💩

ISS

What to study

- Entity Framework

Refactoring

- PascalCase - class names and method names
  - also use them for constant names, both fields and local constants
- camelCase - method parameter and local variables
- static fields start with s_
- use meaningful and descriptive names for variables, methods and classes

Diagrams

- Use Case Diagrams
- Class Diagrams
- Sequence Diagrams
- Component Diagram (no)

Design patterns

- Singleton 😽
- Template 😽
- Strategy 😽
- Facade 😽
- Adapter 😽
- Abstract Factory 😽

- Factory 😽
- Proxy 😽
- Message Bus

Diagrams

Use Case Diagrams

- Huge rectangle with system name at the top

Actors

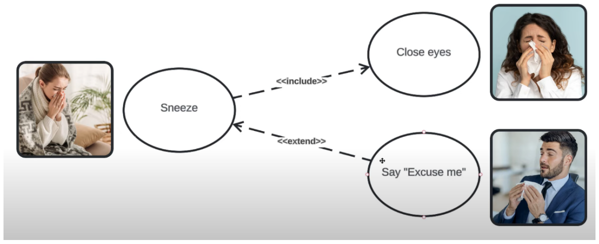
- Actor → whoever uses the system
  draw a little stickman with “customer” or whatever under as his name
  - Primary actors → on the left
  - Secondary actors → on the right

Use Cases

- Use case oval that describe an action that performs a task inside the system
  - Example: Log In, Check Balance, Transfer Funds (start with a verb)
  - put them in the logical order

Relationships

- Types: Associaton, Include, Extend, Generalization
- Association: line from Actor to Use case to show that the actor uses that function
- Include: dependancy between the base use case and the included use case
  - meaning that the base use case needs that included use



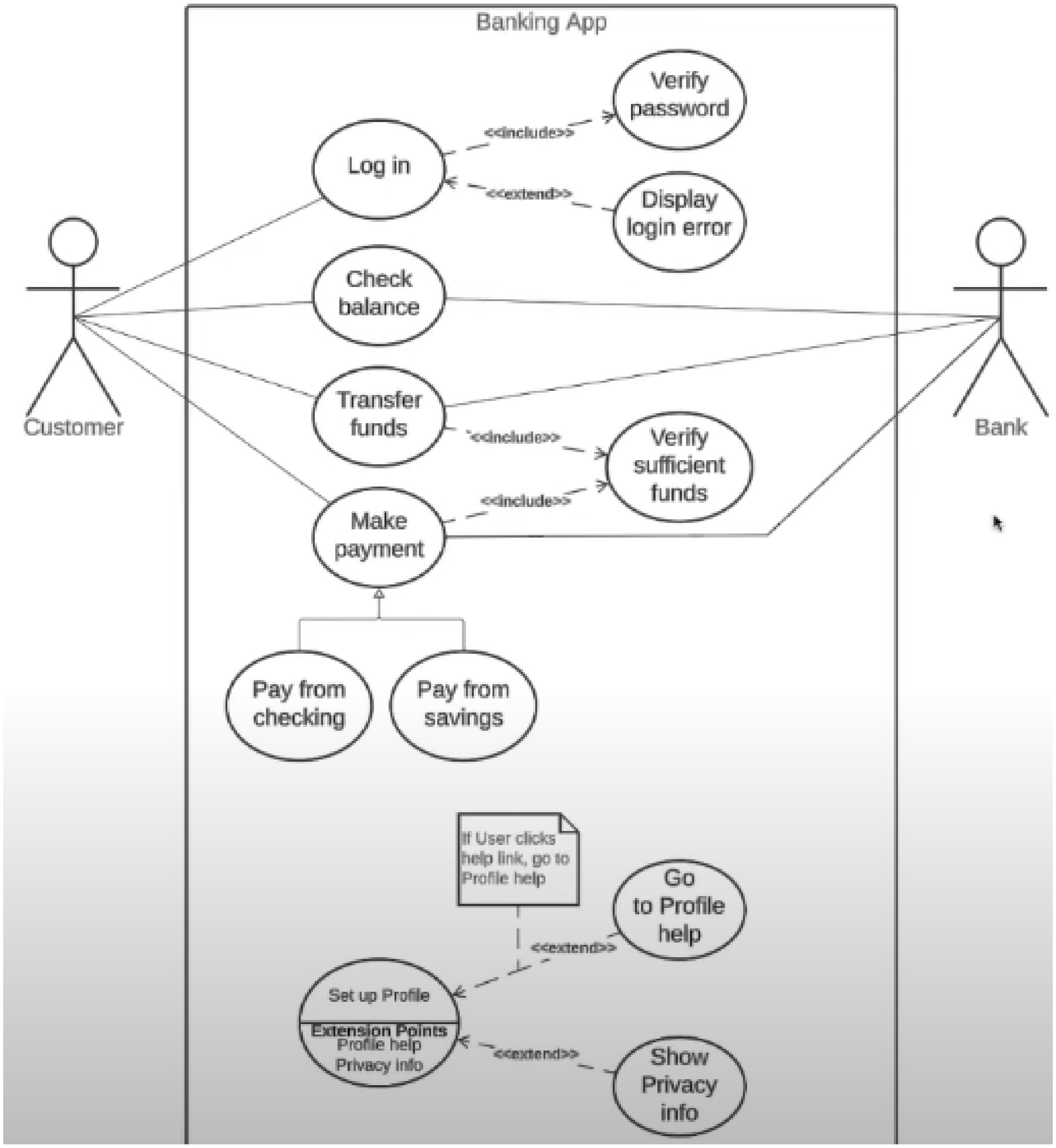
    case to be complete

    - dashed line pointing to the included use case

- Extend: extend the base use case

- Generalization: draw a straight line with a pointing arrow from the actual use case to the generalisation
    - Example: Make payment is the general use case, and the actual ones are Pay from checkings / Pay from Savings
    - can also be between actors

Example



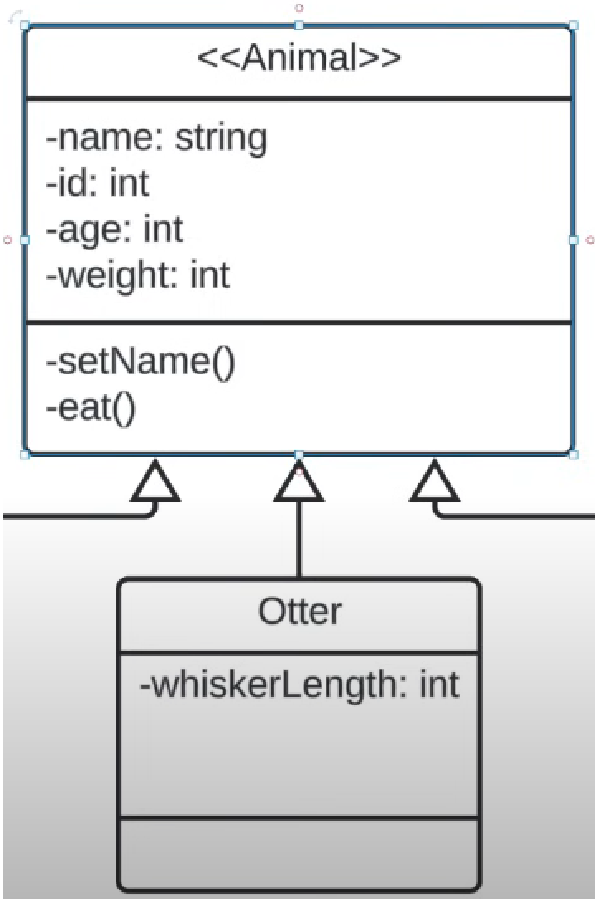
Class Diagrams

Visibility

- “-” - private
- “ + “ - public
- “ # “ - protected
- “ ~ “ - packaged (by ayone in the same package)

Inheritance

- if a class inherits from another, you point an arrow

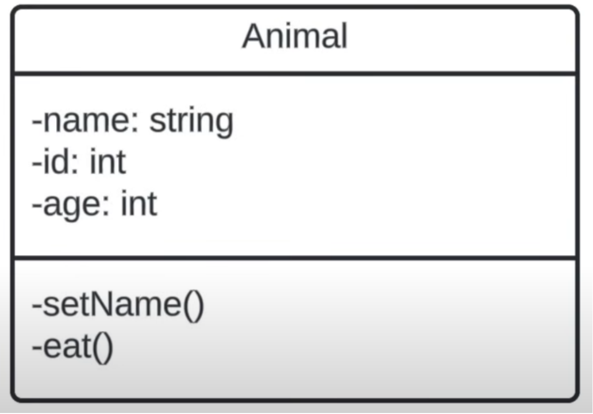


Abstraction

- if you want to make an abstract class, meaning you only dont actually want to instantiate objects of that class:
- add << >> → << Animal >>

Example - Zoo

- you create a class “Animal” for all animals
- then you add Attributes
  - visibility + name: + type
- then you add Methods
  - visibility + name(lowerCase)(variable1, var2...) + type
- then you inherit from it
  - by drawing lines to it from subclasses



Associations

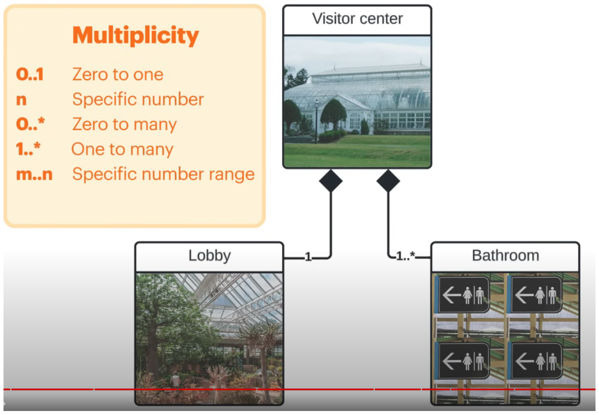
- from a class, draw a simple line to another and write what’s the relation between them
- Example: OtterClass ————eats———— SeaUrchinClass

Aggregation

- let’s say you have a Tortoise class and a Creep class (tortoises live in Creeps like lions live in packs)
- then you can do a diamon shape arrow from tortoise to creep to show that tortoise can be inside of a creep, but also outside of one

Composition

- when a child wouldnt be able to live without its parents ⇒ full diamond



Multiplicity

0..1	Zero to one
n	Specific number
0..*	Zero to many
1..*	One to many
m..n	Specific number range

Multiplicity

- when you want to show how many childs are in a parent
- Example: 1 lobby in each visitor center
- Example: 1..* 1/more bathrooms in a visitor center etc.
- Visitor Center 1————————-1 lobby ⇒ a visitor center must have exactly 1 lobby to exist, and each lobby has exactly 1 visitor center

Sequence Diagrams

- show the order of messages passed between elements, very low level

Lifeline

- vertical dotted line

Messages

- solid line → for function calls
- dotted line → for return value

Activation box

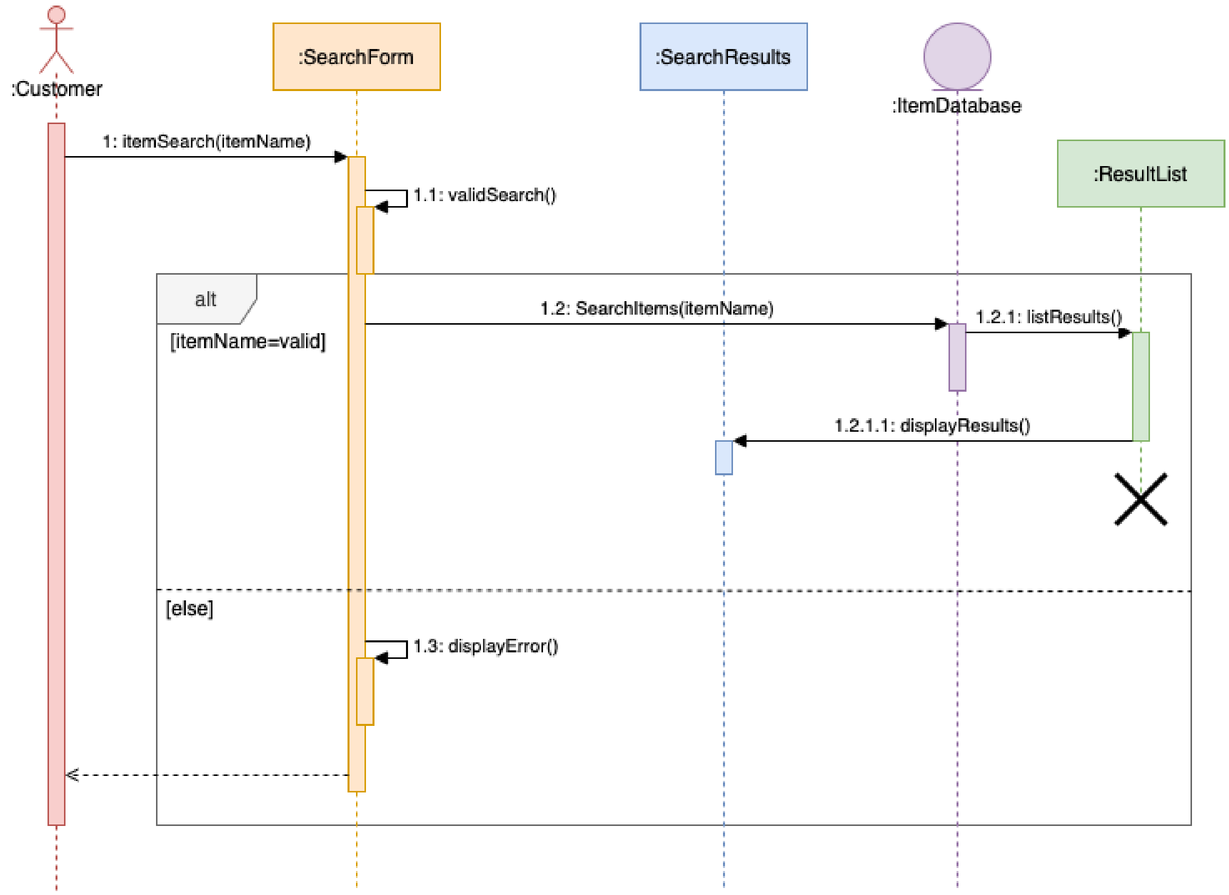
- a vertical box on the lifeline that show show long that element is used for a particular function call

X - terminate lifespan

- if an object instances is deleted before the overall sequence ends, it’s lifeline is terminated with X

Alt - box

- basically a conditional



Design Patterns

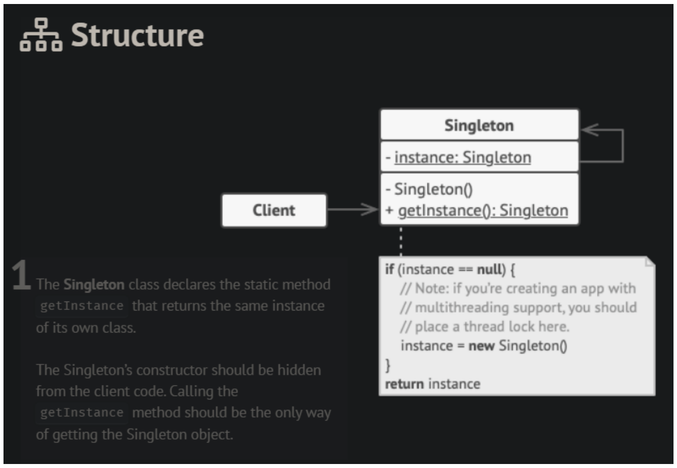
Singleton

Intent

- let’s you ensure that a class has only one instance, while providing a global access point to this instance

Solution

- make the default constructor private → no other objects can call the “new” operator inside the singleton class
- create a static creation method that acts as a constructor → this calls the private constructor to create an object and saves it in a static field; all following calls to this function return the cached object



Structure

```
if (instance == null) {
   // Note: if you're creating an app with
   // multithreading support, you should
   // place a thread lock here.
   instance = new Singleton()
}
return instance
```

1 The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

The Singleton’s constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

How to Implement

1. Add a private static field instance : Database // or whatever u want
2. Declare a public static creation method for that singleton Instance
   ```
   public static Database getInstance(){
        if (this.instance == null ){
              this. instance = new Database();
       }
   return this.instance;
   ```
3. Implement the getInstance method that creates a singleton instance on the 1st call and always returns it afterwards
4. Make the constructor of the class private (private constructor Database)
5. replace all calls to the constructor with the getInstance() function call

C# Example

```
public sealed class Singleton{
    private Singleton() { ... }
    private static Singleton _instance;
```

public static Singleton GetInstace(){

```
if (_instance ==null)
            {
                _instance =new Singleton();
            }
return _instance; }
```

public void someBusinessLogic() { … }

}

- then inside other objects/classes / code call the get instance function Singleton firstSingleton = Singleton.GetInstance();

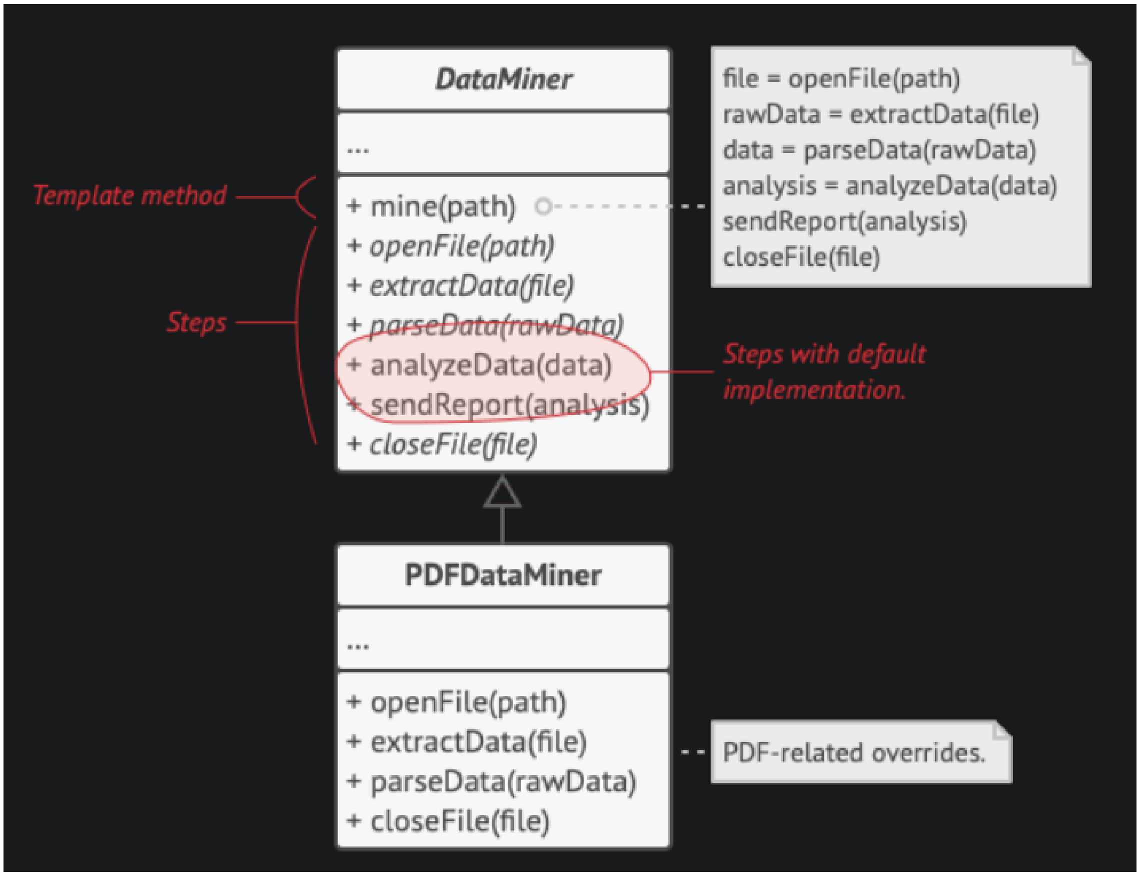
Template Method

Intent

- behavioural design pattern
- defines the skeleton of an algorithm in the superclass and let's subclasses override specific steps of the algorithm without changing it's structure

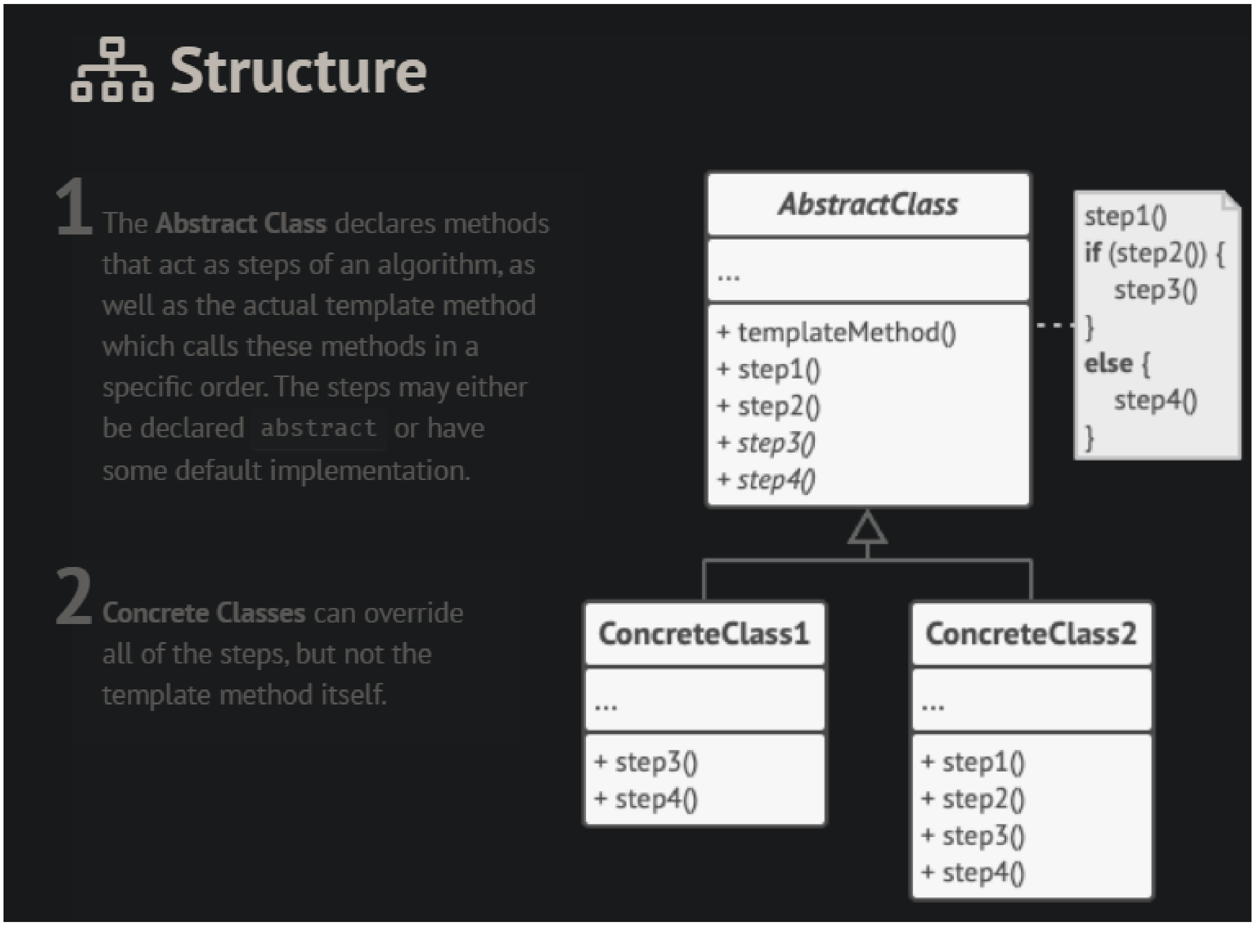
Problem

- Example: you write a DataMiner that works with corporate documents. Users feed the app documents in various formats (PDF, DOC, CSV) and then the app extracts meaningfull data
  - 1st version: supports Doc files; 2nd version: supports CSV files; 3rd one supports PDF too
  - The code for all those are entirely different in reading data from each of those different files, but it's the same for analysing and extracting data once they are read ⇒ Template Method Design Pattern is used to get rid of code duplication



Solution

- Break down an algo into a series of steps, turn these steps into methods and put a series call of all of these methods inside a Single <<Template Method>>
- Template Method contains a call to all those “step” methods; the ones that need to be implemented by subclasses are abstract and the client is required to provide a subclass of the template class and implement each of the abstract methods; even overwrite some of the other methods, but not the template one



Structure

1 The **Abstract Class** declares methods that act as steps of an algorithm, as well as the actual template method which calls these methods in a specific order. The steps may either be declared `abstract` or have some default implementation.

2 **Concrete Classes** can override all of the steps, but not the template method itself.

How to Implement

1. Analyze target algorithm and break it into steps (which steps are common and which ones are unique)
2. Create an abstract base class and declare template method as a set of abstract methods that represent the algorithms steps. Outline the algo structure in the template method by executing the corresponding steps.
3. All steps can be abstract, but some may benefit by having a default implementation
4. Add hooks ? between the crucial steps of the algorithm.
5. For each variant of the algorithm, create a new Concrete subclass. it must implement ALL of the ABSTRACT Methods (steps) and can also override some of the optional ones.

C# Example

```
abstract class AbstractClass{
    public void TemplateMethod(){
        this.BaseOperation1();
        this.BaseOperation2();
        this.Hook1();
        this.RequiredOperation1();
        this.RequiredOperation2();
    }
    protected void BaseOperation1/2();
    protected abstract void RequiredOperation1();
    protected abstract void RequiredOperation2();
    protected virtual void Hook1();
    }
```

```
classConcreteClass1 : AbstractClass
    {
protected override void RequiredOperations1()
        {
            Console.WriteLine("ConcreteClass1 says: Implemented Operation1");
        }

protected override void RequiredOperation2()
        {
            Console.WriteLine("ConcreteClass1 says: Implemented Operation2");
        }
    }
```

Client.ClientCode(new ConcreteClass1());

- basically you override the abstract methods and leave the rest the same

Strategy

Intent

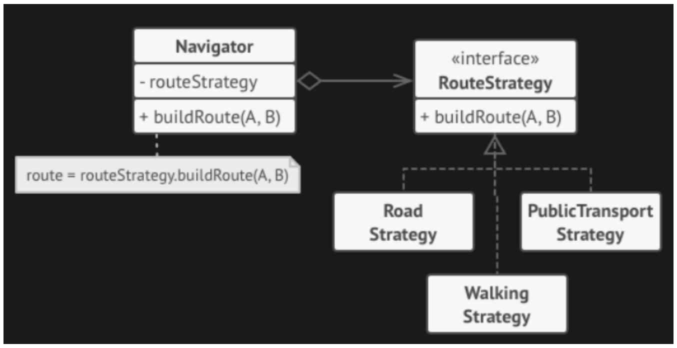
- behavioural design pattern
- let’s you define a family of algorithms, put each of them into a separate class and make their objects interchangeable

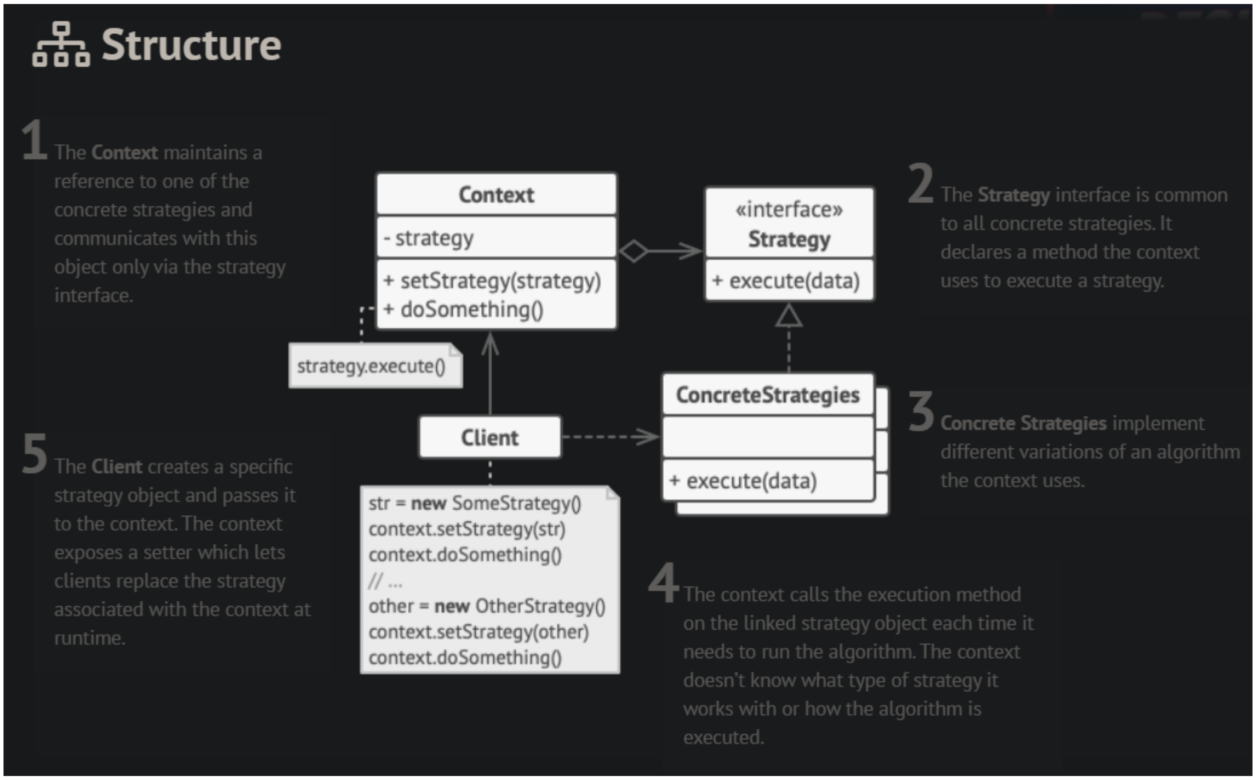
Problem

- Example: you write a class that let's tourist see a map of a city → then add a feature to get the fastest way by car from point A to point B → then by foot → then public transport → then cyclist → and so on
- You write all this code in your base class ⇒ a lot of conflicts, all code depends on each other and blabla ⇒

Solution

- you take that huge class that does something specific in a lot of ways aka get you from point A to point B and extract all of these algorithms into separate classes called
  Strategies
- The original class, called "context" must have a field for storing a reference to one of the strategies; the context delegates work to a strategy object instead of doing it himself
- the client passess the desired strategy and the context just uses an interface of all those strategies ⇒ you can add ass many strategies as you want without changing the context code





Structure

1 The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.

2 The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.

3 **Concrete Strategies** implement different variations of an algorithm the context uses.

4 The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context doesn't know what type of strategy it works with or how the algorithm is executed.

5 The **Client** creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

How to Implement

1. In the Context class, identify an algorithm that's prone to change
2. Declare a strategy interface common to all those variants of the algorithm
3. One by one, extract all those algorithms and create a new class that implements the strategy interface.
4. In the Context class, add a field that stores a strategy and a method that executes that current strategy. Also, provide a setter.
5. Clients of the context have to provide the suitable strategy that matches the way they expect the algorithm to perform.

C# Example

```
public Context(IStrategy strategy)
          {
this._strategy = strategy;
          }
```

```
        // Usually, the Context allows replacing a Strategy o
bject at runtime.
publicvoid SetStrategy(IStrategy strategy)
        {
this._strategy = strategy;
        }

        // The Context delegates some work to the Strategy ob
ject instead of
        // implementing multiple versions of the algorithm on
its own.
public void DoSomeBusinessLogic()
        {
            Console.WriteLine("Context: Sorting data using th
e strategy (not sure how it'll do it)");
varresult =this._strategy.DoAlgorithm(new List<string> { "a",
"b", "c", "d", "e" });

string resultStr =string.Empty;
foreach (varelementin resultas List<string>)
            {
                resultStr += element + ",";
            }

            Console.WriteLine(resultStr);
        }
```

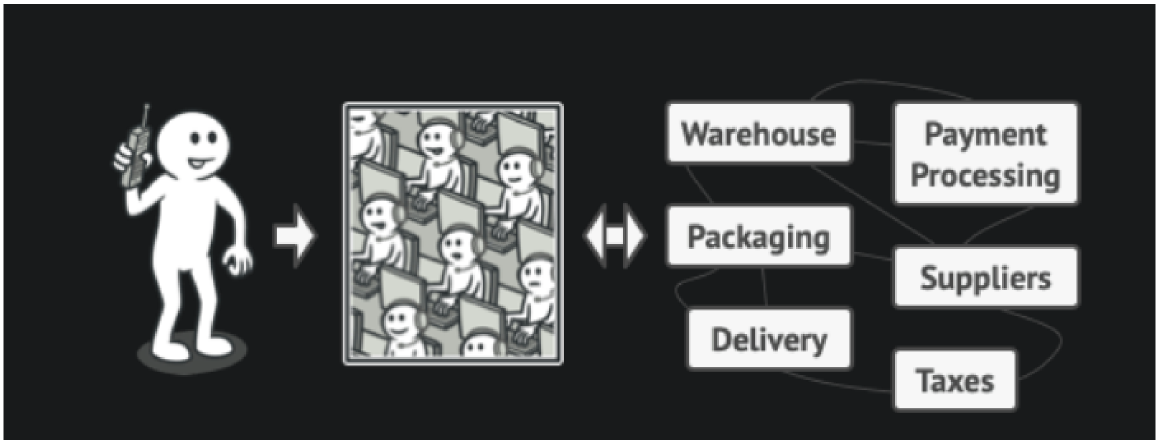
Facade

Intent

- structural design pattern
- provides simplified interface to a library, a framework or any complex set of classes

Problem

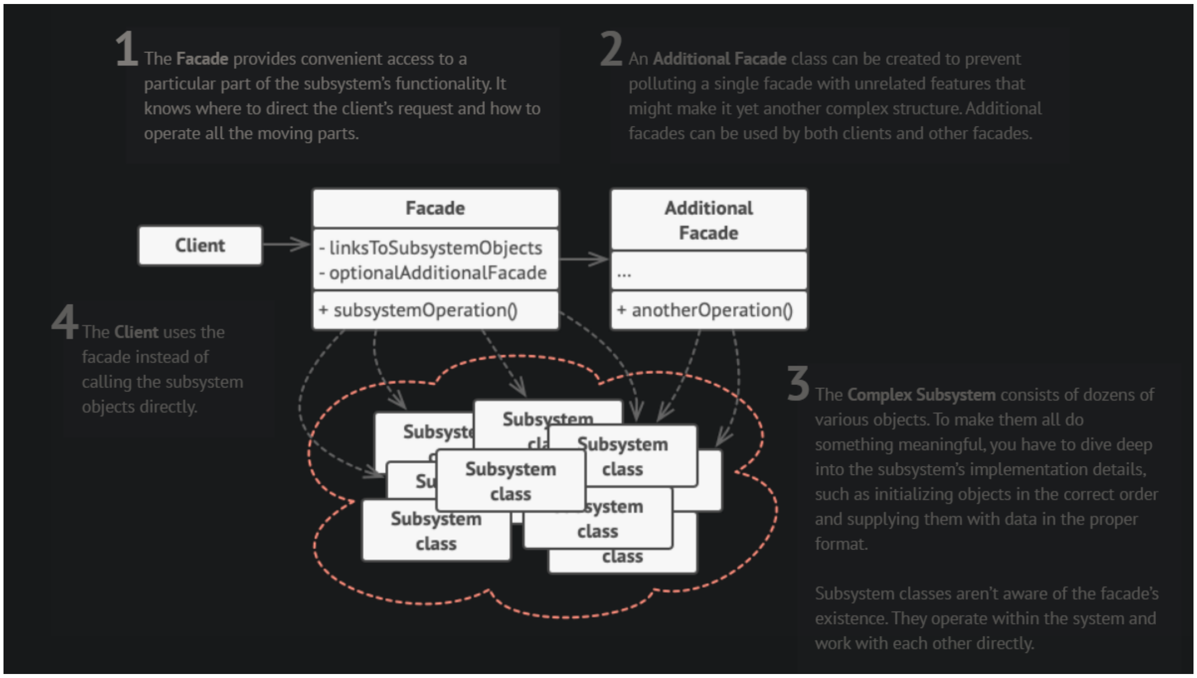
- Real World ANALogy



Placing orders by phone.

Solution

- a class that provides a simple interface to a complex subsystem which contains a lot of moving parts; it provides limited functionally compared to working directly with the subsystem, as it includes only the features required by the client



1 The **Facade** provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts.

2 An **Additional Facade** class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Additional facades can be used by both clients and other facades.

3 The **Complex Subsystem** consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem's implementation details, such as initializing objects in the correct order and supplying them with data in the proper format.

Subsystem classes aren't aware of the facade's existence. They operate within the system and work with each other directly.

4 The **Client** uses the facade instead of calling the subsystem objects directly.

How to Implement

1. Check whether it's possible to provide a simpler interface than what an existing subsystem already provides ⇒ purpose is to make the client code independent from many of the subsystem classes
2. Declare and implement this interface as a facade class. This Facade class → redirects client calls to the appropriate subsystem. Facade is responsible with initiating the subsystem and mananging its life span
3. Make all the client code communicate with the subsystem only via facade. now client code is protected from changes in the subsystem
4. Facade too big → extract a part of it to new facade

C# Example

```
public classFacade
    {
                    protected Subsystem1 _subsystem1;

                    protected Subsystem2 _subsystem2;

                    public Facade(Subsystem1 subsystem1, Subs
ystem2 subsystem2)
        {
                        this._subsystem1 = subsystem1;
                        this._subsystem2 = subsystem2;
        }

        // The Facade's methods are convenient shortcuts to t
he sophisticated
        // functionality of the subsystems. However, clients
get only to a
        // fraction of a subsystem's capabilities.
                    public string Operation()
        {
                        string result = "Facade initializes s
ubsystems:\n";
            result +=this._subsystem1.operation1();
            result +=this._subsystem2.operation1();
            result += "Facade orders subsystems to perform th
e action:\n";
            result +=this._subsystem1.operationN();
            result +=this._subsystem2.operationZ();
                        return result;
        }
    }
```

Adapter

Intent

- structural design pattern
- allows objects with incompatible interfaces to collaborate

Problem

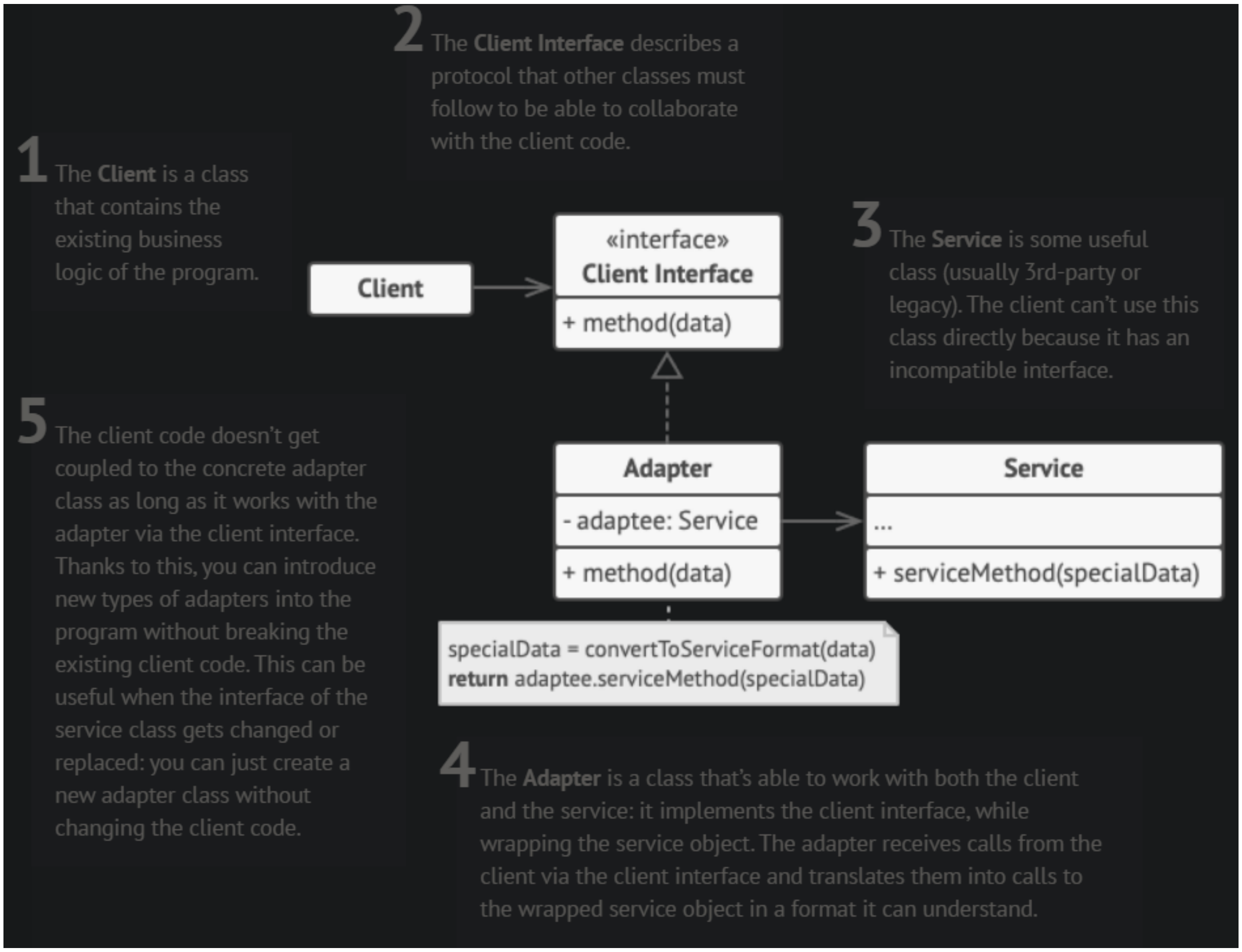
- Example: you have an app that takes XML files and works with them and the u want to integrate a 3rd party library to analyse them further. problem → that library works with JSON files

Solution

- create an Adapter = an object that converts the interface of one object so that another one can understand it
- the Adapter wraps one object to hide the complexity of conversion happening behind the scenes; the object won't even know it's wrapped in an adapter;
- Adapters can not only convert data, but can also help objects with different interfaces collaborate. How it works:
    1. adapter gets an interface compatible with one of the objects
    2. using this provided interface, the adapter can use all the methods of that object

3. upon receiving a call, the adapter passes the request to the 2nd object, but in a format and order that the 2nd object understand

- you can also create adapters that work both ways



1 The **Client** is a class that contains the existing business logic of the program.

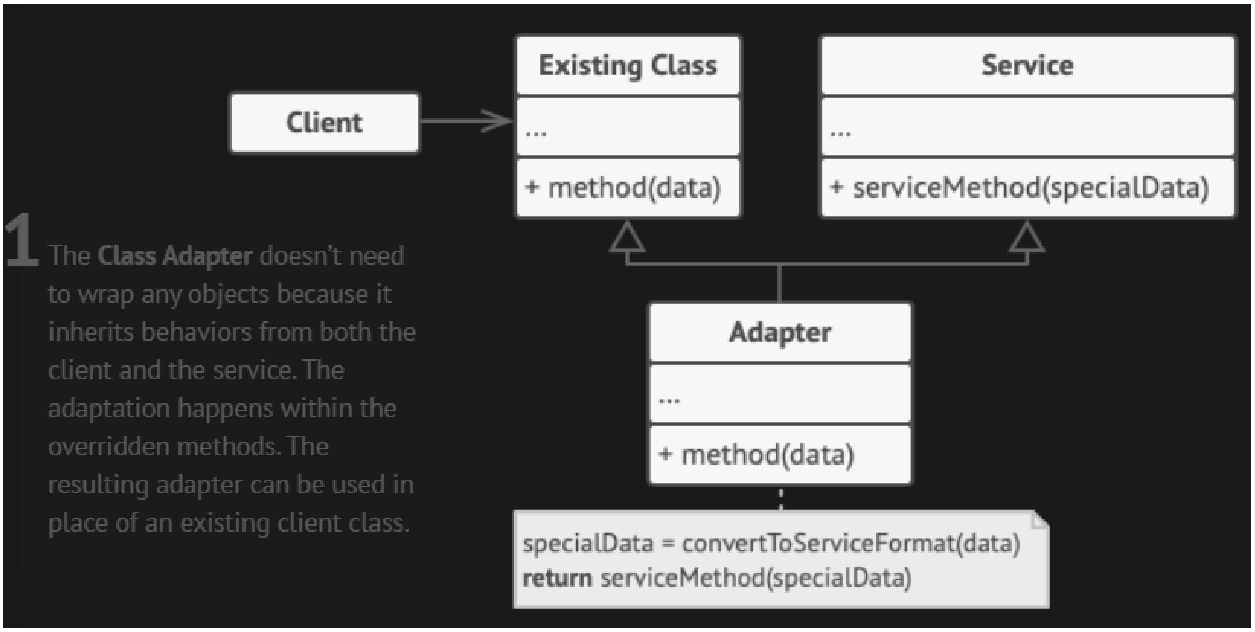
2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

```
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)
```

4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.


Client
Existing Class
...
+ method(data)
Service
...
+ serviceMethod(specialData)
Adapter
...
+ method(data)
specialData = convertToServiceFormat(data)
return serviceMethod(specialData)

1 The **Class Adapter** doesn't need to wrap any objects because it inherits behaviors from both the client and the service. The adaptation happens within the overridden methods. The resulting adapter can be used in place of an existing client class.

How to Implement

1. Identify 2 classes with incompatible interfaces
2. Declare client interface and describe how it communicates with the client
3. Create adapter class and make it follow the client interface;
4. Add a field that stores a reference to the service object
5. One by one, implement all methods of the client interface in the adapter class; the adapter should delegate most of its work to the service, handling only the data format conversion
6. Clients should use the adapter via the client interface

C# Example

```
public interface ITarget
    {
                string GetRequest();
    }

    // The Adaptee contains some useful behavior, but its int
erface is
```

```
    // incompatible with the existing client code. The Adapte
e needs some
    // adaptation before the client code can use it.
classAdaptee
    {
                public string GetSpecificRequest()
        {
                        return "Specific request.";
        }
    }

    // The Adapter makes the Adaptee's interface compatible w
ith the Target's
    // interface.
classAdapter : ITarget
    {
                private readonly Adaptee _adaptee;

                public Adapter(Adaptee adaptee)
        {
                    this._adaptee = adaptee;
        }

                publicstring GetRequest()
        {
                        return $"This is '{this._adaptee.GetS
pecificRequest()}'";
        }
    }

classProgram
    {
                static void Main(string[] args)
        {
            Adaptee adaptee =new Adaptee();
            ITarget target =new Adapter(adaptee);
            Console.WriteLine("Adaptee interface is incompatible with the client.");
            Console.WriteLine("But with adapter client can call it's method.");

            Console.WriteLine(target.GetRequest());
        }
    }
```

Abstract Factory

Intent

Problem

Solution

How to Implement

C# Example

Factory Method

Intent

- provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Problem

Solution

How to Implement

C# Example

Proxy

Intent

Problem

Solution

How to Implement

C# Example