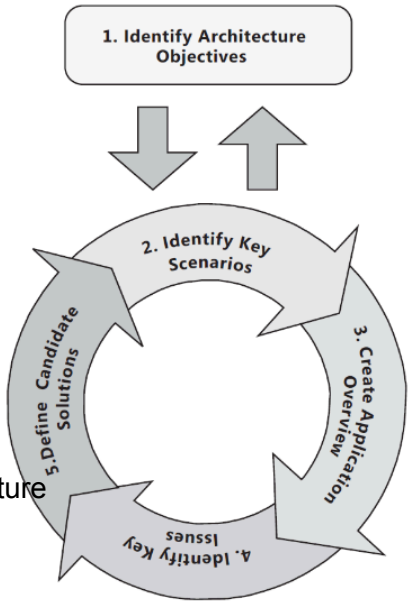


Verb

As verb ‘to architecture’ means **a process -** it means how to architect

- architecture process should be **iterative** and **incremental**
- we should use software development methodologies like RUP, SCRUM and with these methodologies design (architect) our systems

Microsoft architecture process:



1. Identify Architecture objectives
    a. functional and nonfunctional requirements
    b. cross-cutting concerns
2. Identify key scenarios
    a. Critical functionality.
    b. Critical non-functional reqs
    c. Exploration (unknown areas)
    d. Risk mitigation
    e. Look for intersections between the
    f. user, business and system views.
3. Create an application overview
    a. application overview - one or more proposals for an architecture
    b. Determine your application type.
    c. Determine relevant technologies
    d. Identify important architectural design styles.
4. Identify Key Issues
    a. Key issues are Problems that must be solved with this (version of the) architecture
    b. Key Scenarios, are they solved?
    c. Pose relevant hypothetical future changes
        i. Can I swap from one third party service to another?
        ii. Can I add support for a new client type?
        iii. Can I quickly change my business rules relating to billing?
5. Define Candidate Solutions
    a. Propose candidate solutions to key issues
    b. Evaluate against “baseline” architecture
        i. Does this architecture succeed without introducing any new risks?
        ii. Does this architecture mitigate more known risks than the previous iteration?
        iii. Does this architecture meet additional requirements?
        iv. • Does this architecture enable architecturally significant use cases?
        v. • Does this architecture address quality attribute concerns?
        vi. • Does this architecture address additional crosscutting concerns?

Architectural Style

Architectural Style - a family of systems in **terms of a pattern of structural organization.** The components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined

Categories based on focus area

Mostly concerned with code

Mostly concerned with (physical) structure

Structure

Layered, Component-based, Object Oriented

Domain

Domain Driven Design

Communication

Message Bus, SOA, *Event driven, CQRS*

Deployment

Client/Server, N-Tier / 3-Tier, *Microservice, Serverless*

[MS AAG] (+ additions)

The architecture of a software system is almost never limited to a single architectural style but is often a combination of architectural styles that make up the complete system. For example, you might have a message bus design composed of services developed using a layered architecture approach.

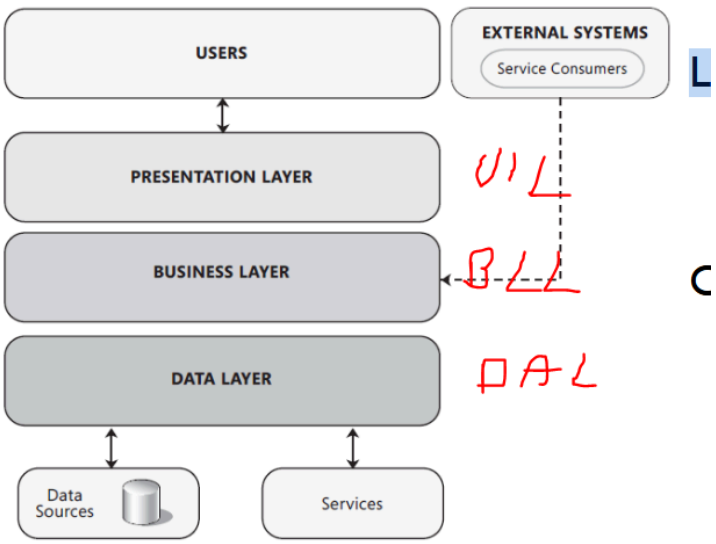
Layers

Partitions the concerns of the application into stacked groups (layers) of: classes / packages / modules / subsystems. Dependencies are only allowed from higher layer to lower layer.

Don't get confused – this doesn't mean that data cannot flow up! Decoupling is important – e.g. using events. The lower layer should work without the higher layer

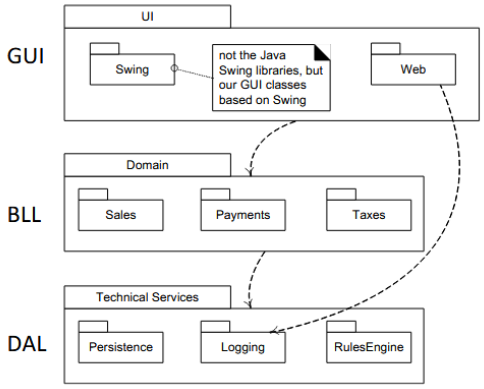
Layer vs. Tier

Layers is a logical separation. Tier represent a physical separation



Layered architecture examples

MVC – Model-View-Control
MVVM – Model-View-ViewModel
Boundary-Control-Domain/Entity
Network Layers (ISO OSI model)



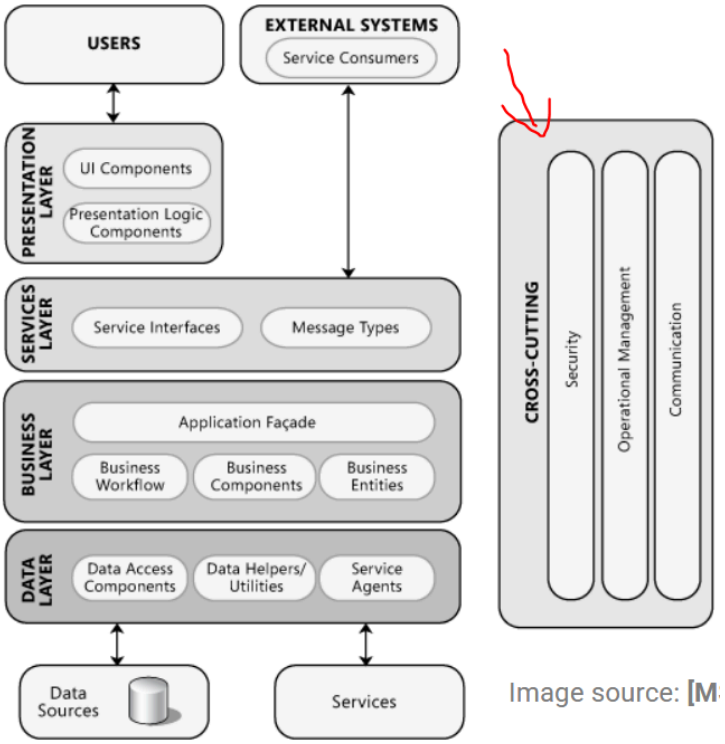


Image source: [M

Benefits of Using Layer

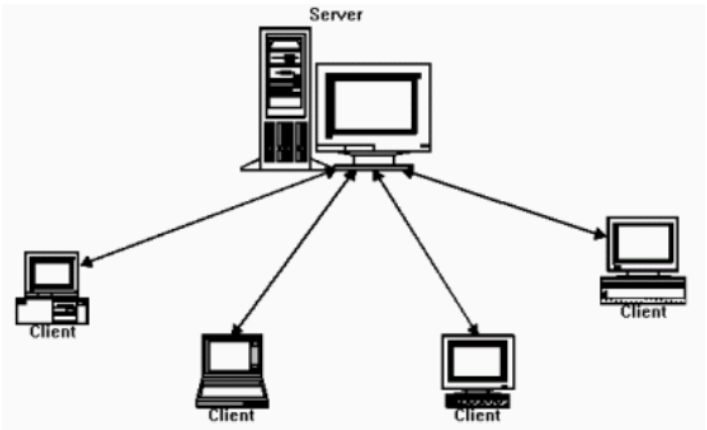
Separation of concerns (SRP)
- separation of application-specific from general services.
- separation of high-level from low-level services

Reduces coupling and dependencies
Improves cohesion
Increases potential reuse
- Lower layers can easily be reused in other applications

Increases clarity
Concurrent development by teams is aided by the logical segmentation
A layer can be replaced
- if interfaced based programming and dependency injection is used

Testing is easier

Client-server (2-tier architectural style)



The clients and servers are connected by some kind of network.
Multiple clients connect to the same server: **Many to One** The Client and the Server have different roles
The clients can vary from **thin to thick clients** based on how much they must process themselves

The server can

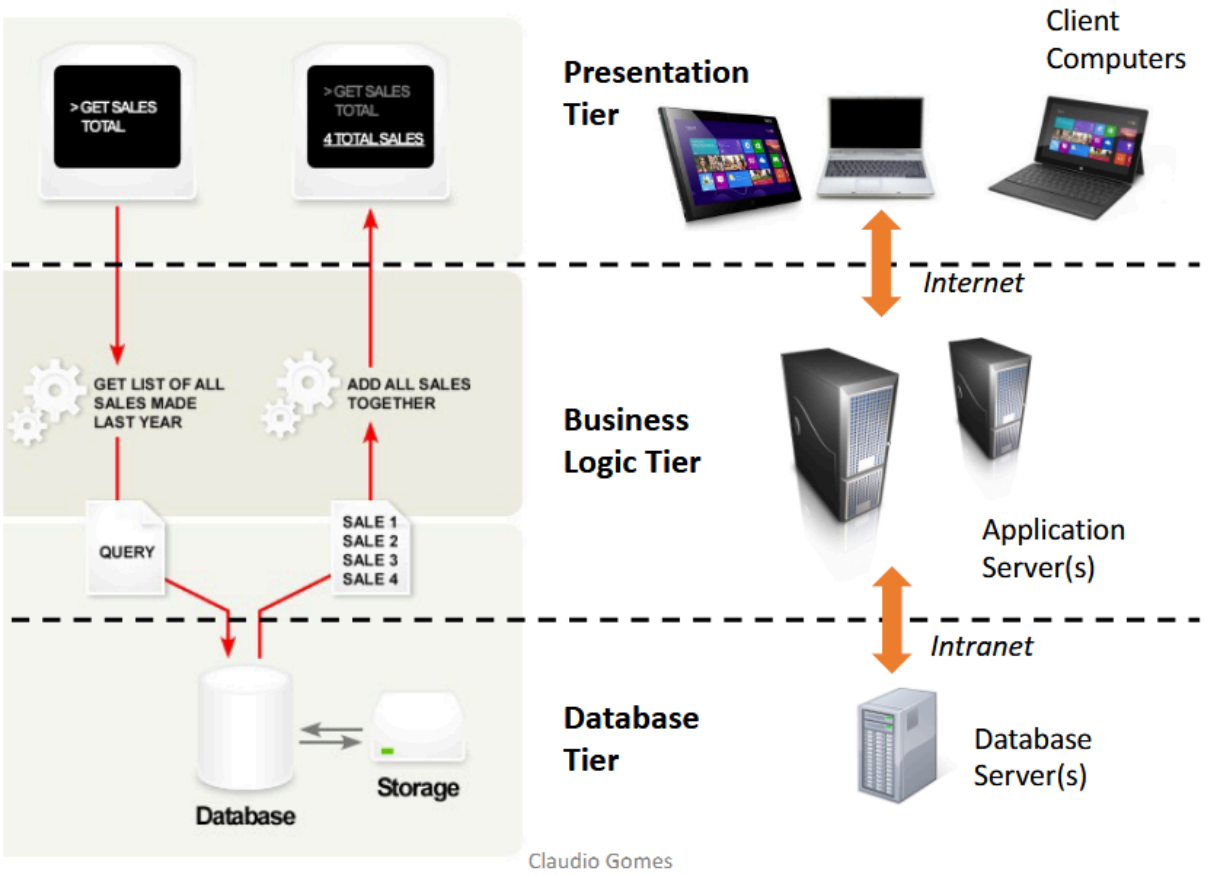
- Handle the request
- Save and serve data
- Calculate or process data

The client can

- Handle user input
- Assemble the request
- Receive the data
- Process and/or display data

N-tier

Is a client–server architecture in which presentation, application processing, and data management functions are physically separated. **The software running on a tier may itself consist of multiple layers.**



- a tier is a physical structuring mechanism for the system infrastructure.
- a tier executes on a node (a processing unit)

In General N-tier Architecture is more complex to build compared to 2-tier Architecture. Communication between tiers has to be defined, developed, and evolved. Servers may crash and communication may fail or be slow. Security has to be considered at all boundaries and also in the communication.

Pipes and filters

Message bus

8. Visitor Pattern

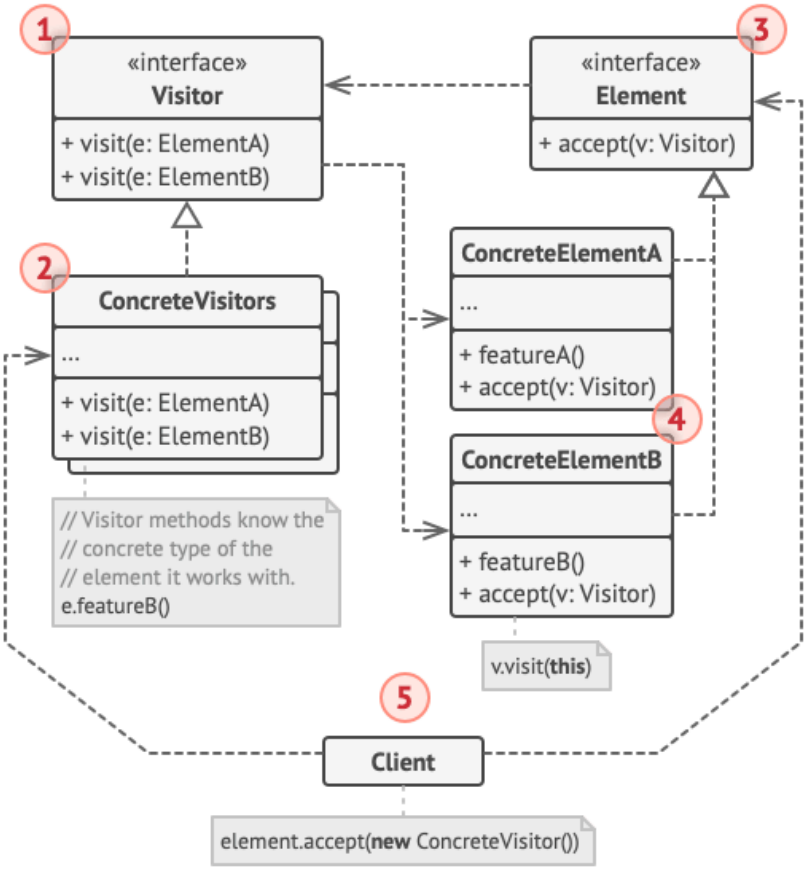
Vistor

The chosen pattern from the list proposed has been the visitor pattern, which is commonly used in Object Oriented coding and belongs to so-called “Behavioral” patterns which are designed to define how objects interact with each other and what are the relationships between them.

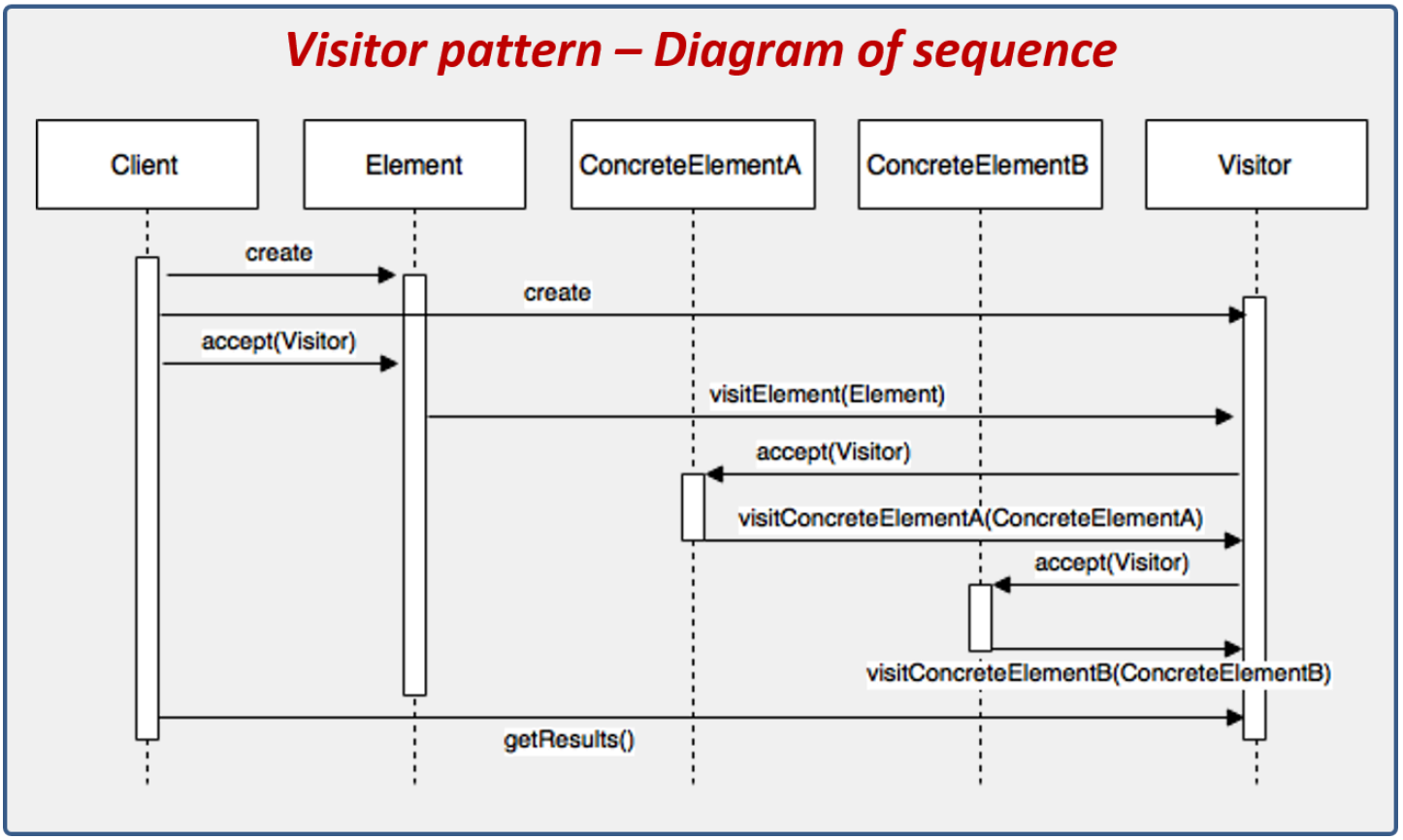
Characteristics

- **Visitor Interface:** This interface declares the methods, one for each different operation that can be performed on the elements of the object structure.
- **Concrete Visitor:** These are concrete classes that implement the Visitor interface and provide the actual implementation of the operations that were thought to perform on the objects.
- **Element Interface**: This interface declares a method, “Accept() method” that takes a visitor as an argument. Each of the elements which are supposed to perform operations over need to use this interface and therefore implement the method accept so it is possible to call a general method accept on some kind of loop and every object will accept a visitor, allowing the visitor to perform operations on the element.
- **Concrete Element:** These are concrete classes that implement the Element interface. They also implement the accept method and usually call the visitor's corresponding method within their own accept method, in order to let the visitor perform the suited operations for itself.

Class diagram



Sequence diagram

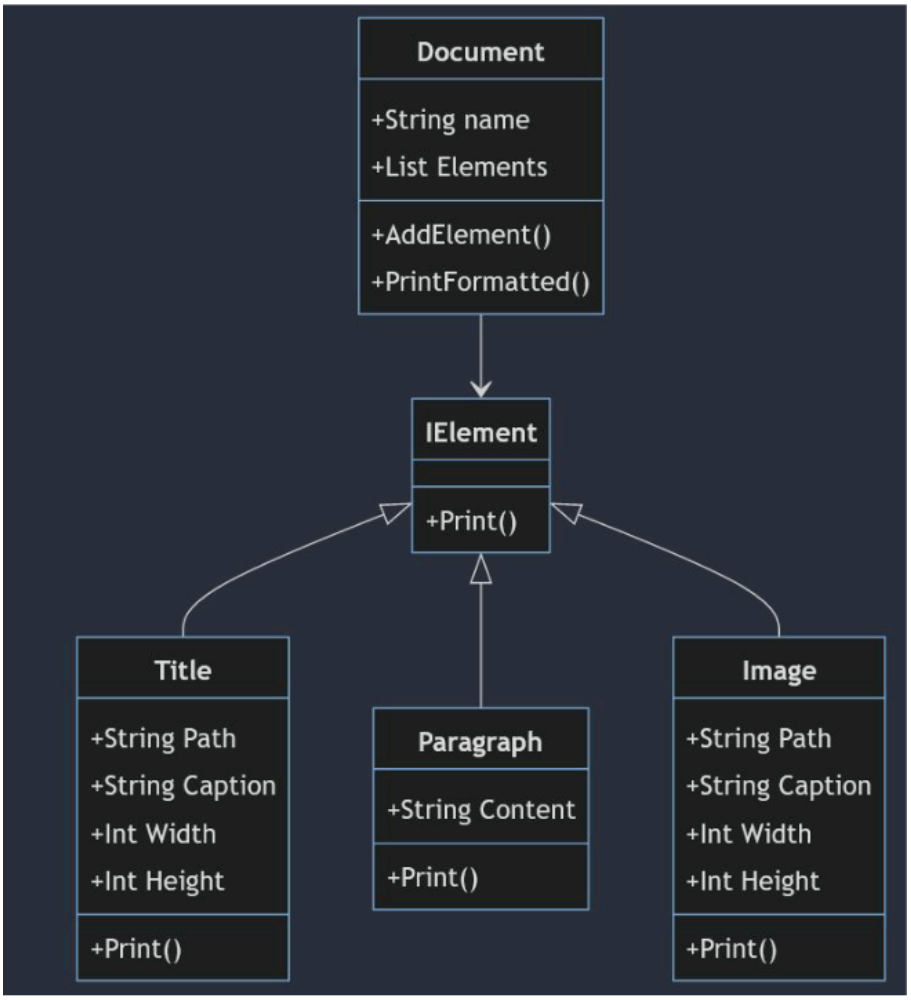


Example

Document

Document may consist of number of the following structures: Title, Table, Image, Paragraph

Diagram



Code

```
namespace DocumentBase.DocumentElements
{
    public interface IDocumentElement
    {
        void Print();
    }
}

namespace DocumentBase.DocumentElements
{
    public class Image : IDocumentElement
    {
        public string Path { get; set; }
        public string Caption { get; set; }
        public int Width { get; set; }
```

```
        public int Height { get; set; }

        public Image(string path, string caption, int width, int height)
        {
            Caption = caption;
            Path = path;
            Width = width;
            Height = height;
        }

        public void Print()
        {
            Console.WriteLine($"Image with caption: {Caption}");
        }
    }

}

namespace DocumentBase.DocumentElements
{
    public class Paragraph : IDocumentElement
    {
        string Content;

        public Paragraph(string content)
        {
            Content = content;
        }
        public void Print()
        {
            Console.WriteLine($"Paragraph with content: {Content}");
        }
    }
}

namespace DocumentBase.DocumentElements
{
    public class Title : IDocumentElement
    {
        string Content;
        public Title(string content)
        {
            Content = content;
        }
        public void Print()
        {
            Console.WriteLine($"Title with content: {Content}");
        }
```

```
    }
}

using DocumentBase.DocumentElements;

namespace DocumentBase
{
    public class Document
    {
        string Name;
        public List<IDocumentElement> Elements { get;}

        public Document(string name) {

            Name = name;
            Elements = new List<IDocumentElement>();
        }
        public void AddElement(IDocumentElement element) {
            Elements.Add(element);
        }

        public void PrintFormattedDocument()
        {
            Console.WriteLine($"Document name: {Name}\n");
            foreach (IDocumentElement element in Elements)
            {
                element.Print();
                Console.WriteLine("-----------------------------------");
            }
        }
    }
}
using DocumentBase.DocumentElements;

namespace DocumentBase
{
    class Program
    {
        public static void Main(string[] args)
        {
            Document document = new Document("SampleDocument.txt");
            Title title = new Title("Sample Title");
            Paragraph paragraph = new Paragraph("This is a sample paragraph");
            Image image = new Image("./image.png", "This is an Image caption", 200,
200);

            document.AddElement(title);
            document.AddElement(paragraph);
```

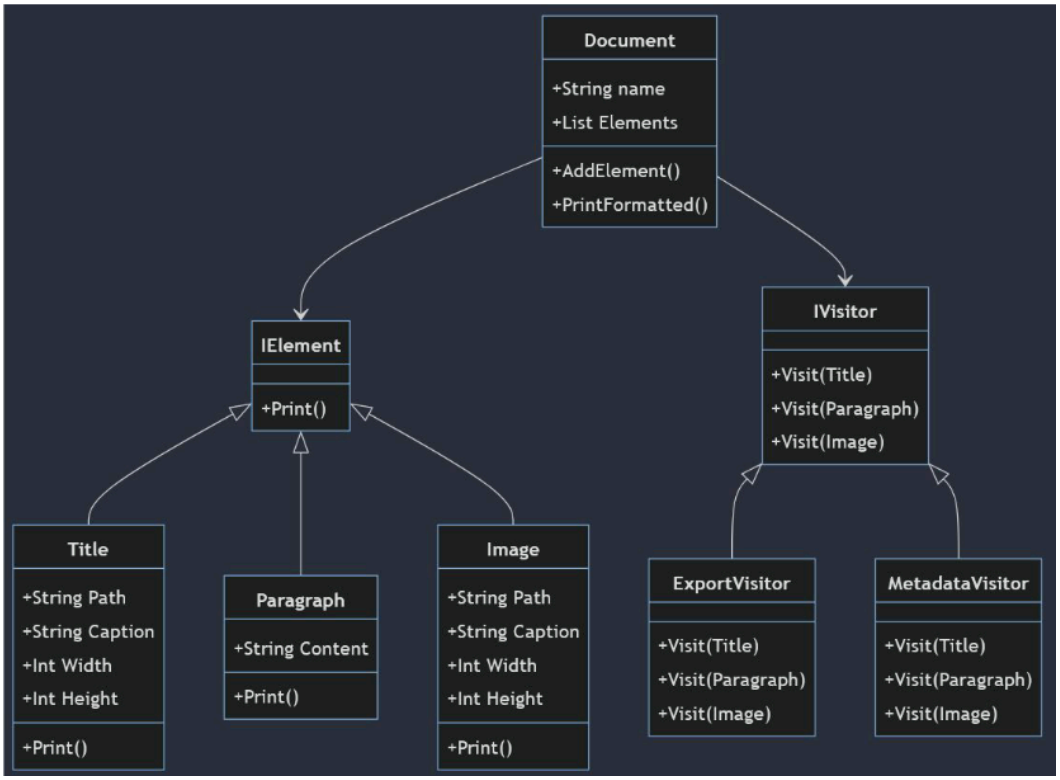
```
            document.AddElement(image);
            document.PrintFormattedDocument();
        }
    }
}
```

Now let’s consider that we want to implement a PDF Export mechanism. If we did it inside each of the ConcreteDocument we would break SRP. Let’s try to do a visitor without modification.

Document Without Modification

We add IVisitor with Visit methods for each of the ConcreteVisitors.

Diagram



Code

```
using DocumentBase.DocumentElements;

namespace DocumentVisitorWithoutModification.Visitors
{
    public interface IVisitDocumentElement
    {
        public void Visit(Title title);
        public void Visit(Image image);
        public void Visit(Paragraph paragraph);
    }
}
```

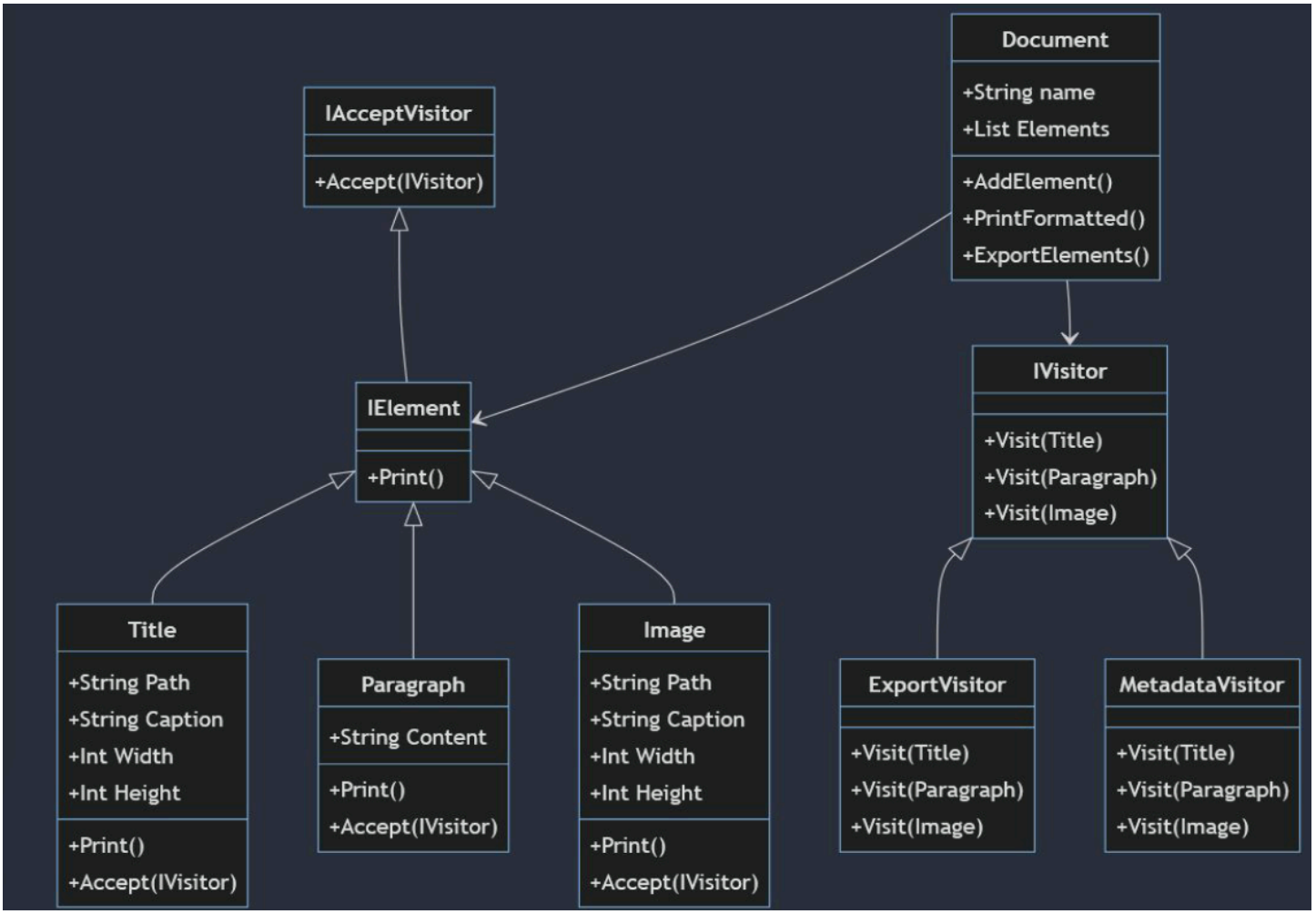
```
using DocumentBase.DocumentElements;

namespace DocumentVisitorWithoutModification.Visitors
{
    public class ExportToPdfVisitor : IVisitDocumentElement
    {
        public void Visit(Title title)
        {
            // assume using title in this function
            Console.WriteLine($"Visit Title: Exporting Title to PDF file");
        }
        public void Visit(Paragraph paragraph)
        {
            // assume using paragraph in this function
            Console.WriteLine($"Visit Paragraph: Exporting Paragraph to PDF file
successfull");
        }
        public void Visit(Image image)
        {
            // assume using image in this function
            Console.WriteLine($"Visit Image: Image to PDF file");
        }
    }
}

using DocumentBase.DocumentElements;
using DocumentVisitorWithoutModification.Visitors;

namespace DocumentBase
{
    public class Document
    {
        String Name;
        List<IDocumentElement> Elements { get; }

        public Document(string name)
        {
            Name = name;
            Elements = new List<IDocumentElement>();
        }
        public void AddElement(IDocumentElement element)
        {
            Elements.Add(element);
        }

        public void ExportAllDocumentElementsToPDF()
```

```
        {
            IVisitDocumentElement visitor = new ExportToPdfVisitor();

            foreach (IDocumentElement element in Elements) {
  // there is no way to invoke this way, that's why this implementation is bad
 //  visitor.Visit(element);
 // we need to do it like that
                if(element is Title)
                {
                    visitor.Visit((Title)element);
                }
                else if (element is Paragraph)
                {
                    visitor.Visit((Paragraph)element);
                }
                else if(element is Image)
                {
                    visitor.Visit((Image)element);
                }

 // Pros:
 // - We did not touch anything in IDocumentElement, Image, Paragraph or Title
 // Cons:
 // - Each time we add new DocumentElement we need to edit this function and check
type
 // - Using general type (Interface IVisitDocumentElement) didn't help cause we had
to cast it anyway
 // - We break OCP and probably SRP (we could make seperate class for PDF generation)

            }
        }
        public void PrintFormattedDocument()
        {
            Console.WriteLine($"Document name: {Name}\n");

            foreach (IDocumentElement element in Elements) {
                element.Print();
                Console.WriteLine("-----------------------------------");
            }
        }
    }
}
```

Now let’s proceed to do it with class modification.

Document With Modification

Diagram



We could also embed Accept method inside IElement

Code

```
using DocumentVisitorWithoutModification.Visitors;

namespace DocumentVisitorWithModification.DocumentElements
{
    public interface IAcceptVisitor
    {
        public void Accept(IVisitDocumentElement visitor);
    }
}

using DocumentVisitorWithModification.DocumentElements;

namespace DocumentVisitorWithoutModification.DocumentElements
{
    public interface IDocumentElement : IAcceptVisitor
    {
```

```
        void Print();
    }
}


using DocumentVisitorWithoutModification.Visitors;

namespace DocumentVisitorWithoutModification.DocumentElements
{
    public class Image : IDocumentElement
    {
        public string Path { get; set; }
        public string Caption { get; set; }
        public int Width { get; set; }
        public int Height { get; set; }

        public Image(string path, string caption, int width, int height)
        {
            Caption = caption;
            Path = path;
            Width = width;
            Height = height;
        }

        public void Print()
        {
            Console.WriteLine($"Image with caption: {Caption}");
        }

        public void Accept(IVisitDocumentElement visitor)
        {
            visitor.Visit(this);
        }
    }
}

using DocumentVisitorWithoutModification.DocumentElements;
using DocumentVisitorWithoutModification.Visitors;

namespace DocumentVisitorWithoutModification
{
    internal class Document
    {
        public string Name { get; set; }
        public List<IDocumentElement> Elements { get; }


        public Document(string name) {
```

```
            Name = name;
            Elements = new List<IDocumentElement>();
        }
        public void AddElement(IDocumentElement element) {
            Elements.Add(element);
        }

        public void PrintFormattedDocument()
        {
            Console.WriteLine($"Document name: {Name}\n");
            foreach (IDocumentElement element in Elements)
            {
                element.Print();
                Console.WriteLine("-------------------------------------");
            }
        }

        public void ExportAllDocumentElementsToPDF()
        {
            IVisitDocumentElement visitor = new ExportToPdfVisitor();
            foreach (IDocumentElement element in Elements) {
               element.Accept(visitor);

            }

        }

    }
}
```

We stopped breaking OCP in ExportAllDocumentElementsToPDF() method.

SOLID

Open/Closed Principle (OCP):

The Visitor Pattern aligns well with the OCP by allowing the introduction of new operations (or behaviors) without modifying the existing classes. Visitors are external classes that define new operations and can be added without changing the structure of the elements being visited.

Single Responsibility Principle (SRP):

The Visitor Pattern helps to maintain the Single Responsibility Principle by separating the concerns of the object structure and the algorithms that operate on that structure. Each visitor class is responsible for a specific behavior, promoting a clear separation of responsibilities.

Liskov Substitution Principle (LSP):

The Visitor Pattern itself doesn't directly impact the Liskov Substitution Principle. However, by allowing for the addition of new behaviors without modifying existing code, it indirectly supports the idea that subclasses should be substitutable for their base classes without affecting the correctness of the program.

Interface Segregation Principle (ISP):

The Visitor Pattern typically involves defining an interface for visitors, and concrete visitors only need to implement the methods relevant to their specific behavior. This aligns with the Interface Segregation Principle, as each visitor only needs to be concerned with the methods it requires.

Dependency Inversion Principle (DIP):

The Visitor Pattern can adhere to the Dependency Inversion Principle by relying on abstractions. The object structure being visited can define an interface that visitors depend on, allowing for flexibility and ease of extension without violating the principle.